THE ETC: AN ALL-SKY MONITOR OF CELESTIAL OPTICAL FLASHES

by
ROLAND KRAFT VANDERSPEK

A.B., University of California, Berkeley (1980)

SUBMITTED TO THE DEPARTMENT OF PHYSICS
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS OF THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December, 1985

© Massachusetts Institute of Technology, 1985
DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

Some pages in the original document contain text that runs off the edge of the page.

* This disclaimer pertains to the Appendix A-C section of this thesis.
To all Vanderspeks,
Past, present and future.
Webster's New Collegiate Dictionary defines **plenary** as "complete in every aspect: absolute, unqualified."

Roget's International Thesaurus, Third Edition, adds:

(56.12)

full, filled, replete, ample, good, plump, **plenary**, pleny [naut.], pregnant, flush, round; brimful, brimming; chock-ful, chuck-ful, choke-ful, chug-ful [dial.], chock or chuck [coll.], cram-ful, top-ful; jam-ful, cramp-ful, cram-jam-ful, jam-packed, pack-jammed, jam-up, full-up [all slang]; stuffed, packed, crammed; bursting, ready to burst, fit to bust [slang]; as full as a tick, as full as a vetch, as full as an egg is of meat, packed like sardines or herrings; replete with, crawling or oozing with; saturated, capacitated [coll.]; congested, overful.
The Explosive Transient Camera: 
An All-Sky Monitor for Optical Flashes 
by
ROLAND KRAFT VANDERSPEK

Submitted to the Department of Physics on November 14, 1985, in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Physics

ABSTRACT

The discovery in 1973 (Klebesadel, et al., 1973) of the phenomenon of gamma-ray bursts (GRBs), seen as short (durations of 1-60 seconds), sudden (risetimes of less than a second) outbursts of γ-rays from deep space, has led to intense efforts to discover the source of these mysterious emissions. Observations in the last ten years with a series of interplanetary and terrestrial satellites have led to hundreds of detections of GRB events. Analyses of observational data support the hypothesis of a highly-magnetized ($10^{12-13}$ Gauss) neutron star as the source of GRBs, yet the low precision of localization of most GRBs (tens of arc-minutes to degrees) has hindered the a posteriori identification of a quiescent counterpart to a GRB source in any energy band. To date, no convincing quiescent optical counterparts to GRB sources have been established. The discovery by Schaefer (1981) of transient optical radiation from a small GRB error region, recorded on an archived photographic plate in 1928, led to the hope the precision of localization of GRB sources might be greatly improved through the detection of optical radiation emitted during the GRB.

In 1982, the Explosive Transient Camera (ETC), a wide-field sky monitor sensitive to celestial optical flashes with risetimes of the order of one second, was proposed as a ground-based counterpart to gamma-ray satellites with the expressed intent of detecting optical radiation from outbursting GRBs (Ricker, et al., 1983). In 1983, construction was begun of a sub-unit of the plenary ETC, designed to test the feasibility of a full wide-field ETC. This thesis discusses the motivation, design, construction and implementation of the ETC test unit. Calculations of estimated event rates from several known sources of celestial optical transients in the plenary ETC are presented. In addition, this thesis includes the presentation and discussion of results from observations made the test unit, which comprise the most complete wide-field search for celestial optical flashes to
date. The observations with the ETC test unit covered a solid-angle-time product of 3.0 steradian-hours and included the error regions of GRB1200+21 (24 November 1978), GRB1152+20 (1 January 1979) and GRB1140+20 (2 May 1979) (Baity, et al., 1984) as well as the flare stars V475 Her, Ross 867 and Ross 868 (Gurzadyan, 1980). The observations were expected, based on assumptions presented within, to have detected optical transient events from 1.5 flare stars and 0.008 GRBs. These observations resulted in the determination of a new upper limit on the celestial optical flash rate of 2.2 optical flashes per hour per steradian at 10th magnitude, lower by a factor of 10 than the previous best upper limit determined by Schaefer, Vanderspek, Bradt and Ricker (1984).

Thesis Supervisor: Dr. George R. Ricker

Title: Senior Research Scientist
TABLE OF CONTENTS

Chapter 1: Introduction ................................................. 1
Chapter 2: Motivation .................................................. 4
Chapter 3: Theories of Radiation from GRBs ..................... 20
Chapter 4: The Concept of the Explosive Transient Camera ...... 46
Chapter 5: ETC Operations .......................................... 58
Chapter 6: The ETC Test Instrument ............................... 65
Chapter 7: The ETC Overseer Computer ......................... 76
Chapter 8: The ETC Trigger Processor ............................ 91
Chapter 9: The Instrument Control Electronics ................. 106
Chapter 10: Expected Results ................................... 116
Chapter 11: Results of Observations with the ETC ............ 131
Chapter 12: Future Work ......................................... 155
Chapter 13: Acknowledgements .................................. 161
References .......................................................... 165
Appendix A: The Overseer Computer Software .................. 169
Appendix B: The Trigger Processor Software ................... 170
Appendix C: The Instrument Control Electronics Software ... 171
Appendix D: ETC-specific Driver Code ............................ 172
Appendix E: ETC Computational Algorithms ..................... 173
Appendix F: Thermal Analysis of the ETC ....................... 175
CHAPTER 1

Introduction

In 1969, orbiting γ-ray satellites intended to detect γ-rays from nuclear explosions in near-Earth orbit detected unusual, sudden flashes of γ-rays from deep space (Klebesadel, et al., 1973). Analysis of the flashes showed that 1) the bursts were not created by the interaction of high-energy particles with the γ-ray detectors and 2) that the Earth or Sun were not the sources of the γ-rays, and therefore that the γ-rays were cosmic in origin. Several hundred of these flashes, dubbed gamma-ray bursts (GRBs), have been detected by balloon- and satellite-borne γ-ray detectors since 1969. Despite the large number of GRB detections, the source of GRBs has largely remained a mystery, primarily due to the low precision of localization of most GRBs.

Many of the observed characteristics of GRBs provide clues to the source, location and mechanism of GRB production. The short risetimes of many GRBs (typically 10-200 ms; one lower than 200 μs; Mazets, et al., 1981) point to a small γ-ray emitting region (<60-1000 km). This fact, combined with the detection in some GRB spectra of line features which can be interpreted as gravitationally-redshifted e⁺-e⁻ annihilation radiation (near 400 keV) and cyclotron resonance features (near 50 keV) and the detection of pulsations in the tails of a few GRB light curves suggest that a GRB originates near the surface of a highly-magnetized (10^{12} Gauss) neutron star.
Due to the large error regions of typical GRBs, this association has not been confirmed by observations of a quiescent GRB source.

In 1981, B. Schaefer of MIT discovered an optical flash on an archived photographic plate in the atypically-small error region of a known GRB (Schaefer, 1981). This discovery made clear the possibility that GRBs may emit optical radiation during outburst. Schaefer's finding is very significant, since the detection of optical radiation from an outbursting GRB would permit precise localization of the burst source, thus leading to more meaningful follow-up observations in all energy bands. Since 1981, two further archived optical flashes from GRB error regions have been found by Schaefer (Schaefer, et al., 1984), further supporting the claim that GRB sources can emit bursts of optical radiation.

In 1983, a program was initiated at MIT to design and construct an instrument capable of detecting and precisely locating optical flashes from GRBs in real time. This instrument, known as the Explosive Transient Camera (ETC), was to be a wide-field sky monitor sensitive to tenth-magnitude optical flashes with risetimes of the order of one second. The instrument would operate automatically, and would be able to provide the location of an optical flash with sub-arc-minute precision within a fraction of a second after its detection (Ricker, et al., 1984).

The design, construction and testing of the initial stage of the Explosive Transient Camera is the subject of this thesis. The following four chapters will discuss the motivation and concept of the
ETC, as well as some of the possible mechanisms for the production of optical light from GRBs. Thereafter, the ETC instrumentation will be presented in detail. Finally, estimates of event rates from known possible sources of optical flashes, both celestial and terrestrial, are presented, as well as the results of observations made with the ETC test unit. These observations, made during March and May of 1985, comprise the most sensitive wide-field search for optical flashes made to date. These observations have defined an upper limit of 2.2 optical flashes per hour per steradian at a visual magnitude of $m_v < 10$, a factor of 10 lower than the previous best limit defined by the work of Schaefer, Vanderspek, Bradt and Ricker (1984).
CHAPTER 2

Motivation

Introduction

This chapter is intended to give the reader an overview of the history and morphology of detected gamma-ray bursts. More detail on GRB morphology can be found in reviews by Mazets (1981), Cline (1983), and Hurley (1983). Following the morphology, a review of observations at quiescence of GRB sources at other energies is given, with an emphasis on follow-up work in the optical band. The chapter concludes with a comparison of present methods of searching for optical flashes from GRBs, indicating the pressing need for a dedicated all-sky monitor for optical flashes, such as the Explosive Transient Camera.

2.1. Gamma-Ray Burst Morphology

Gamma-ray bursts were first discovered in 1973 by the Los Alamos Group from data taken with Air Force Vela satellites to detect γ-rays from nuclear explosions in space (Klebesadel, et al., 1973). In the discovery paper from the Los Alamos group, a GRB was reported as generating an intense burst of gamma-rays (fluence $S > 10^{-5}$ erg cm$^{-2}$) with a risetime of a fraction of a second. Since 1969 a series of interplanetary and terrestrial satellites, (including the Vela satellites, the Soviet Venera spacecraft, the Pioneer Venus Orbiter, ISEE-3, Prognoz 7, Helios-B, IMP-6 and IMP-7) have detected more than
100 bursts (Cline, 1983; Baity, 1984).

2.1.1. Characteristics of Typical GRB

The characteristics of detected GRBs vary over a wide range: it is therefore difficult to present information about a "typical" GRB. (See Baity, et al. (1984) and references within for a complete review of GRB observations). GRBs as a group can be described by characteristics common to all bursts and the range of values of these characteristics in detected bursts. The GRBs detected to date are distributed roughly isotropically over the celestial sphere (Hurley, 1983). The peak fluxes of detected GRBs range from $10^{-2}$ to $10^{-7}$ erg cm$^{-2}$s$^{-1}$. The fluences, $S$, of all GRBs detected to date are between $10^{-3}$ and $10^{-8}$ erg cm$^{-2}$. The number of burst sources $N(>S)$ having a fluence greater than a value $S$ roughly follows a power law, $N(>S) \sim S^{-\alpha}$, where $\alpha \approx 3/2$ for values of $S > 10^{-5}$ erg cm$^{-2}$ and upper limits indicate $\alpha \approx 0.7$ for $S < 10^{-5}$ erg cm$^{-2}$. A plot of log $N(>S)$ vs. log $S$ is shown in Figure 2.1.

The light curves of most GRBs are characterized by a fast rise (risetimes of ~50-1000 ms) and an exponential decay (decay times of 1-30 seconds). Total durations of typical GRBs range from less than one second to minutes. A few GRB sources have shown periodic pulsations (periods of 4-10 seconds) during the decay of the brightness of the GRB (Mazets, et al., 1979b; Wood, et al., 1981). A few "typical" GRB light curves are shown in Figure 2.2.
Figure 2.1: The log $N(>S)$ - log $S$ curve for detected GRBs, taken from Jennings (1982). The solid lines indicate the results of disk models of GRBs with an intrinsic luminosity distribution.
Figure 2.2: Four GRB light curves. Figures (a) and (b), taken from Klebesadel (1982) and Cline (1983) show the lightcurves with the slowest and second-fastest rise to peak recorded to date. Figures (c) and (d) are two views of a GRB whose lightcurve is typical of the GRB phenomenon.
The spectra of GRBs can, in general, be fit by a power-law function, \( F(\nu) \sim \nu^{-1}e^{hv/kT} \), where \( F(\nu) \) is the flux (in \( \text{cm}^{-2}\text{s}^{-1}\text{erg}^{-1} \)) of \( \gamma \)-rays of energy \( hv \). Typical GRB temperatures (\( kT \)) fall in the range of 100 to 500 keV. The spectra show substantial time variations, presumably due to changes in the physical characteristics of the \( \gamma \)-ray source (Mazets, et al., 1981). Some GRB spectra show line features at energies in the range of 30-70 keV. Roughly 15% of all GRB spectra show an emission feature near 400 keV (Cline, 1983). A few characteristic GRB spectra are shown in Figure 2.3.

These characteristics allow one to make a rough sketch of the source of the GRB. Many workers in the field argue that GRBs are associated with neutron stars with strong (~10\(^{12}\) Gauss) magnetic fields, based on the following interpretations:

1) The short burst risetimes (~50 ms) point to a source size <10\(^9\) cm.
2) The line features near 30-70 keV can be interpreted as the cyclotron resonance features in a \( 10^{12}-10^{13} \) Gauss magnetic field.
3) The emission features near 400 keV are consistent with the 511 keV \( e^+ - e^- \) annihilation line gravitationally redshifted in the field of a neutron star.
4) The brief pulsations (with periods of 4-10 second) are reminiscent of a slow pulsar.

2.1.2. GRB0528-66: An 'Atypical' GRB

On 5 March, 1979, nine interplanetary and terrestrial satellites detected an exceptionally strong GRB from the direction of the Large Magellanic Cloud (Mazets, et al., 1979b). This burst, designated GRB0528-66 from its celestial coordinates, has significantly added to
Figure 2.3: Four GRB spectra, taken from Teegarden (1982). The spectra show the standard $(1/E)\exp(-E/\kappa T)$ dependency, along with line features present in $\sim 15\%$ of all GRB spectra. The 30-70 keV feature is interpreted as being due to cyclotron radiation in a high magnetic field and the $\sim 400$ keV feature as 511 keV positron-electron annihilation line gravitationally-redshifted in the gravitational field of a neutron star.
the controversy surrounding GRBs because of its many unique characteristics (Cline, 1982):

1) A very fast risetime (200 microsecond) — shorter than any other burst.

2) An extremely high peak flux \( (2 \times 10^{-3} \text{ erg/cm}^2/\text{s}) \) — an order of magnitude higher than the flux detected from any other GRB.

3) A very soft spectrum \( (\text{kT}_{\text{burst}} \sim 30 \text{ keV}) \) — softer than that of almost all other GRBs.

4) Prolonged, repetitive afterpulses (8 second period), lasting much longer than those of any other detected GRB (see Figure 2.4).

5) The possible association of the GRB source with a known celestial object: the error box of GRB0528-66, 20"x80" in size, contains part of the supernova remnant N49 in the Large Magellanic Cloud.

6) The burst source has been seen to recur more than a dozen since the initial burst, although none of the recurrences had a fluence greater than \( 10^{-3} \) of the original burst (Golenetskii, et al., 1984). It is the only GRB source to recur more than a few days after the original burst.

The unique characteristics of GRB0528-66 have lent much support to the theory that a neutron star is the source of a GRB. In particular, the short risetime (<0.2 ms, limited by instrument precision) points to an emission source size of less than 60 km. In addition, the strong 8-second pulsations favor a rotating neutron star as the source of the burst. Finally, the association of the error region with the supernova remnant N49 — if it is true — strongly favor the neutron star remnant of the supernova as the source of the GRB.

It is this last point — the possible association of the GRB with the LMC — that has stirred the most controversy. The proba-
Figure 2.4: The light curve of the peculiar GRB0528-66 (5 March 1979) as seen by ISEE-3 (above) and Venera 12 (below; both taken from Cline (1983)). Note the fast rise to peak, the high peak flux and the strong 8s pulsations.
bility of the GRB error region randomly overlapping the supernova remnant is small. Yet, if the association is correct, the total energy of the burst was prodigious: the total energy would have been \(5 \times 10^{44}\) ergs, assuming isotropic emission, and the average luminosity during the first 120 ms of the burst would have been \(\sim 3 \times 10^{45}\) ergs/second, or \(\sim 10^{12} L_{\text{solar}}\). The possible association of GRB0528-66 with the LMC lends support to the theory that GRBs are generally located at distances of 50-500 kpc from the Earth: this idea is discussed further in section 2.4.

2.1.3. GRB source localization

The precision of localization of \(\gamma\)-ray detectors used on balloon gondolas and \(\gamma\)-ray satellites is generally low. Consequently, a single \(\gamma\)-ray satellite detecting a GRB cannot provide the precise coordinates of the outbursting GRB. The celestial coordinates of an outbursting GRB can be determined much more precisely by measuring the difference in arrival times of \(\gamma\)-rays at three or more satellites detecting the burst. The precision of the localization of the burst increases with the number of detecting satellites (see Figure 2.5 and its caption). Typical error regions determined by this method have dimensions ranging from tens of arc-minutes to degrees. Several (\(\sim 10\)) bursts have been detected by many widely-separated satellites and have been localized to a precision of a few arc-minutes. Such small error regions allow for reasonable searches for the quiescent GRB source. Despite this, no GRB has yet been positively associated with any other celestial object (see section 2.2).
Figure 2.5: Gamma-ray Burst Arrival-Time Localization Method. The location of a gamma-ray burst in space is determined from the differences in the arrival times of the gamma-ray wavefront at several terrestrial and interplanetary satellites. Detection of the GRB by two satellites allows localization of the burst source to an annulus on the celestial sphere. Detection by three satellites allows localization to two diametrically opposite "diamonds" formed by crossings of two inclined annuli. If the GRB is detected by at least four satellites, the localization is unique.
2.1.4. GRB Source Distances

Since no definite correlation exists between a GRB source and another celestial object, the distances to GRB sources are unknown. An estimate of source distance or the population as a whole can be made by determining the spatial distribution of the GRB population: whether the GRBs we see are generally local (d<300 pc), belong to a disk or halo population, or are extragalactic.

Some insight into the question of GRB source distances can be gained by examining the log N(>S) - log S distribution of GRBs, shown in Figure 2.1. Jennings and White (1980) and Jennings (1982) have attempted to reconcile models of GRB source distribution with the log N(>S) - log S curve. An infinite spherical distribution of GRB sources around the Sun will follow an N(>S) ~ S^{-3/2} function, while a disk population would follow N(>S) ~ S^{-1}. The S^{-3/2} function is superimposed on Figure 2.2. The actual logN-logS curve roughly follows N(>S) ~ S^\alpha, with \alpha=0.7 for S < 10^{-5} \text{ erg cm}^{-2}. The monoluminosity models of GRB distribution of Jennings and White (1980) could not account for this value of \alpha. Jennings (1982) has calculated a theoretical logN-logS distribution based on a galactic population of GRB sources with an intrinsic luminosity function. His calculations show that the observed log N(>S)-log S curve can be explained by varying parameters in his model: most notably, biasing the concentration of the luminosity function of GRBs toward low-luminosity bursts permitted a good match to the observed log N(>S)-log S curve. However, in a recent paper, Jennings (1985) questions the validity of
using GRB statistics in determining the distance scale to GRBs.

The nearly-isotropic distribution of GRB sources on the celestial sphere is an indication of the nature of the GRB population (Jennings, 1982). This isotropy favors models that visible GRBs belong to either a local population of GRB sources \(d<\sim 200\) pc or that place GRB sources in an extended halo about the galaxy \(d\sim 50-200\) kpc. A disk population of visible GRBs is excluded by the lack of concentration of GRBs in the plane of the galaxy. Models of GRB source population in a halo of \(\sim 10\) kpc radius about the center of the galaxy are not considered because there is no concentration of GRBs in the galactic hemisphere containing the galactic center. The validity of an extragalactic population of GRBs is reduced by the absence of associations of GRBs with known extragalactic objects, as well as by the unimaginable energetics involved in the creation of a GRB at extragalactic distances.

There is no definitive proof for either a local or extended halo distribution. The association of the GRB 0528-66 with the LMC may be considered in favor of an extended halo distribution, but 1) this association is not definite and 2) GRB 0528-66 is generally considered an anomalous event (because of its many unique features) and its association with the LMC would not be considered contrary to the concept of a local GRB population.
2.2. Observations of Known GRB Sources

The a posteriori observation of GRB source is difficult for two reasons:

1) GRBs do not repeat regularly or often (only two GRBs have been seen to recur: GRB0528-66 (5 March 1979a) has rebursted more than a dozen times since the initial outburst (Golenetskii, et al., 1984) and the source GRB1900+14 recurred twice within four days of the initial event (Mazets, et al., 1981)).

2) The GRB error boxes are generally too large for a reasonable follow-up observation. Only a few (~10) error boxes are small enough for sensible follow-up work (Hurley, 1982).

Most of the follow-up observations of GRB sources are intended to detect the GRB source in quiescence at various energies. Few attempts have been made, to date, to detect radiation from a GRB source during outburst. The error regions of GRB0528-66 and GRB0116-29 (19 November 1978) have been most extensively searched: the former because of its unique features and the latter because of the discovery of an archived optical transient in its error region (Schaefer, 1981). Cline (1983) provides an excellent detailed review of the observations of these and other GRB sources. The results of observations of several error regions are briefly summarized below.

2.2.1. Radio Observations

Hjellming and Ewald (1981) searched the error region of GRB0116-29 for quiescent radio emission at 6 and 18 cm with the VLA. They found three point radio sources (designated B, C and Q) inside the error region: one (Q) is also located inside the error circle of
a weak X-ray source detected by Einstein (Pizzichini, et al., 1985; see section 2.2.3). None, however, is consistent with the error box of the associated archived optical transient (section 2.2.4.2).

2.2.2. Infrared Observations

Infrared observations of the error regions of GRB2312-50 (6 April 1979) and GRB0528-66 and of the radio sources in the error region of GRB0116-29 were carried out by Apparao and Allen (1982) on the 3.9m Anglo-Australian telescope. The error region of GRB2312-50 was seen to be empty to J=17.5. Their observations of the point radio sources detected by Hjellming and Ewald (section 2.2.1) revealed a J=18.4 magnitude object consistent with one of them (B), but no source consistent with the error circle of a weak X-ray source in the field (Pizzichini, et al., 1985; section 2.2.3). The search of the error region of GRB0528-66 was confusion-limited and resulted in no reliable detection.

Schaefer and Ricker (1983) searched the error box associated with the archived optical transient in the error region of GRB0116-29 (section 2.2.4.1). They found no infra-red sources in the error box to a limiting magnitude of K=18.8.

Recently, B. Schaefer searched the IRAS data base for infra-red sources in the 23 smallest known GRB error regions. The sensitivity of the observations used in the search varied considerably: Schaefer reports no infra-red sources detected with an estimated average 4σ sensitivity of ~1 Jansky (B. Schaefer, private communication).
2.2.3. X-ray Observations

Pizzichini, et al. (1985) report observations of five GRB error regions (those of GRB2008-22 (4 November 1978), GRB0116-29, GRB1704+01 (21 November 1978), GRB0528-66 and GRB2312-50) with the Einstein Observatory. Although no overwhelming evidence for the existence of quiescent X-ray counterparts in any of the regions was found, X-ray observations of the error regions of GRB0116-29 and GRB0520-66 have yielded some positive results. The optical and γ-ray error regions of GRB0116-29 are consistent with the error circle of a weak ($10^{-13}$ erg/cm$^2$/sec -- a $3\sigma$ level-of-confidence detection) X-ray source (Pizzichini, et al., 1981), which may be a quiescent X-ray counterpart. The supernova remnant N49, which is included in the error region of GRB0528-66, was also detected by Einstein (Helfand and Long, 1979).

2.2.4. Optical Observations

Optical observations of GRBs have been carried out in a variety of ways. These optical searches have looked for optical radiation from both quiescent and outbursting GRB sources.

2.2.4.1. Search for Quiescent Optical Counterparts

Deep searches of several small GRB error boxes with large telescopes have been carried out by Chevalier, et al. (1981), Fishman, et al. (1981), Laros, et al. (1981), Pedersen, et al. (1983), Schaefer and Ricker (1983), Schaefer, Seitzer and Bradt (1983), among others. The investigation of the error region of GRB0116-29 has uncovered
several faint ($m_V \approx 22$) sources, including an apparently highly-variable ($\Delta m \approx 2$) object (Schaefer, Seitzer and Bradt, 1983; Schaefer and Ricker, 1983; Pedersen, et al., 1983). No definite quiescent optical counterpart to GRB0116-29 has yet been confirmed.

The error region of GRB2312-50 is empty to a limiting visual magnitude of 22.5 (Laros, 1981). The error region of the GRB1412+78 (13 June 1979) contains >5 faint ($m_V = 22$) objects (Vanderspek, Ajhar and Ricker; work in progress). The error region of GRB0528-66 contains the supernova remnant N49 (Cline, 1982).

2.2.4.2. Searches for Optical Light from an Outbursting GRB

Optical transient events have been noted in the literature for more than half a century. In addition to transient events of short timescales from known astrophysical objects, such as cataclysmic variables and flare stars, several unknown optical transient events have been reported. Klemola (1983) reports two possible optical transient events, first recognized by Hertzsprung (1927) and Popovic (1982). Although the Hertzsprung object has been recently recognized as a plate defect by Schaefer (1983), the Popovic object, which was seen as a fifth-magnitude event with a duration of less than 20 minutes, remains without explanation or verified quiescent counterpart. In addition, recent analysis of SEC Vidicon meteor observations made in 1969 revealed that the 4th-magnitude double star β Cam underwent a 0.7 magnitude brightening in a period of 0.25 seconds (Wdiowiak and Clifton, 1985).
In 1981, B. Schaefer of MIT began a survey of historic photographic plates of three small GRB error regions, in the hope of finding an optical transient which may have been associated with a GRB. In the scanning of plates stored at the Harvard plate archives totaling roughly three years of exposure, Schaefer discovered three transient images which are now generally accepted as being optical flashes from historical GRBs (Schaefer, 1981; Schaefer, et al., 1984). The three events were of magnitude 3.0, 6.6 and 4.3, assuming the optical radiation was emitted in one second. Schaefer reported an upper limit on the total duration of the optical bursts of \( \tau < 500 \text{s} \). The ratio of optical fluence in the three archived optical transients, \( S_{\text{opt}} \), to the \( \gamma \)-ray fluence, \( S_{\gamma} \), of the associated GRB is roughly \( 10^{-3} \) for each of the three bursts.

Schaefer's work initiated an entirely new approach to GRB research: it created the hope that the location of GRB sources could be precisely determined by observing optical light emitted during a GRB. His work has sparked a series of new, real-time searches for optical counterparts to outbursting GRBs: some of these experiments are described below.

2.2.4.2.1. Pic du Midi SIT TV Flash Search

In the summer of 1982, Kevin Hurley and collaborators (Hurley, private communication) observed the night sky at Pic du Midi with a wide-field lens on a SIT TV camera in an effort to measure the optical flash background rate and perhaps detect an optical counterpart to a GRB. The Pic du Midi system records images of the night sky at
Figure 2.6: The archived optical transient in the field of GRB0116-29 (19 November 1978), discovered by Schaefer (1981). The upper plate shows the transient event, found on a 45 minute exposure taken in 1928. The lower plate was taken of the same field 45 minutes before the upper plate. The lack of trailing of the burst image puts an upper limit of 500s on the burst duration. The ratio of 1928 optical fluence to the 1978 gamma-ray fluence is 0.001.
video rates, which are viewed after the fact by a human worker. Its
time resolution was 0.04 seconds and its sensitivity $m = 5.5$.
Because of its low angular resolution (one degree), the Pic du Midi system cannot differentiate between head-on meteors and real celestial optical flashes, thus making them reliant on simultaneous detection of a GRB by a $\gamma$-ray satellite to confirm any optical counterpart. In over 100 hours of observations with a three-steradian field-of-view, no optical counterparts to GRBs have been reported.

2.2.4.2.2. Two-Schmidt Sky Survey

In October, 1982, a unique set of observations designed to
detect celestial optical flashes was carried out (Schaefer, Vander-
spek, Bradt and Ricker, 1984b). Simultaneous observations of several
patches of sky were made with identical 0.4m Schmidt telescopes
located at the Cerro Tololo Inter-American Observatory in Chile and
at Kitt Peak National Observatory in Tucson, Arizona. Two telescopes
were used in order to confirm any optical flashes detected, thus
eliminating local sources of optical flashes. In addition, the 6000
km baseline between the sites allowed the use of trigonometric paral-
lax to recognize sources of optical flashes within ~10 AU of the
Earth. A total of 890 square-degree-hours (0.27 sr-hrs) of observa-
tions were made, with a median one-second sensitivity of 13th magni-
tude. No flashes were detected, resulting in a $3\sigma$ upper limit on the
celestial optical flash rate of 54 flashes/hr/sr at 13th magnitude
and 22 flashes/hr/sr at 10$^{th}$ magnitude.
2.2.4.2.3. **GRB0528-66 Monitoring**

Holger Pedersen and co-workers at the European Southern Observatory (ESO) have recently monitored the error box of the peculiar GRB0528-66 (5 March 1979) with a photometer mounted on a 50 cm telescope (Pedersen, et al., 1984). The aperture of the photometer matched the shape of the 3σ level-of-confidence error region of GRB0528-66. The output of the photometer was stored on magnetic tape. The ESO team has published the detection of three significant optical flashes in 910 hours of observation of the error region of GRB0528-66. No coincident gamma-ray events were detected by any satellite operating at the time. Pedersen notes, however, that none of these bursts could have been detected at γ-ray energies by any of the satellites, based on the ratio \( \frac{L_\gamma}{L_{\text{opt}}} \geq 10^3 \).

Pedersen's method, although quite sensitive, suffers from the inability to reject terrestrial sources of optical flashes. The detector is a simple photometer without any anticoincidence detector, so that any object moving quickly through the field of view, such as a satellite or meteor, could create a light curve similar to that of an optical flash. Indeed, the three detections may be consistent with a meteor or satellite crossing the detector field (see Chapter 11).

2.2.4.2.4. **Conclusion**

The discoveries by Schaefer (1981; Schaefer, et al., 1984) of optical transients associated with GRBs have demonstrated that bursts
of optical radiation can be expected from GRB sources. The detection and study of optical radiation from GRBs would lead to better understanding of the GRB phenomenon. In addition, a comparison of the characteristics of the optical and \( \gamma \)-radiation would provide greater insight into the mechanism of the production of \( \gamma \)-rays and optical light in GRBs. Optical detections of outbursting GRBs would lead to a more precise source localization than presently possible with \( \gamma \)-ray satellites, leading to more fruitful \textit{a posteriori} observations of the GRB source.

The methods for searching for optical light from GRBs described in the preceding section are all effective for searching for flashes from limited regions of the sky, yet are ineffective as general monitors of optical flashes from GRBs. To be more specific:

(1) Hurley's Pic du Midi sky monitor has the advantage of a dedicated, wide-field instrument, yet the data collected must be analyzed after the fact, in real time, by a human observer. This method is time-consuming and fraught with human frailties. In addition, the low detector resolution and the absence of a coincidence detector limit the effectiveness of the method in general.

(2) Schaefer's archived plate method leaves thousands of plates at the investigator's disposal, yet no optical flash detected can ever be confirmed as coming from a GRB. In addition, the method is very time-consuming, since each plate must be visually scanned by the investigator.

(3) The Two-Schmidt survey method combines large viewing solid angle with a moderately large observing time, and with its use of coincidence is effective as a survey method. However, it relies on the acquisition of observing time on two telescopes at the same time, and suffers from the long analysis time of Schaefer's method.
Pedersen's monitoring of GRB0528-66 has the advantage of being done with a dedicated telescope, but its small field-of-view limits the applicability of the method to a single object. In addition, its lack of anticoincidence detector significantly reduces its reliability as a detector of optical flashes from a point source. (This limitation has been recognized by the ESO team, and they are planning to incorporate a second, imaging instrument operating in coincidence with their photomultiplier detector).

Ideally, it would be desirable to assemble an instrument which combines the positive features of all methods: a dedicated, automated, wide-field detector of sudden optical flashes with coincidence detectors to confirm any flashes. Such an instrument is the Explosive Transient Camera (ETC), described in the following chapters.
CHAPTER 3

Theories of Radiation from GRBs

Introduction

Since the discovery of GRBs (Klebesadel, et al., 1973), many theories have been proposed to explain the phenomenon of the GRB. It is only in the last ten years that the understanding of the GRB has progressed to the point where the number of GRB detections has exceeded the number of models attempting to explain GRBs. The increased number of detected GRBs has enabled workers in the field to cull out implausible theories of the mechanism and space distribution of GRBs. Still, because of the large number of unknown facts about GRBs, many theories can still explain the observed characteristics of GRBs.

With the discovery in 1981 of transient optical radiation from a GRB error region (Schaefer, 1981), new data about GRB sources have become available. As a result, several new theories of the production of optical radiation from GRB sources have been proposed since 1981. The discovery of transient optical radiation from a GRB source has introduced new constraints on the theories of GRB emission which would predict optical radiation from the same source. Only a few self-consistent models of transient $\gamma$-ray and optical radiation from a GRB source have emerged since 1981.
This chapter is designed to give the reader an overview of the most accepted theories of the production of gamma-rays and of optical light during a gamma-ray burst. Space limitations dictate that the discussion of these theories be in the form of short explanations: the reader should refer to the appropriate reference for more detailed information about a specific model. Ventura (1983), Katz (1984), and Lamb (1984) also provide excellent reviews of the physics and proposed theories of γ-ray emission from GRB sources.

3.1. Models of Gamma-Ray Production in GRBs

Any theory of GRB production and source location must be able to explain the most common observed characteristics of GRBs:

1) Short risetimes (0.05 - 1 second).

2) Spectral shape \( N(E) \sim E^{-1}e^{-E/kT} \), with \( kT \approx 100-500 \) keV).

3) Total energies (based on detected fluences ranging from \( 10^{-3} \) to \( 10^{-8} \) erg/cm\(^2\) and an assumption of the distance to the source).

4) Line features near 30-70 keV and near 400 keV in some burst spectra.

5) Pulsations (4-10 second periods) in a few (2-3) burst lightcurves (Mazets, et al., 1979b; Wood, et al., 1981).

6) Low quiescent flux in energy bands from radio to γ-rays.

7) An estimate of the recurrence rates of some GRBs of the order of 1 yr\(^{-1}\) (Schaefer and Cline, 1985), based on the detection of three archived optical transient events of Schaefer, et al (1984a).

Because of the wide variety of detected GRB characteristics, it is possible that no single GRB theory can explain every characteristic of every observed GRB.
3.1.1. GRB Spectral Shape and Features

The physics explaining the continuum shape and low-energy features of a GRB spectrum are in principle independent of the physics explaining the other GRB characteristics. The energy dependence of the GRB continuum spectra \( (E^{-1}e^{-E/kT}) \) are consistent with the emitting material being an optically-thin plasma. The 30-70 keV line features are consistent with cyclotron emission (or absorption) lines from an optically-thin plasma in a high (~10^{12} Gauss) magnetic field.

The energy dependence of the continuum spectra can be described by a variety of different models. Liang (1984b) points out that single-temperature thermal bremsstrahlung or inverse-Compton models cannot explain GRB continuum spectra because the high-energy cutoff (at a few times \( kT \)) predicted by these models has not been observed. The thermal synchrotron model of Liang (1984a) fits the observed spectra well out to high energies and predicts the 30-70 keV line features. However, Liang (1984a) notes that the exponential continuum shape can be explained by any number of models. (For a good review, see Lamb (1984)).

3.1.2. The Energy Sources of GRBs

As discussed in section 2.1.1, the observed characteristics of GRBs (short risetime, pulsations, gravitationally-redshifted \( e^+e^- \) line) point to a neutron star as the source of the burst. Various energy sources have been proposed to power the burst, including the neutron star's gravitational and rotational energy, the gravitational
energy of impacting matter and the nuclear energy of matter on the surface of the neutron star. These energy sources will be discussed in the sections below.

Several models of the energy source of GRBs require a companion star and/or an accretion disk. The existence of a companion has profound consequences for the detection of a quiescent source: any companion star or accretion disk would most likely be more visible than the neutron star primary in the optical band, and any significant accretion onto local ($d \leq 200$ pc) neutron stars will create an X-ray flux detectable at the Earth (see Rappaport and Joss (1985) and section 3.1.2). These models are also important in the discussion of mechanisms producing optical burst radiation to follow in section 3.3.

Most models of the source of the total energy of GRBs are based on an assumption of the distance to the GRB source. The theories of $\gamma$-ray production from a local ($d < 500$ pc) distribution of GRB sources will propose total burst energies in the range of $10^{35}-10^{39}$ erg (based on the range of detected fluences of $10^{-3}-10^{-7}$ erg/cm$^2$). The theories of $\gamma$-ray production from an extended-halo population of GRBs ($d > 50$ kpc), or to explain GRB0528-66 (5 March 1979) as being in the LMC, derive total burst energies of $10^{39}-10^{43}$ erg.

It should be noted that the Eddington luminosity for a 1.4$M_{\odot}$ neutron star is ~$10^{38}$ erg/s. Any GRB mechanism which predicts significantly super-Eddington luminosities has to contend with a fraction of the total burst energy going into the kinetic energy of matter.
driven from the surface of the neutron star by radiation pressure (Colgate and Petschek, 1981). One proposed means of avoiding this problem is to confine the ejected matter near the surface of the neutron star with a large surface magnetic field, thereby increasing the \(\gamma\)-ray production efficiency of the burst (Woosley and Wallace, 1982).

The following sections contain discussions of some of the more widely-accepted mechanisms for the production of GRBs. Each section is based on the source of the energy of the GRB, as follows:

1) Sudden accretion of matter onto a neutron star, which liberates \(~10^{20}\) ergs of gravitational energy per gram of accreted matter.

2) Thermonuclear detonation of accreted matter on the surface of the neutron star, which liberates \(~10^{38}-10^{39}\) ergs in \(\gamma\)-rays (Woosley and Wallace, 1982).

3) Physical changes in the state of the neutron star, which may liberate large amounts of energy (up to \(~10^{46}\) ergs).

The energies quoted here are total energies liberated by the particular mechanism. The energy of the GRB in \(\gamma\)-rays depends of the efficiency of the burst mechanism.

3.1.2.1. GRB Production by Sudden Accretion onto a Neutron Star

The sudden accretion of matter on to a neutron star involves the collision of a 5-10 km solid body with a neutron star. Various theories have been put forth describing the effects of such a collision. A body approaching close enough (within \(~10^{5}\) km) to a neutron star is broken up tidally and continues to orbit the neutron star as a stream of particles. These particles can lose enough of their angular momentum to the neutron star's magnetic field to strike the
star, releasing $G M_{\text{NS}}/R_{\text{NS}} \approx 10^{20}$ ergs per gram of accreted matter in gravitational energy. Most of this energy would appear as thermal X-rays from the heated neutron star's surface, but nuclear collisions and non-thermal radiation from infalling and re-ejected material could lead to an appreciable $\gamma$-ray flux (Colgate and Petschek, 1981).

Colgate and Petschek (1981) and Van Buren (1981) discuss event rates based on impacts of interstellar asteroids onto neutron stars, while Joss and Rappaport (1983) propose the in situ formation of asteroids in a cold accretion disk. Harwit and Salpeter (1973) and Tremaine and Zytkow (1985) explore the energetics and event rate of collisions of comets from a comet cloud about a neutron star or white dwarf.

3.1.2.1.1. Collision of an Asteroid with a Neutron Star

Colgate and Petschek (1981) analyze the direct collision of a 5 km body with a neutron star. In their model, the body is tidally disrupted within $\sim 10^8$ cm: the resulting matter is deformed tidally and thermally into a long ($\sim 10$ km), thin ($\sim 3$ mm) curtain of matter, which strikes the neutron star surface along a line of magnetic longitude. The total impact time of the matter is of the order of $\sim 1$ ms. The impact sends up a plume of plasma, which then radiates in the magnetic field of the neutron star. The total gravitational energy available for the burst is $\sim 10^{40}$ ergs from a body with $m \approx 5 \times 10^{19}$ g. A strong surface magnetic field is required in this model to confine the bursting material. If no magnetic field is present, the efficiency of the GRB would be low because the impact energy is converted
to the kinetic energy of material ejected by radiation pressure.

This model does not require any neutron-star companion or accretion disk and does not predict any detectable quiescent flux in any energy band. However, the recurrence rate based on the random collision with an asteroid-sized body with a neutron star has been estimated to be low (a few times $10^{-7}$ yr$^{-1}$; Newman and Cox (1980)). Calculations by Van Buren (1981) of the rate of collisions of interstellar asteroids deflected by gravitational interactions with a planetary system about the neutron star (thereby increasing the collision cross-section of the neutron-star system) have yielded slightly higher, yet similarly low collision rates ($\sim 10^{-6}$ yr$^{-1}$).

Joss and Rappaport (1983) have proposed the possibility of the condensation of iron-nickel bodies at a rate of up to $\sim 1$ yr$^{-1}$ from a cold accretion disk about a neutron star. In their scenario, the neutron star is in a close orbit with a low-mass companion from which matter had been accreting for several billion years. When the companion mass drops below a certain level, the rate of accretion to the disk and onto the neutron star drops steadily. The viscosity of the disk may then very well decrease, in which case the disk cools slowly until the temperature is such that iron-nickel grains could condense out of the disk. These grains would settle into a thin plane inside the accretion disk and condense into asteroid-sized bodies by a series of inelastic collisions. The resulting body could then possibly give up its angular momentum to the neutron star's magnetic field, causing it to strike the neutron star surface and create a GRB
as in the model of Colgate and Petschek (1981).

This scenario has many positive aspects. The recurrence rate estimated by Joss and Rappaport agrees well with the estimates of Schaefer and Cline (1985). The absence of accretion onto the neutron star's surface accounts for the absence of a quiescent X-ray flux. In addition, the existence of an accretion disk and companion has important implications for some theories of the generation of optical radiation, as will be seen in section 3.3.

3.1.2.2. Impact of a Comet onto a Neutron Star

Harwit and Salpeter (1973) have proposed that GRB could be produced on a regular basis by impacts onto a neutron star of comets from a comet cloud surrounding the neutron star. In their model, comets straying within \( \sim 10^5 \) km of the neutron star are tidally broken up into a stream of smaller bodies which spread along the comet's orbit. Comets with periastron distances much less than \( 10^5 \) km are compressed and heated and spread into a set of orbits about the neutron star. Disrupted comet matter could lose its angular momentum to the magnetic field of the neutron star and then be guided onto the surface of the neutron star along magnetic field lines. Such a collision of a comet of mass \( 3 \times 10^{17} \) g would release \( \sim 3 \times 10^{37} \) erg of gravitational energy: if \( \sim 3\% \) of the available energy were released in the GRB (a very uncertain estimate), \( \sim 10^{36} \) ergs of energy would be available for the burst.
Harwit and Salpeter did not discuss a key aspect of the creation of GRBs by the impact of comets from a comet cloud about the neutron star: the retention of the comet cloud by the neutron star during the formation of the neutron star. Their model has been reanalyzed by Tremaine and Zytkow (1985). Besides rediscussing the basic aspects of the collision of a comet with a neutron star, Tremaine and Zytkow address the problem of the retention of the comet cloud during the formation of the neutron star. They conclude that it is indeed possible to create a neutron star without losing the cloud of comets present around the parent star, thus enabling the system to be a possible source of GRBs.

In their paper, Tremaine and Zytkow note that a cloud of comets orbit a star at a mean distance of ~20,000 AU is very loosely bound to the star (the escape velocity is of the order of 1 km/s). If, during the creation of a neutron star from the parent star, the neutron star is given a peculiar velocity significantly greater than ~1 km/s, the comet cloud will not remain bound to the neutron star. Measurements of pulsar radial velocities indicate that many neutron stars are created with high peculiar velocities, presumably due to asymmetries in the supernova explosion creating the neutron star. At typical velocities of ~100 km/s, such a neutron star would escape a cloud of comets at a mean distance of 20,000 AU from the progenitor star (which has an escape velocity of the order of 1 km/s) within $10^3$ years. Tremaine and Zytkow discuss four scenarios for the creation of a neutron star with low enough peculiar velocity so that the comet
cloud remains bound. These scenarios are:

1) The symmetric type II supernova explosion of a single massive star. Tremaine and Zytkow quote results of calculations by Hills (1983) that imply that a good fraction (roughly half) of the comets in high eccentricity orbits would remain bound during a sudden mass loss by the central star. The fraction of type II supernovae that satisfy this criteria is unknown.

2) The creation of a neutron star in a cataclysmic variable due to mass accretion by the white dwarf until its mass exceeds the Chandrasekhar limit. The result is a binary system, generally including a neutron star and a low-mass secondary. Those systems which would retain their comet cloud would have secondaries with masses less than \(0.03M_{\odot}\). (This binary system is very similar to one proposed by Rappaport and Joss (1985) to explain optical flashes from GRBs (see section 3.3)).

3) The creation of a neutron star from a white dwarf accreting matter from a giant companion in a wide binary orbit. Tremaine and Zytkow propose that the resulting system would be a neutron star in orbit with the white dwarf core of the giant companion. The peculiar velocity of the resulting system would be small due to its large period.

4) The merging of a close pair of white dwarfs, possibly creating a Type I supernova and/or neutron star.

The analysis of the physics of the interaction of the comet with a neutron star is similar to the analysis of Colgate and Petschek (1981). The comet is tidally disrupted at a distance from the neutron star dictated by the tensile strength of the cometary material. Gravitational forces compress the disrupted comet into a long, thin stream of conducting material, which can lose angular momentum to the neutron star's magnetic field through the generation of Alfvén waves. If the energy loss by the cometary material is not large enough to allow the material to strike the neutron star, the probability is
high that the material will impact the neutron star in a following passage. Indeed, since the maximum impact parameter for accretion on the first encounter is relatively small, most bursts will occur when the comet is disrupted on the first pass by the neutron star and accretes on the second pass. The total time in which the cometary material strikes the neutron star depends on the extent of its spread during previous encounters with the neutron star. GRBs of duration 0.1 to 30 seconds are in principle allowed by this model, with burst times less than one second restricted to comets with an impact parameter of less than a few hundred kilometers.

The estimated rate of impacts onto the neutron star by comets "straying" near the neutron star is $\approx 10^{-4}$ yr$^{-1}$ for either a solitary neutron star or one in a binary system. Tremaine and Zytkow note that Hills (1981) has pointed out that a close encounter of the comet cloud with a field can create a relatively brief period (duration $\approx 30,000$ years) of high comet influx into the neutron star system. During these periods, the observed impact rate is enhanced: burst recurrence rates of $\approx 1$ yr$^{-1}$ are easily explained by this model. As a result, the mean impact rate increases to $\approx 10^{-3}$ yr$^{-1}$ Tremaine and Zytkow emphasize that these numbers are conservative and fairly uncertain, and that the rates could be much higher.

The intriguing possibility of creating bursts of optical radiation by impacting a comet onto the white dwarf companion of the neutron star is discussed further in section 3.3.1.
3.1.2.2.1. Unstable Accretion of Interstellar Matter

Lipunov et al. (1982) propose that interstellar material can accumulate as an envelope of matter in the magnetosphere of a highly-magnetized neutron star. When enough matter has accumulated, the envelope becomes unstable and accretes quickly onto the poles of the neutron star, releasing \( \sim 10^{37} \) ergs of gravitational energy. The accretion rate, and therefore the recurrence time, is very sensitive to the neutron star velocity. Slower neutron stars will accumulate mass more quickly than faster ones and will therefore recur more often. Lipunov estimates a recurrence time of \((0.1 \text{ years}) \times (0.1V)^3\), where \( V \) is the neutron star's velocity in km/sec. According to this model, then, a neutron star with \( V=100 \) km/sec will therefore recur every \( \sim 100 \) years.

3.1.2.3. GRB Emission from the Detonation of Accreted Matter

Woosley and Wallace (1982) and Fryxell and Woosley (1982) propose models of GRB production in which matter accreted onto a neutron star from a companion star or accretion disk ignites explosively to create a gamma-ray burst. Matter (mostly hydrogen) accreting from an accretion disk onto a neutron star accumulates on the surface of the neutron star and fuses non-explosively into helium. The accreted matter can collect in a kilometer-sized area, either due to the funneling of the matter onto the poles of the neutron star by the strong magnetic field or due to the presence of "wrinkles" in the magnetic field at the neutron star surface (Woosley and Wallace, 1982). The accreted hydrogen and helium form a "blister" on the surface of the
neutron star. The blister measures tens of meters deep and has a
surface area of the order of $\sim 1-10 \text{ km}^2$. The matter in the blister
will tend to spread over part of the neutron star surface: the extend of the spread determines the total mass of the blister at the
time of detonation, and so the energy of the burst. When the pres-
sure and temperature at the base of the blister reach the point where
helium at the base of the blister undergoes runaway thermonuclear
fusion, a blast wave of fusion propagates through the blister,
releasing $10^{38}-10^{39}$ ergs in $\gamma$-rays per $\text{km}^2$ of accreted matter. The
hot ($T = 10^9-10^{10} \text{ K}$) plasma thrown up by the explosion interacts
with the magnetic field of the neutron star, creating the GRB.

It should be noted that matter accreting onto the surface of a
neutron star will emit X-rays. Depending on the distance to the
source and the mass transfer rate, these X-rays may be detectable at
the Earth. According to Rappaport and Joss (1985, equation 4), the
quiescent X-ray flux of GRB0116-29 ($10^{-13}$ erg/cm$^2$/s; cf. section
2.2.3) is such that the recurrence time between bursts is $4\times10^5$ years
if the source is at 100 pc, 3500 years at 1 kpc and four months at
100 kpc, assuming isotropic emission.

Application of this model to GRB0116-29 (19 November 1978)
presents some difficulties. The apparent detection of two bursts
from this source within 50 years imply an accretion rate which would
create the detected X-ray flux if the source is at a distance of $\sim 8$
kpc.* On the other hand, a total burst energy of $\sim 10^{38}$ ergs places
GRB0116-29 (fluence = $3.2\times10^{-4}$ erg/cm$^2$) at a distance of 50 pc from
the Earth. Since all models of optical radiation from GRBs favor a local \((d < 100 \text{ pc})\) population of GRB sources, it is probable that either this model does not apply to GRB0116-29 or the X-ray flux is from a serendipitous source in the error region of GRB0116-29.

3.1.3. GRB Emission from Phase Changes inside a Neutron Star

A phase change inside a rotating neutron star is usually seen as a starquake accompanied by a release of energy from the neutron star. Pulsar "glitches", where the period of a pulsar changes suddenly and discontinuously, are thought to be associated with some change in the physical state of the neutron star. The amount of energy release in a neutron star glitch is roughly \(E_{\text{total}}(\Delta P/P)\), where \(\Delta P\) is the change in the neutron star rotational period \(P\). Pulsar glitches have been observed at intervals of ~10 years, so each glitch model would allow recurrence times of the order of 10 years.

In pulsar-glitch models of the production of GRBs, the starquake in the neutron star causes a sudden change in the magnetosphere of the neutron star, which creates a strong electric field near the neutron star surface. This electric field pulls charged particles from the surface of the neutron star, which than radiate in the neutron star magnetosphere. The fraction of the energy released by the

*Assuming the X-rays are emitted isotropically from the neutron star: if the emission occurs in a small area on the surface of the neutron star, the distance to the source will decrease as the ratio of the emission area to the area of the neutron star. However, any significant concentration of emission onto one spot on the surface of the neutron star would most likely be noticed as the neutron star rotates; yet, no reports of pulsations in the quiescent X-ray flux have been published.
glitch that appears in fast particles and the total energy radiated in γ-rays is model-dependent.

Several theories of GRB production from starquakes have been proposed, with different results. Mitrofanov (1984) has calculated that a starquake in an old pulsar could release of the the order of $10^{46}$ ergs of the gravitational energy of the neutron star (~$10^{53}$ ergs), based on the assumption that the change in the neutron star period is accompanied by a change in the neutron star radius, and that $\Delta P/P = \Delta R_{\text{ns}}/R_{\text{ns}}$. Pacini and Ruderman (1974), on the other hand, calculate total energies of $~10^{35}$ ergs, assuming the GRB derives its energy from the rotational energy of the neutron star ($E_{\text{rotational}} = 10^{45}$ erg for a neutron star with a period of 6s).

The starquake model of GRBs are, in principle, capable of creating the short risetimes (10-500 ms) seen in GRBs. If the source of the GRB energy is the rotational energy of the neutron star, the characteristic timescale of radiation of energy is the time needed for an Alfven wave to cross the neutron star magnetosphere, which is about 1 ms (Lamb, 1984). If the energy source is the gravitational energy of the neutron star, the characteristic timescale of energy conversion is the orbiting time of a particle just above the neutron star surface, which is $~0.1$ ms (P. Joss, private communication).

Because of the six orders of magnitude difference in the burst energy predicted by the models of Pacini and Ruderman (1974) and Mitrofanov (1984), the mean distances to GRB sources predicted by the two theories differs by three orders of magnitude. As a result, the
two theories prefer different source distributions: Pacini and Ruderman have proposed a local GRB source population, while Shklovskii and Mitrofanov (1985) have proposed an extended-halo GRB source population based on the model of Mitrofanov (see section 3.2).

3.2. The Distribution of GRB Sources

Since GRBs are thought to originate near the surface of neutron stars, some insight into the space density of GRB sources can be gained by an analysis of the space density of neutron stars. Isotropy arguments (section 2.1.4) point to a spherical distribution of detected GRB sources about the Earth. The GRB sources are either close to the Earth \(d < 100-300 \text{ pc}\) or in an extended halo about the Galaxy \(d = 50 - 200 \text{ kpc}\).

The present best estimate of the rate of creation of neutron stars is \(\sim 0.03 \text{ yr}^{-1}\) (Shklovskii and Mitrofanov, 1985). If this rate has persisted throughout the life of the Galaxy \(\sim 10^{10} \text{ years}\), there have been \(N_{ns} = 3 \times 10^8\) neutron stars created in our Galaxy. If only a fraction, \(f_{GRB}\), of these neutron stars are responsible for GRBs observable by present instruments (perhaps because only a fraction of all neutron stars can create GRBS, or perhaps because only a fraction of all neutron stars are close enough to be detected in outburst), then there are \(f_{GRB} N_{ns}\) observable GRB sources in the Galaxy.

If one can assign some sort of mean recurrence time, \(\tau_{rec}\), to GRBs as a population, then the observed GRB detection rate (of the order of \(30 \text{ yr}^{-1}\)) can be compared with the expected rate.
\( f_{GRB}^{N_{ns}}/\tau_{rec} \). From this comparison, it is clear that if every neutron star is a potential site of a detectable GRB \( (f_{GRB} = 1) \), the mean recurrence time is \(~10^7\) years. On the other hand, if the recurrence time of about one year calculated by Schaefer and Cline (1985) is correct, the value of \( f_{GRB} \) is \(~10^{-7}\).

Shklovskii and Mitrofanov (1985) point out that typical pulsar peculiar velocities are high (\(~100-200\) km/s) and that, therefore, the galactic population of old pulsars is spherically distributed within 100-200 kpc of the center the Galaxy. It is not clear whether this statement can be made of neutron stars in general. First, the observable pulsars make up only a small fraction (<10%) of the expected number of young (age < 10^6 years) neutron stars in the Galaxy. Second, Tremaine and Zytkow (1985) have suggested four scenaria for the production of neutron stars with low peculiar velocities (cf. section 3.1.2.2), implying that not all neutron stars are born with velocities typical of pulsars. Neutron stars as a group have some intrinsic velocity function: the validity of the use of pulsars as a tracer of this function is questionable.

Shklovskii and Mitrofanov have used the arguments listed above and suggested that GRBs belong to a class of "switched-off" radio pulsars. These old pulsars all have high peculiar velocities (>100 km/s) and populate an extended halo about the Galaxy at a mean distance of \(~100\) kpc. The old pulsars create GRBs through the mechanism proposed by Mitrofanov (1984), described in section 3.2.3.3. Shklovskii and Mitrofanov suggest that high peculiar velocity, early
pulsar activity and late GRB activity are intimately related: only fast, old pulsars are capable of bursting. Because the fast pulsars are far away from the plane of the Galaxy before they start their bursting phase, no nearby GRBs can be expected. In this case, the value of $f_{\text{GRB}}$ is just the fraction of all neutron stars capable of pulsar activity and, therefore, of GRB activity. If this value is roughly 0.05 (from the ratio of the number of known pulsars to the number of neutron stars expected to have been created in one pulsar lifetime ($\sim 10^6$ years)), then there are roughly $10^7$ GRB sources; the average recurrence time is, then, $\sim 5 \times 10^5$ years.

On the other hand, perhaps most (>50%) neutron stars are potential GRB sources, and the observed GRBs are distributed close to the Earth. The fraction, $f$, of all neutron stars capable of GRBs would be dictated by the mechanism of the burst: if, for example, all GRB source are in binary systems, then the value of $f$ would be dominated by the fraction of binary systems which would allow and survive the transformation of one member into a neutron star.

It is interesting to examine the specific case where $\tau_{\text{rec}} \approx 1$ yr and $f_{\text{GRB}} \approx 10^{-7}$. If all neutron stars are evenly distributed within 100 kpc of the center of the Galaxy, and all neutron stars are GRB sources, then the nearest $10^{-7} N_{\text{ns}}$ GRB sources would be located within 300 pc of the Earth. If $\tau_{\text{rec}} = 10$ yr, the distance scale increases to $\sim 600$ pc. Pacini and Ruderman (1974) have suggested that GRB sources are distributed in a disk 2 kpc thick in the plane of the Galaxy. If the radius of this disk is 15 kpc, the nearest 300 GRB
sources (corresponding to $\tau_{\text{rec}} = 10$ years) are located within 70 pc of the Sun.

3.3. Optical Radiation from Gamma-Ray Burst Sources

Since the discovery of transient optical radiation from outbursting GRB sources, several theories have been published attempting to explain its existence. The number of theories has remained small to date, much lower than the number of theories originally proposed to explain $\gamma$-ray emission from GRBs. The emission of optical light is much more strongly constrained by observations than the emission of $\gamma$-rays. Specifically, any theory of optical emission from GRBs must satisfy the following requirements:

1) The ratio of gamma-ray to optical fluence, $F_\gamma/F_{\text{opt}}$, must be of the order of $10^3$. This value was determined from the three archived optical flashes found by Schaefer (1981) and Schaefer, et al. (1984a), assuming a short (<5 second) optical burst time. This ratio could drop by a factor of several if the duration of the optical flashes significantly exceeds a few seconds (Rappaport and Joss, 1985).

2) The quiescent blue magnitude of the object $m_B > 23$, based on various deep searches of small GRB error regions.

3) The duration of the optical flash, $\tau_{\text{opt}}$, must be less than ~500 seconds, based on the limit of image trailing in the 1928 archived optical flash found in the error region of GRB0116-29 (Schaefer, 1981).

The theories describing optical emission from GRBs describe the emission as being either 1) an integral part of the burst emission spectrum, or 2) a by-product of the burst (such as in the absorption of high-energy photons from the burst and their re-emission at optical wavelengths). Each of these types of emission model brings new
restrictions on the mechanism producing the optical radiation.

It has been noted by several workers in the field that if the mechanism of emission of optical radiation is thermal, the radiation cannot be emitted at or near the surface of a neutron star because of the neutron star's low surface area. (Katz (1985) calculated that the brightness temperature of the archived 1928 optical flash was \( \sim 10^{16} \, \text{K} \) (based on an estimate of \( m_v=3 \) for \( \tau_{\text{opt}}=1\text{s} \); if \( \tau_{\text{opt}} = 500\text{s} \), \( T_b \) is still greater than \( 10^{15} \, \text{K} \)). Because typical temperatures seen in GRBs are of the order of \( 10^9 \, \text{K} \), Katz concluded that any mechanism producing optical radiation from the surface of a neutron star must not be thermal in nature.)

Schaefer and Ricker (1983) calculated that if the optical emission process is thermal, the emission must come from a region of radius \( > 10^8 \, \text{cm} \) (for a GRB source distance of \( \sim 50 \, \text{pc} \)) in order to be able to explain the optical flux at the Earth. Katz (1985) also noted that if the optical light is the result of thermal reprocessing of \( \gamma \)-rays from the burst in a neighboring object (a companion star and/or accretion disk), the quiescent temperature of the reprocessing surface must be \( \sim 1600 \, \text{K} \) to be able to explain the 20 magnitudes difference between the quiescent and outburst visual magnitudes.

Several theories for the production of optical light from GRBs have been proposed to date:

(1) Tremaine and Zytkow (1985) noted that thermal optical and UV radiation would be produced by the impact of a comet onto the surface of a white dwarf. This interaction would lead to optical/UV bursts with no GRB counterpart.
(2) Woosley (1984) proposed that optical light could be produced by cyclotron radiation at a distance of ~10^8 cm from a highly-magnetized, outbursting neutron star. Woosley's assumptions on the structure and content of the neutron star magnetosphere are rather uncertain, yet are critical to the details of his model. For this reason, Woosley states that his results are uncertain but that the observational criteria could "in principle" be satisfied.

(3) London and Cominsky (1983) first investigated the reprocessing of γ-rays from a GRB on a main-sequence companion star. Their model, which did not work for a main-sequence companion, was improved by work of Rappaport and Joss (1985), whose model is able to explain the observed characteristics.

All of these theories point to a local (~100 pc) distribution of GRB sources. Some details of these theories are listed below.

3.3.1. The Impact of a Comet onto a White Dwarf

An interesting possibility presented by the model of Tremaine and Zytkow (1985) is for the creation of GRBs from neutron star impacts of comets from a comet cloud about the neutron star (cf. section 3.1.2). In the case of the formation of a neutron star by accretion of matter from a giant companion, impacts of comets onto the white dwarf remnant of the giant companion could give rise to bursts of radiation in the UV and optical bands from the heated surface of the white dwarf. The gravitational energy released in the impact is inversely proportional to the radius of the star impacted, so the ratio of optical to γ-ray burst energy would be R_{NS}/R_{WD}, or ~10^{-3}, which is consistent with observations. Typical quiescent absolute magnitudes of white dwarfs of M=11-16 yield source distances of ~250-2500 pc (given that the minimum quiescent visual magnitude of a GRB source is ~23). The intriguing aspect of this theory is that
optical and γ-ray bursts would not be simultaneous, since comet impacts onto the white dwarf are independent of impacts onto the neutron star. The impact rate onto the two stars would be comparable, so the γ-ray and optical burst rates from the system would be comparable.

This theory can also easily be extended to solitary white dwarfs which have retained their Oort cloud. The same analysis of event rates applies to a solitary white dwarf as to a solitary neutron star: the average impact rate is ~10^{-3} yr^{-1}, with periods of comet storms when an average impact rate of ~1 yr^{-1} can easily be achieved.

3.3.2. Self-absorbed Cyclotron Emission

Woosley (1984) proposes that optical radiation emitted from a GRB is cyclotron radiation intrinsic to the GRB itself. In his model, gamma-rays from the burst strike and accelerate electrons in the neutron star magnetosphere (whether from a burst wind or from accretion). These electrons then gyrate in the magnetic field of the neutron star and emit cyclotron radiation. The γ-ray emitting region near the neutron star is optically thick to the cyclotron fundamental frequency and its first ~100 harmonics, leading to self-absorption of the cyclotron radiation. Due to Doppler smearing of the lines and the magnetic field gradient (assuming a roughly dipole magnetic field about the neutron star), the emitted radiation is the Rayleigh-Jeans tail of a blackbody spectrum, spanning the energy range from the cyclotron fundamental to the ~100th harmonic.
In a dipolar magnetic field with surface strength of \( \sim 10^{12} \) Gauss about a neutron star, the cyclotron energy at a distance of \( 10^8 \) cm is 0.046 eV (27 \( \mu \)m), so the emitted spectrum, which spans a factor of \( \sim 100 \) in energy, covers the range of \( \sim 2700 \) A to 27 \( \mu \)m, which includes the optical band. With this model and taking into account the large uncertainties in the magnetic field strength and distribution and the electron density distribution about the neutron star, Woosley states that the observed ratio of \( F_\gamma / F_{opt} \) of \( \sim 10^3 \) could be achieved. The duration of the optical flash would be the same as that of the GRB (\( \tau_{opt} < 100 \)s), and the intrinsically faint neutron star satisfies the criterion of a faint quiescent source.

Because of its general applicability, the Woosley model of optical radiation from GRBs should be valid in any case: i.e., optical radiation should be emitted from virtually every GRB at some level. The validity of the application of this model to the optical transient events detected to date depends on whether the observed ratio of \( F_\gamma / F_{opt} \) can be achieved.

3.3.3. Reprocessed Gamma-Radiation

London and Cominsky (1983) have proposed that optical radiation could be produced by the reprocessing of GRB \( \gamma \)-rays in a close companion star. They have roughly calculated the brightness of the absorbing surface of a close (separation \( \sim 10^{11} \) cm) main-sequence or white-dwarf companion. They concluded that although this scenario would indeed produce optical light, it was inadequate to explain the observed quiescent criteria. Specifically, they found that their
The proposed binary companion object was too bright and should be detectable if it were a main-sequence star or white dwarf. (As a reference, a star of $m_V = 23$ at a distance of 100 pc has an absolute magnitude of 18. Typical white dwarfs have absolute magnitudes of 11-16 and main sequence stars have absolute magnitudes of up to $M \approx 9$).

Rappaport and Joss (1985) improved on the analysis of London and Cominsky by considering a low-mass brown dwarf as a companion. They found that this system could produce optical radiation which satisfied the observed criteria over a range of companion masses, burst energies and source distances. Their model favors a companion mass of $M_c < 0.05 M_{\text{solar}}$, quiescent temperature $T_{\text{quiescent}} < 1800^0 \text{ K}$ and burst energy $E_\gamma < 10^{38} \text{ erg}$; the favored distances range from $< 50$ pc to $< 250$ pc for Schaefer's three archived optical transients (Schaefer (1981) and Schaefer, et al. (1983)).

3.4. Discussion

The previous sections have outlined the various theories for the production of $\gamma$-rays and optical bursts. Many possible stellar systems were proposed to explain GRBs and optical transients. However, only a few stellar systems were able to act as the source of both GRBs and optical transients. These scenarios are the most important for instruments such as the Explosive Transient Camera, since one can hope to learn about GRB sources from optical transients directly associated with the GRB itself.
The model of optical radiation from GRBs proposed by Woosley predicts optical burst radiation from virtually all GRBs, regardless of the GRB production mechanism. The optical burst would have a duration comparable to that of the GRB, and the optical light curve would be very similar to the $\gamma$-ray light curve, after accounting for the smearing of the optical light curve features over the larger optical emission region. The Woosley model of optical radiation from GRBs would therefore predict optical transients of short duration (a few seconds) with light curves that track the $\gamma$-ray light curve well.

On the other hand, the models of Rappaport and Joss (1985) and Tremaine and Zytkow (1985) both predict optical burst durations of tens to hundreds of seconds. All of these models require $\gamma$-ray/optical burst sources to be in binary systems. The only scenario of the model of Tremaine and Zytkow that allows GRBs and optical transients from the same binary system places the neutron star in a wide binary orbit with a white dwarf. The optical and $\gamma$-ray bursts are created by the impact of a comet with the white dwarf and the neutron star, respectively. In principle, of course, the $\gamma$-ray burst could be created by many of the $\gamma$-ray production mechanisms described above. The model of Rappaport and Joss puts the neutron star in a close binary orbit with a low-mass dwarf star. In their model, GRBs are created by virtually any mechanism described in section 3.3. The optical radiation would be $\gamma$-radiation from the neutron star, reprocessed in the surface layer of the companion star.
Both optical reprocessing models predict similar optical burst durations, consistent with the upper limit of Schaefer (1981). The model of Rappaport and Joss predicts optical burst duration of <500s. In order to satisfy the constraint of $F_\gamma/F_B = 10^3$, the model of Tremaine and Zytikow predicts that the temperature of the white dwarf surface during outburst is $\sim 9000^\circ$ K, so that most of the thermal energy of the burst emerges in the blue band. In that case, the burst duration would be $\sim 100s$ (based on a total burst energy of $10^{33}$ ergs and a white dwarf radius of $10^9$ cm).

Both of these models explicitly predict that the $\gamma$-ray and optical burst rates from such systems should be similar. The main difference between the models is the timing of the $\gamma$-ray and optical bursts. Woosley (1983) and Rappaport and Joss (1985) predict simultaneous optical and $\gamma$-ray bursts, while Tremaine and Zytikow implicitly predict independent GRBs and optical transients. The prediction of the model of Tremaine and Zytikow that isolated white dwarfs will not be able to differentiate between the models, since the calculated optical transient rate from solitary white dwarfs is very low (see Chapter 10).
CHAPTER 4

The Concept of the Explosive Transient Camera

Introduction

The discovery in 1981 of an apparent optical transient associated with a gamma-ray burst error region (Schaefer, 1981) sparked the interest in trying to detect optical light from a GRB source in outburst. Specifically, it became clear that an instrument to monitor the full sky for optical transient events was needed to complement the γ-ray detectors currently in operation in deep space. Data taken from γ-ray and optical transient detectors operating in parallel would provide useful information about the nature and mechanism of GRBs.

In early 1982, George Ricker and co-workers at MIT began the conceptual development of an automated, wide-field instrument to detect celestial optical flashes from space. This instrument, dubbed the Explosive Transient Camera (ETC), would be sensitive to increases of brightness of celestial objects on timescales of 1-4 seconds. The design, construction and testing of a sub-unit of the plenary ETC began in the Spring of 1983.

The ETC would be a very effective ground-based sky monitor, combining a large viewing solid angle (~1.5 steradians) with the large observing time available to a dedicated, automated instrument. As a detector of celestial brightenings on timescales of 1-4 seconds, the
ETC would investigate an entirely new range of parameter space in astronomy, since most astronomical measurements are made on timescales of minutes to hours (standard imaging or spectroscopic observations) or milliseconds (photometric studies with photomultiplier tubes). It is quite possible, therefore, that the ETC may detect optical transients with no astrophysical precedent.

The following chapters describe the ETC test instrument and the operations of the ETC in detail. This chapter discusses the ETC as it will stand in its final configuration; a description of the operation of the ETC will follow in Chapter 5. Chapters 6 through 9 describe the test instrument and the details of the software-controlled operation, data flow and instrument control in the ETC.

4.1. The Plenary ETC

The plenary ETC will consist of 32 CCD cameras monitoring the night sky. Each CCD camera consists of a wide-field lens illuminating a cooled CCD, resulting in a $15^\circ \times 20^\circ$ field-of-view. The 32 CCD cameras will monitor 16 $15^\circ \times 20^\circ$ fields: two CCD cameras will monitor a given patch of sky. In this way, the ETC will operate in coincidence: no optical flash detected in one camera will be considered real unless confirmed by the detection of a flash at the the same celestial coordinates in the other camera monitoring the field. Operating the CCD cameras in coincidence virtually eliminates the possibility of the detection of false optical flashes due to local effects, either in the CCD or in the immediate vicinity of the
The two sets of 16 CCD cameras will be located on two sites separated by ~1.5 km on Kitt Peak, near Tucson, Arizona. (Site 1 is located on the summit of Kitt Peak; Site 2 will be located on the southwest ridge of Kitt Peak, near the picnic area and the NRAO 12m telescope: see Figure 4.1). Sources of optical flashes located in the Earth's atmosphere can be recognized by virtue of the parallax afforded by the distance between the two sites. The precision of localization of an optical flash by an ETC camera is 1.5 arc-minutes, sufficient to recognize (and reject as non-celestial) sources of optical flashes at altitudes of up to 3000 km.

4.2. ETC Observations

The sixteen cameras at each site will be mounted on two sidereal drives, which will track the sky in order to avoid the smearing of stellar images (see Figure 4.2 for an illustration of the layout of eight ETC cameras on one tracking drive). Each set of eight cameras will begin observations at a predetermined hour angle east of the meridian and monitor the sky for about two hours. The drives will then slew back to their original hour angles and monitor a new patch of sky for the next two hours.

The plenary ETC will be entirely automatic. Operations will be initiated by a human user: the ETC computers will then be in complete control of the the ETC instrument and operations. The ETC will operate between evening and morning astronomical twilight, and then
Figure 4.1: The proposed layout of the full-up ETC at Kitt Peak. Site 1, presently the location of the ETC test unit, is the former Airglow Laboratory and Twelve-Inch Schmidt Dome on the summit of Kitt Peak. Site 2 is to be located on the southwest ridge at Kitt Peak, several hundred meters east of the sites of the NRAO and McGraw-Hill telescopes. Intersite data flow will likely be over fiber-optic cable. The Rapidly Moving Telescope (RMT) will be located in the refurbished Airglow Laboratory adjacent to the ETC dome at Site 1. Through the use of two sites, terrestrial sources of optical flashes (such as satellites and meteors) can be eliminated by the use of parallax afforded by the separation between the sites.
Figure 4.2: A schematic layout of the future ETC tracking mount, showing four of eight CCD cameras.
only under photometric or nearly-photometric conditions. The continuous monitoring of weather sensors near each site will allow the ETC control computer to know whether precipitation is present or imminent and thus whether the protective dome should be closed or not opened. The ETC will be able to tell whether the sky is cloudy from an analysis of stellar brightnesses in the CCD camera images; thus, the ETC will be able to judge the quality of the night sky and from this, to be able to determine whether observations should commence or continue. Synthesized voice communication will allow for telephone calls for human assistance or intervention if required.

Observations with the ETC consist of a series of contiguous, short (1-4 second) exposures of the night sky. The observations are controlled and directed by a small, powerful microprocessor known as the Overseer computer; the real-time analysis of ETC image data (at a rate of $10^5$ bytes/camera/second) is done by a Trigger processor associated with each camera. The plenary ETC will consist of one Overseer computer, 32 CCD cameras and 32 Trigger processors analyzing CCD image data in parallel. During observations with the ETC, all CCD images are read out simultaneously at regular intervals of 1-4 seconds. Since the duty cycle of the ETC is 100% (i.e., a new integration in each CCD commences immediately after the readout of an image), the ETC computer system must complete its analysis and storage of image data in less than an exposure time. When the exposure time of an image has expired, the CCDs are read out in parallel: the output of each CCD is amplified and digitized and sent to a dedi-
cated Trigger processor, which analyzes the image for sudden brightenings by an arithmetic comparison of the image to its immediate predecessor. The location on the CCD of any optical flashes detected by a Trigger processor are sent to the Overseer computer, which calculates the celestial coordinates (right ascension and declination) of the event from its location on the CCD. After all Trigger processors have finished image analysis, the Overseer computer compares the celestial coordinates of the events reported by the Trigger processors during the last analysis period. If the Trigger processors associated with a pair of cameras pointing at the same patch of sky report events from the same celestial coordinates (to a programmed precision), the reports are considered to have come from a true celestial optical flash.

In the time between the end of the analysis of the image data by the Trigger processors and the readout of the images being exposed concurrently, the Overseer computer will record and analyze data from any celestial flashes detected in this or previous images. The data stored from a flash are taken from both cameras detecting the event, and include 30'x30' image subarrays centered on the flash and on several photometric standards near the flash. These data are moved from the Overseer computer's volatile memory and stored on magnetic tape at regular intervals. If, in subsequent exposures, the Overseer computer determines that the brightness of a flash has returned to its quiescent level or has subsided below the detection threshold of the ETC, it will no longer store data from that flash event.
The celestial coordinates of any confirmed celestial flash will be sent upon detection to the Rapidly Moving Telescope (RMT; Teegarden, et al., 1984) (presently under construction at the Goddard Space Flight Center), which will be located in a building adjacent to the building housing ETC Site 1. The RMT (see Figure 4.3) consists of a 7" telescope pointing down at a two-axis gimballed mirror, capable of slewing to any spot on the sky within one second with an accuracy of one arc-second. The RMT, with a one-second image sensitivity of $m_v^{-14}$ (10σ significance), will take contiguous <1s exposures of the event and store data from the event until the brightness of the event drops below the detection threshold of the RMT.

4.3. The Sensitivity of the Plenary ETC

The sensitivity of the ETC can be described in two ways: 1) the sensitivity of a single camera in imaging the sky and 2) the sensitivity of a single camera to the detection of an event. The sensitivity of an ETC camera as an imaging device is given by the equation

$$q_i = \frac{S}{(S + \sigma_R^2 + B\tau)^{1/2}}$$

(4.1)

where $q_i$ is the significance of the detection in a single camera, $S$ is the signal at the CCD (in electrons), $\sigma_R$ is the CCD readout noise (in electrons/pixel), $B$ is the sky rate per pixel at the CCD (in electrons/second/pixel), and $\tau$ is the exposure time (in seconds).

An image of signal $S$ electrons in one CCD pixel will be detected at a signal-to-noise ratio $q_i$ in a CCD camera of readout noise $\sigma_R$ and sky brightness $B\tau$. 
Figure 4.3: Cross-sectional view of Rapidly Moving Telescope (RMT) presently under construction at the Goddard Space Flight Center. The two-axis gimbaled mirror can slew to any part of the sky within one second with arc-second accuracy. Light is reflected from the mirror into the aperture of a standard 7" Questar telescope and then focused onto a thermoelectrically-cooled CCD. The field of view of the RMT is 5x8 arc-minutes. The sensitivity of the RMT is $m = 14$ in a one-second exposure (ten sigma level-of-confidence).
The sensitivity of the ETC to the detection of a brightening in a CCD image frame differs from the \( q_d \) because a difference measurement from two image frames is necessary to detect a brightening. Because both images have an associated sky noise and readout noise, the total noise associated with the difference between the images is larger than the total noise associated with a single image. In the calculation of the total noise of the difference of two images, the sky and readout noises must be taken into account twice. The formula for the significance of a difference detection is, then,

\[
q_d = \frac{\Delta S}{(\Delta S + 2\sigma_R^2 + 2Br)^{1/2}}
\]  

(4.2)

where \( q_d \) is the significance of a difference detection in a single camera, and \( \Delta S \) is the signal level (in electrons) above the sky+bias level in the CCD. (This equation is a specific instance of equation 11.1 for the situation where the star is not visible before the event).

In order to relate the detected signal \( S \) to stellar magnitudes, one must have a complete understanding of the effects of absorption due to the various media between the star and the CCD. A star which is seen as \( N \) photons/s/cm\(^2\)/m\(^2\) at the top of the atmosphere will be measured as \( e_a e_f e_l e_w e_c N \Delta \lambda \tau \) electrons at the CCD, where \( e_a, e_f, e_l, \) and \( e_w \) are the transmissivities of the atmosphere, filter, lens and window, respectively; \( e_c \) is the efficiency of the CCD (in e\(^-\)/photon); \( e_v \) is the averaged effect of the reduction of overall lens transmissivity due to vignetting; \( A \) is the area of the lens; and \( \Delta \lambda \) is the
bandpass of the filter (in Å). When the fact that the point-spread-function of the lens spreads the image charge over several pixels is taken into account, the number of electrons in the peak pixel of a stellar image is \( e_{\text{s}} e_{\text{f}} e_{\text{w}} e_{\text{v}} e_{\text{s}} \Delta A \Delta \lambda \tau \), where \( e_{\text{s}} \) is the fraction of a stellar image in the highest pixel of the image (see Figure 4.4 for a schematic illustration of the transmissivity of the ETC CCD camera system). Thus, given a detected signal of \( S \) electrons above the sky+bias level in a CCD, the value of \( N \) can be calculated as \( N = S / e_{\text{s}} e_{\text{f}} e_{\text{w}} e_{\text{v}} e_{\text{s}} \Delta A \Delta \lambda \tau \). For visual magnitudes, the value of \( N \) can be related to \( m_v \) by the formula \( N = 1.05 \times 10^3 10^{-0.4 m_v} \) photons/cm²/s/Å.

The sky rate, \( B \) electrons per pixel per second, can be calculated from the sky brightness, \( H \) photons/cm²/s/Å², as \( B = e_{\text{f}} e_{\text{w}} e_{\text{C}} A H \Delta Q \Delta \lambda \) electrons/pixel at the CCD, where \( \Delta Q \) is the solid angle subtended per pixel. The value of \( H \) is 2.35 \times 10^{-6} \) in V-band (5000Å to 6000Å).

The values of the above variables projected for the full-up ETC are tabulated in Table 4.1 along with the expected ETC threshold sensitivity to detected events, as calculated from equation 4.2. The plenary ETC can be expected to detect a flash at a signal-to-noise ratio of 10 at visual magnitude \( m_v = 10.3 \) in a one-second exposure and at 11.3 in a four-second exposure. (A 10\( \sigma \) flash detection in the ETC is achieved by 7\( \sigma \) flash detections in each of two cameras in the ETC; a 7\( \sigma \) flash detection corresponds to a level of confidence of 0.9999997 in a single pixel (this level of confidence does not correspond directly to that given by Gaussian statistics: cf. Chapter
Figure 4.4: This figure schematically represents the path of photons incident to the Earth's atmosphere in reaching the ETC as digitized data. The values given here are those corresponding to the ETC test unit (see Chapter 11).
11 and Figure 11.3)). Note from Table 4.1 that the a posteriori signal-to-noise ratio of a detected event is greater than the detection signal-to-noise ratio, because the signal electrons from the full stellar image can be taken into account. These a posteriori detection sensitivities are also listed in Table 4.1, under "full image sensitivity".

4.4. The Sky Coverage of the Plenary ETC

The ideal arrangement of the ETC's sixteen 15x20 degree fields-of-view on the night sky is one that maximizes the observed solid angle coverage of the CCD cameras while minimizing atmospheric effects. The total solid angle is maximized by minimizing the overlap of adjacent fields. Atmospheric effects, which include a reduction of atmospheric transmissivity and an increase in image and field distortion, increase with zenith angle. Atmospheric effects are, in general, minimized by minimizing the mean angular distance of all fields-of-view from the zenith.

The mapping of the sixteen ETC fields-of-view is dependent on the structural constraints of the ETC instrumentation. The sixteen cameras at each site are divided into eight pairs: each pair of cameras is structurally required to point at the same right ascension. The relative right ascensions of all pairs is continually variable, as is the declination of each individual ETC camera.

The problem of finding an analytic method to find the ideal orientation of the fields-of-view was presented to a graduate
### Table 4.1: Optical Characteristics of Plenary ETC

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>25 mm</td>
</tr>
<tr>
<td>f</td>
<td>0.85</td>
</tr>
<tr>
<td>A</td>
<td>6.8 cm²</td>
</tr>
<tr>
<td>Q</td>
<td>9.55'²</td>
</tr>
<tr>
<td>Δλ</td>
<td>4500 Å - 7500 Å</td>
</tr>
<tr>
<td>H</td>
<td>2.4x10⁻⁶ ph/cm²/s/Å</td>
</tr>
<tr>
<td>ea</td>
<td>0.85</td>
</tr>
<tr>
<td>ef</td>
<td>1.0 (no filter)</td>
</tr>
<tr>
<td>ea</td>
<td>0.95</td>
</tr>
<tr>
<td>e_a</td>
<td>0.53 e⁻/photon</td>
</tr>
<tr>
<td>g</td>
<td>12 e⁻/adu</td>
</tr>
<tr>
<td>σR</td>
<td>25 e⁻</td>
</tr>
<tr>
<td>e_r</td>
<td>0.7</td>
</tr>
</tbody>
</table>

| 10σ event sensitivity (1s) | 10.3 |
| 10σ event sensitivity (4s) | 11.3 |
| 10σ event sensitivity (10s) | 11.8 |
| 10σ full image sensitivity (1s) | 10.7 |
| 10σ full image sensitivity (4s) | 11.7 |
| 10σ full image sensitivity (10s) | 12.2 |

Table 4.1: Optical Characteristics of Plenary ETC

mathematics seminar led by Professor Dan Kleitman, the chairman of the mathematics department at MIT. This group concluded was that a simple analytic solution was not possible: they suggested that an adequate solution could be found by trial-and-error, mapping the fields-of-view onto the night sky by hand according to the structural
constraints. Figure 4.5 shows the result of such a trial-and-error analysis: the proposed layout of the sixteen ETC camera fields-of-view.

4.5. Survey Capabilities of the Plenary ETC

The ETC will operate under photometric conditions during dark and grey time (in the period between third and first quarters of the Moon) at Kitt Peak. Thus, the ETC could potentially observe ~180 nights per year. Given that the mean night is nine hours long, if half of these nights are clear enough to allow observations, then the ETC will observe roughly 820 hours per year. Combining this with the observed ETC efficiency of ~75% (see Chapter 11) leads to the estimate of the time the ETC is actually observing per year of 616 hours.

The total observing solid angle of the plenary ETC is $16 \times 282^2 = 4512^2 = 1.37$ steradians. The total actual solid-angle-time product of the plenary ETC, taking all above considerations into account, is 850 sr-hours per year.

An estimate of event rates from known sources of celestial optical flashes can be made from the estimate of the observing time per year of the plenary ETC and the analysis presented in Chapter 10. Specifically, the value of 850 sr-hrs observed per year yields direct estimates of event rates from Figure 10.6. Given the sensitivities of event detection listed in Table 4.1, the ETC can be expected to detect optical flashes from GRBs at a rate of 0.5 per year for one-second exposures, 4 per year for four-second exposures and 8 per year
Figure 4.5: Possible layout of the plenary ETC camera fields-of-view on the night sky. The view shown is a zenith projection from Kitt Peak. Coordinates inside each field are the coordinates of the field center. Each field-of-view has an angular size of $15 \times 20$ degrees.
for ten-second exposures. In addition, the ETC should detect 14, 420 and 3800 flare stars per year in one-, four- and ten-second exposures, respectively, assuming a mean flare risetime of 30 seconds (see Chapter 10).

* Assuming the duration of the optical burst to be five seconds.
CHAPTER 5

ETC Operations

Introduction

The plenary ETC is intended to be a completely independent, automated instrument completely under computer control. It must be able to think and react with the flexibility of a human observer, but more quickly and more consistently. In addition, the operation and data storage of the ETC must occur smoothly and efficiently. As a result, controlling software for the ETC instrument is quite complex. Through its interactions with the peripheral computers, the ETC software must control every aspect of ETC operations. In addition, it must operate in a fashion that assures that any and every failure mode which we can possibly anticipate is rendered "failsafe".

This chapter is intended to give the reader an overview of ETC operations, ranging from the highest (day-to-day) level to the lowest (second-to-second) level. (Extensive details of the ETC software are given in chapters 7, 8 and 9.)

5.1. ETC Instrument Control

All aspects of ETC operations are controlled by computer. In the plenary ETC, the control system will be set into operation by a human user, and then operate for months at a time with no significant human interaction. The only exceptions to this complete computer control are 1) when the system detects an instrument failure and
requests human help, 2) the periodic replacement of magnetic storage media and 3) the periodic cleaning of optical surfaces.

The computers controlling the ETC are divided into three distinct units. Each computer has a specific set of tasks and works independently of the others. The master computer is a small, powerful Motorola 68000-based computer known as the Overseer computer. The Overseer computer is responsible for controlling and coordinating ETC operations and thus is in a very real sense the "brain" of the ETC.

The Overseer computer controls ETC hardware operations and data flow through two sets of slave computers. The first, the Instrument Control Electronics (ICE), is the Overseer computer's link to the system's CCD cameras and most of the ETC instrumentation. The ICE is truly a slave computer, having no independent control over any part of the system: it simply executes commands given by the Overseer computer.

The second, the Trigger processor, on the other hand, is more of an "idiot savant": its primary responsibility is to analyze incoming CCD data quickly and efficiently. Each Trigger processor analyzes data from a single ETC camera: the plenary ETC will therefore include 32 Trigger processors. Each Trigger processor accepts successive images from its associated camera and examines them for flash events. Any potential events detected by a Trigger processor is reported to the Overseer computer at the time of detection.
ETC operations consist of the reading out and analysis of contiguous, precisely-timed exposures of the night sky. Data flow in the ETC is initiated by the Overseer computer, which, at the end of an exposure, commands the ICE to read out all CCDs. The CCD image data is amplified and digitized in the ICE and then sent to the Trigger processors, which analyzes the image for any significant brightenings by, among other things, comparing it to the preceding exposure. Reports of brightening are sent from the Trigger processor to the Overseer computer over RS-232 serial link, while image data is transferred over a custom high-speed serial link (HSSL) at a rate of 250 kpixels/second. After the analysis and any data storage are completed, the next image is read out for analysis.

5.2. A Typical Observing Night

A typical observing night for the ETC is moonless and nearly photometric, and is about nine hours long. The ETC begins operations shortly before astronomical twilight, when an on-board clock alerts the Overseer computer that it is time to observe. The Overseer computer then commands all sidereal drives to slew to their designated starting points, generally two hours east of the meridian, and then to begin tracking the sky. The Overseer computer then checks all weather sensors to make sure that it is safe to open the protective dome. If so, the Overseer computer opens the dome with a command to the ICE (a flow diagram of the setup operation is given in Figure 5.1).
Figure 5.1: Flow chart of the setup of ETC operations
Once the dome is open, the Overseer computer analyzes images of the night sky for patchy clouds or an extended cloud cover, since any clouds in the field-of-view can create false optical flashes by briefly covering a star in an ETC field. The Overseer computer compares the brightnesses of stellar images in a CCD exposure to their expected brightness: if the transmissivity of the atmosphere is high enough (>~80%) and if no individual stars are significantly dimmer than expected (e.g., due to patchy clouds), the Overseer computer initiates observations.

A single observation cycle consists of monitoring a given patch of sky for about two hours, after which the sidereal drives will be commanded to slew back to their initial position. A new observation cycle then begins, with all cameras observing a new patch of sky. This method assures that the cameras will always be observing at or near the meridian, so that the effects of atmospheric extinction are minimized.

5.2.1. An Observation Cycle

An observation cycle consists of a series of short (1-4s) contiguous exposures of the night sky taken simultaneously by all CCD cameras. (A flow diagram of an exposure cycle is given in Figure 5.2; a timing diagram of an ETC exposure can be found in Figure 5.3). The exposures are timed by the Overseer computer's high-precision countdown timer (the frame timer). Near the end of an exposure, the Overseer computer notifies all Trigger processors that data transfer is imminent. When the frame timer expires, the Overseer computer
Figure 5.2: Flow chart of an ETC exposure cycle during observations.
Frame timer expires | Sift timer expires
---|---
Reports of any events detected by a Trigger Processor are sent to the Overseer computer | The Overseer Computer may use this time to acquire image data from the Trigger Processors

- t=0.0 s
- CCD readout complete
- Sifting starts
- t=1.0 s
- Sifting complete
- t=2.0 s
- Frame timer expires

The Overseer Computer sends an "end-queries" command to all Trigger Processors and then commands the ICE to read out all CCDs.

Figure 5.3: ETC exposure cycle timing diagram. This figure schematically describes the sequence and timing of events during a single exposure cycle. Soon after the CCDs are read out, each Trigger processor begins the process of analysis (sifting) of the incoming CCD image data. Any event detected by a Trigger processor are reported to the Overseer computer, which tabulates and correlates them as they come in. Sifting continues until completion or until the sift time expires, whichever comes first. After the sift timer expires, the Overseer computer has time to request data from the Trigger processors. Immediately before the frame time expires and the CCDs are to be read out, the Overseer Computer issues an "end queries" command to the Trigger processors, notifying them of incoming data. (See text for details).
commands the ICE to read out all CCDs. The CCD images from each camera are transmitted to the corresponding Trigger processor: each Trigger processor then analyzes its image for brightenings by, among other things, comparing the image to its direct predecessor (see Chapter 8 for details). Any potential flash candidate noted by a Trigger processor is reported to the Overseer computer in the form of a candidate report, which lists the location on the CCD of the potential flash. The Overseer computer converts this location to celestial coordinates, and stores the coordinates in a table. By comparing the celestial coordinates of all candidate reports from a given exposure, it can use a coincidence requirement to determine whether a legitimate celestial flash has been detected.

Any confirmed celestial flash is added to the Overseer computer's active flash list. In the time after the analysis of a set of images by the Trigger processors and before the next set of images is read out, the Overseer computer collects and stores image data from each flash in the active flash list. This data, which consists of small subarrays (9x9 pixels) centered on the locations of the flash and several nearby photometric standards, is transferred from the appropriate Trigger processor to the Overseer computer at the request of the Overseer computer. If the Overseer computer determines that a flash has subsided to its pre-event brightness, it removes the flash from the active flash list: after this point, no further data from the flash is collected.
The Overseer computer will report the coordinates of any confirmed celestial flash to the RMT (see section 4.1.2), which will immediately slew to the coordinates of the flash and collect data from the flash until the flash brightness has subsided below the detection threshold of the RMT.

This series of exposures will continue until the end of the observation cycle. The Overseer computer will automatically interrupt the observation cycle periodically in order to a) check the sky conditions by measuring atmospheric throughput, b) recalibrate the mapping of CCD location to celestial coordinates (see section 7.1.4.2) and c) store onto magnetic tape any data taken in the last observing period.

In addition, observations can be unexpectedly interrupted for the following reasons:

1) Data has been taken from the maximum allowable number of flashes. If more than a certain number of flashes (presently 6) are detected in any observing period, the Overseer computer must interrupt operations in order to store this data on magnetic tape.

2) The weather sensor reports impending precipitation. A protective dome is then automatically closed and the Overseer computer halts operations.

5.2.2. End of the night

When the Overseer computer's onboard clock reports that the observing night is over (generally shortly after morning astronomical twilight), the Overseer computer will shut the system down. The last data are stored onto magnetic tape, the mounts are slewed to the
meridian and locked into position and the dome is closed. The Overseer computer can then proceed with any daytime activities, which potentially include some further reduction of stored data, the analysis of cosmic-ray interactions with the CCDs or the transmission via modem of collected data to MIT. Otherwise, the Overseer computer waits for night to fall, when the cycle begins anew.
CHAPTER 6

The ETC Test Instrument

Introduction

The completed ETC will consist of 32 CCD cameras and their associated hardware located at two sites on Kitt Peak. The instrument described in this thesis is a sub-unit of the plenary ETC, intended to test the concept and design of the Explosive Transient Camera as a whole. The expansion of this instrumentation to the complete, 32-camera ETC requires the construction of several copies of this sub-unit and their integration with the ETC system as a whole.

The test unit is a four-camera "mini-ETC" set up at Site 1 at Kitt Peak (see Figure 4.1). The test unit is controlled by a complete Overseer computer (the same Overseer computer which will control the plenary ETC). The CCDs and ETC instrumentation are controlled by a set of Instrument Control Electronics (ICE), which respond to commands issued over RS-232 serial link by the Overseer computer. Image data from the four CCD cameras is analyzed for sudden brightenings by four parallel Trigger processors, which communicate with the Overseer computer over RS-232 serial links. (A schematic layout of the data and communications paths in the prototype ETC can be found in Figure 6.1). The ETC sub-unit is completely functional, and through the Overseer computer and Trigger processor software is able to conduct observations in a semi-automatic mode.
Figure 6.1: A schematic view of the ETC test unit command and data flow. The Overseer computer controls the peripheral Trigger processors and Instrument Control Electronics over RS-232 serial links. Mass data transfer is made over a custom high-speed serial link (HSSL) at 2 Mbits/second.
The ETC Test Unit

The ETC test unit was constructed between May, 1983 and October, 1984, at the Center for Space Research's Balloon Laboratory at MIT. The test unit consists of four cooled-CCD cameras mounted on a sidereal drive, sharing a common vacuum and cooling source.

The test unit's CCD cameras were based on an existing LN$_2$-cooled CCD camera used in X-ray experiments at MIT. As the ETC is intended to be a completely automatic sky-monitoring instrument, the use of expendibles (such as LN$_2$) which require daily human maintenance must be avoided. As a result, a closed-cycle refrigerator, which can run unattended for months or years without maintenance, is used to cool the CCD cameras.

Detailed design for several important elements of the ETC test unit was performed by DFM Engineering, Inc., of Longmont, Colorado. The cameras, support structure and sidereal drive were all constructed by DFM Engineering, Inc. The test unit's CCD cameras, each with a field-of-view of $15^\circ \times 20^\circ$, are mounted on a single sidereal drive in order to track the sky during an observation. The interface between the cameras and the drive is the manifold: the cameras are mounted to the manifold, which in turn is mounted to the sidereal drive. The expansion probe of the closed-cycle refrigerator is situated on the axis of the manifold: the CCDs in the cameras are kept at a temperature of $\sim-85^\circ$C by thermal contact to the expansion probe.
A picture of the ETC test unit is shown in Figure 6.2, and a sketch of the manifold and cameras, giving views of the insides of both, is in Figure 6.3. A sketch of the ETC prototype building and the layout of the instrument in the building can be seen in Figure 6.4. Detailed descriptions of each aspect of these figures can be found in the figure captions and in the text of this chapter.

The following sections describe each part of the ETC test unit in some detail.

6.1. The ETC CCD Camera

The ETC CCD camera consists of a cooled CCD placed at the focal plane of a 25mm lens. A vacuum-tight aluminum camera body houses the CCD, its thermal-control heater unit, its protection electronics and an output signal preamplifier. Signals from the Instrument Control Electronics (ICE) reach the CCD through the connectors in the camera back plate. (A sketch of an ETC CCD camera is included in Figure 6.3).

6.1.1. Camera Body Construction

The camera body is cylindrical, measuring 7" length by 6" diameter. Four bolts through the side of the camera body fasten the camera body to the manifold. Small (1.5 inch) holes in the camera body and manifold at the camera/manifold interface allow the manifold and camera to share a common vacuum cavity as well as create room for a thermal path from the CCDs to the cooling probe inside the manifold.
Figure 6.2: A view of the ETC instrument during test operations. The four ETC cameras are mounted on the manifold, which is mounted to the sidereal drive. The vacuum hose at lower left runs from the ion pump (not in picture) to the base of the manifold. In this picture, the four cameras are outfitted with three 75 mm lenses and one 25 mm lens. (See Figure 6.4 for a complete schematic overview of the ETC instrument.)
Figure 6.3: Cross-sectional view of the ETC prototype instrument. View is of manifold and four cameras, showing various aspects of the instrumentation.
Figure 6.4: Schematic representation of the ETC test instrument within its building at Site 1 on Kitt Peak.
The CCD rests on an aluminum cold sink, which makes thermal contact with the cooling probe through a braided copper strap, as described in section 6.3. The temperature of the cold sink (and therefore of the CCD) can be measured with a thin-film, temperature-sensitive resistor mounted on the cold sink. The temperature of the cold sink can be raised by passing current through a 25 ohm, 10 Watt power resistor mounted on the cold sink.

A thin, triangular aluminum plate with a rectangular opening for the CCD imaging area presses the CCD against the cold sink, assuring good thermal contact. This triangular plate is mounted on a structure of thin-walled stainless steel tubing extending up from the back plate of the camera. The thinness of the wall and length of the tubing create a large thermal resistance between the CCD and the back plate.

Electrical connections to the camera contents are made through two hermetically-sealed connectors in the back plate of the camera. A 32-pin connector feeds the CCD clocking signals, power to the preamplifier and the signals to and from the CCD temperature sensor and heater from the ICE to the CCD camera. A 6-pin connector feeds the preamplified CCD output signal out of the housing.

6.1.2. The ETC Optical System

The CCDs are Texas Instruments virtual-phase devices, consisting of an array of 390 x 584 22.3 μm pixels. They are divided into imaging and memory halves, and are operated in "frame-store" mode. In
frame-store mode, an exposure made in the imaging area can be clocked quickly (in a few milliseconds) into the memory area. Then, as the first image is being read out into the processing circuitry at a slower rate (in about 0.5 seconds), the imaging area can be collecting photons from the next exposure, assuring a duty cycle of nearly 100%.

The camera lens used in the ETC survey is a commercial 25mm, f/0.85 CCTV lens manufactured by Kowa, Inc, of Japan. It is a wide-field lens (focal plane scale = 2.3 degrees per millimeter) with a large collecting area (6.8 cm$^2$) due to its low f-number. Its back-focal-distance, unfortunately, is quite low (4.5 mm), thereby requiring that the CCD imaging area be within 1.5 mm of the front plate of the camera.

The chromatic aberration of the lens allows good focus to be achieved only with the use of a filter (typical $\Delta \lambda = 1000$ Å). Even then, the point-spread-function of the lens is such that a point source is imaged onto two to four pixels at image center. The off-axis reduction in response of this lens is also significant: the vignetting of the lens is ~30% at an image radius of 5 mm. The average loss of transmissivity over the CCD due to vignetting is ~15%. Typical images made by the ETC are shown in Figure 6.5.

6.1.3. The Internal Electronics of the CCD camera

CCD clocking signals entering the camera body pass through the internal protection electronics. The protection electronics prevent
Figure 6.5: Two ten-second exposures made with ETC camera C. The images were made through a) the ETC 25 mm lens, as described in section 6.1.2, and b) a high-quality 75 mm lens. The vertical streaks visible in Figure 6.5a are due to a column defect in the CCD in camera C.
voltage spikes on CCD clocking lines from reaching the CCD. The CCD output signal is fed directly into the preamplifier inside the CCD camera. The preamplifier has a gain of ~25, and its output travels to the analog signal processor board in the ICE through the 6-pin connector in the camera back plate.

6.2. Manifold

The manifold is a hollow aluminum cylinder, 24" long by 4" diameter, flattened on two sides. Two holes in each flattened side locate the camera mounting points. A unique design of the interface between camera body and manifold allow for a continuous rotation of the cameras parallel to the flattened face while holding vacuum, so that the declination of each camera is continuously adjustable.

The cooling probe of the closed-cycle refrigerator is located along the axis of the manifold. The mounting flange of the expansion probe of the closed-cycle refrigerator mounts with an O-ring seal to the southern end of the manifold. To prevent cantilevering, the northern end of the expansion probe is supported by means of a short thin-walled stainless-steel tube (for a large thermal resistance) between the tip of the probe and the end cap of the manifold.

The manifold is bolted to a short aluminum spacer which is bolted onto the disk of the sidereal drive. The flexible hose from the cooling probe is wrapped once around this space to assure that the stress on the hose from the motion of the drive is longitudinal, since the hose is very sensitive to axial twists.
6.3. CCD Cooling System

The cooling probe is the expansion head of the MFC-100 freon-based closed-cycle refrigerator manufactured by FTS Systems, Inc. The MFC-100 is a 1/2 horsepower, two-stage closed-cycle refrigerator, quoted to deliver ~200 Watts of cooling power at -80 C. The MFC-100 pumps liquid freon through a flexible line to the expansion head, where it evaporates and cools the head. The gaseous freon returns along a second, parallel flexible line to the refrigerator's condenser.

Thermal contact from the expansion head to the CCD cold sinks is made at four points on the expansion head which appear at the camera mounting ports on the manifold. Short (4") threaded copper rods screw into tapped bosses on the expansion head and extend 2" into each camera housing. A short metal strap completes the thermal path from the end of the copper rod to the CCD cold sink (see Figure 6.3).

6.4. Vacuum System

The ETC camera and manifold system is kept at a pressure of ~4x10^-6 Torr by a 20 l/s Vacion pump. Rough vacuum (~5 microns Hg) is achieved using a standard LN$_2$-trapped oil roughing pump; a sorption pump reduces the pressure to ~10^-5 Torr, where the ion pump is safely started.

The pressure of the system is reduced significantly when the refrigerator is in use, since the expansion probe acts as a cryopump, adsorbing many of the molecules leaking into the system. On the
other hand, when the electronics inside the camera head are powered up, the outgassing of the warm elements of the electronics raises the pressure a small, but noticeable amount ($\sim 2 \times 10^{-6}$ Torr).

The ion pump is mounted at the back of the sidereal drive. A flexible metal hose connects it to the manifold. Vacuum access to the manifold is a 2.75" Conflat flange located at the southern end of the manifold.

6.5. Sidereal Tracking Drive

The sidereal drive is a polar-mount tracking drive for small telescopes designed and built by DFM Engineering of Boulder, Colorado (this drive is in its concept a twin to the 2.4m telescope recently constructed at the McGraw-Hill Observatory on Kitt Peak). The unit has a 14" steel disk which is friction-driven by one of the two rollers upon which it rests: one roller is driven by the tracking motor, the other by the slewing motor. The manifold bolts directly to the steel disk.

The tracking drive motor is a standard 1 RPM DC motor. The rate of tracking can be varied by a thumbwheel switch mounted on the drive. The slewing motor is a 0.88 amp Slo-Syn stepper motor, which is controlled by a stepper motor controller circuit in the ICE. The maximum slewing rate is greater than 40 degrees/minute. Both motors and the clutches associated with each are controlled by the Instrument Control Electronics.
The angle of the friction disk about the polar axis can be measured through a synchro shaft encoder mounted directly onto the southern end of the polar axis of the drive. The shaft encoder delivers three 400 Hz signals, mutually 120° out of phase, to a synchro-to-digital converter (SDC) in the ICE. The SDC can calculate the absolute angle of the shaft encoder to 14-bit precision from the relative amplitudes of the three signals.

Electrical connections between the ICE and the tracking drive pass through a 14-pin cable. This cable contains signals for clutch control, motor control, synchro input and output. The drive is powered with standard 115 VAC; the stepper motor is powered by a 24 Volt, 6 amp power supply located near the sidereal drive.

6.6. Thermal Analysis of the ETC

The operating temperature of the CCD is determined by the cooling rate of the closed-cycle refrigerator and the ambient warming of the chips and how each varies with temperature. The rate of cooling is determined by the cooling power of the MFC-100 and the thermal resistance of the metal strap between the cold copper rods and the CCD cold sink. The ambient warming of the chip is due to heat gain through conduction and radiation from the walls of the camera. The final temperature of the CCDs cannot be calculated exactly because the variation of the cooling power with final temperature is not known. The various rates of heating and cooling are analyzed in Appendix F. This analysis shows that the heat gain by the cold surfaces in the CCD cameras is insignificant when compared to the
cooling power of the MFC-100 at $-80^\circ$C. Therefore, the minimum temperature of the CCDs should be well below $-80^\circ$C.

6.7. ETC Site 1

The ETC test unit is located in the former twelve-inch Schmidt observatory, a two-story circular building on Kitt Peak, near Tucson, Arizona. The second story is covered with a 16' dome with a 42" slit. The angle subtended by the open slit is $\sim 24^\circ$, (as seen by the ETC cameras), which is slightly smaller than a single ETC camera's field-of-view. Because the diagonal angular length of an ETC field is $25^\circ$, partial occultation by the dome can occur for some camera orientations. The ETC test instrument views the sky through this slit: the dome is rotated at appropriate intervals to prevent substantial occultation of a camera's field-of-view by the dome. For the plenary ETC, the dome will be replaced by a roll-off roof to allow coverage of the entire night sky. Schematic views of the ETC dome and contents can be seen in Figure 6.4.

The ETC instrument is mounted on a 2' x 2' concrete pier in the second story. The Instrument Control Electronics and ion pump are located near the test instrument in the second story. The vacuum roughing stages are located on the first floor and are plumbed to the upstairs vacuum through a network of vacuum plumbing. The closed-cycle refrigerator is located on the first floor: the flexible hose to the cooling probe feeds through a 6" hole in the floor between.
The first floor is primarily occupied by the Overseer computer, Trigger processors and all user peripherals (e.g. color graphics display, printer, terminals, magnetic tape drives, etc.). Cables leading to the ICE are fed through holes in the ceiling.
CHAPTER 7

The ETC Overseer Computer

Introduction

The Overseer computer is a fast, powerful microcomputer that is the center of organization for the ETC. It controls the ETC through software written in C. Through RS-232 serial links, the Overseer communicates with and controls the peripheral instrumentation of the ETC. The Overseer computer is capable, through its sophisticated control software, of making the ETC a completely automated observational instrument.

This chapter describes the Overseer computer hardware and software in some detail. The full complement of Overseer computer software is located in Appendix A.

7.1. Overseer Computer Hardware and Peripherals

The Overseer computer is a powerful microcomputer based on the single-board, Motorola 68000 microprocessor. It is the third "generic" computer developed and constructed at the Center for Space Research at MIT for use in X-ray and optical astronomy. The Overseer computer consists of eight Multibus computer boards, which, together, control the Overseer computer operation, data flow and several peripheral units responsible for data storage and display.
The Overseer computer consists of the following hardware:

1) A PM68K 8 MHz, 68000-based CPU board.

2) A Rodime R0204 20 Mbyte, 5.25" Winchester disk.

3) A Konan Taisho disk-controller board, which controls the system's Winchester and floppy-disk drives.

4) A Texas Instruments TMM40020-04 512 kbyte memory board, used as CPU core memory.

5) A high-speed serial link (HSSL) receiver board, used for the high-speed (2 Mbits/second) acquisition of image data from the Trigger processors.

6) A second TMM40020-04 memory board, used as dedicated memory for the HSSL.

7) Two Central Data Corporation octal serial I/O boards. Through these boards, RS-232 serial connections can be made to sixteen peripheral units, including terminals, modems and other computers (specifically, the ICE and Trigger processors — see chapter 5). Data rates can be varied between 110 and 19200 baud.

8) An Omnibyte OB68K230 parallel I/O board, which allows for communication with any peripheral over parallel interface.

9) A Ciprico Tapemaster tape drive controller board, which controls the Cipher tape drive used for mass storage of image data.

10) An AED 512x512 color graphics monitor for the inspection of CCD image data. Images are transferred to the AED over parallel interface.

11) A GC-1000 WWV receiver. The GC-1000 receives WWV signals and can communicate the time to the Overseer computer via serial link.

12) A Tandon 101-4 5 1/4" floppy disk drive.

13) A Cipher F880 1600 bpi streaming magnetic tape drive.

The Overseer computer runs Pacific Microcomputer's Unix Version 7 operating system. Programs written in C and 68000 assembler
control all aspects of the Overseer computer. The high-speed serial link (HSSL) receiver card, the only completely custom-built card in the Overseer computer, is the receiving end of a high-speed data link developed at MIT. The transmitters are mounted on the Triggers and are used to transfer image data to the Overseer computer at ~250 kpixel/second. More details on the Multibus HSSL can be found in section 8.1.2. The HSSL has its own 512 kbytes of RAM for image data storage.

Three custom-built circuits on the OB68K230's prototyping area play key roles in the operation of the ETC. One circuit is used to transmit image data to the AED color graphics unit. A second circuit makes use of signals available on the OB68K230 board. These signals, when sent to a Trigger processor, reset the Trigger processor to its start-up state. The third circuit taps a high-precision (4 μs) timer on the OB68K230 board and makes it available to the Overseer computer for the precise timing of CCD images (the frame timer).

7.2. Interprocessor Communication

The Overseer computer communicates with both the Instrument Control Electronics and the Trigger processors in its execution of ETC operations. All communication with the ICE and the Trigger processors takes place over RS-232 serial link at 4800 baud. The Overseer computer has the control software for the ICE and the Trigger processors stored on disk: when necessary, the Overseer computer downloads the software to the particular unit over RS-232 serial link. A schematic diagram listing possible Overseer computer commands or
requests to the Trigger processors and the ICE is given in Figure 7.1.

7.2.1. Overseer computer-ICE communication

The primary communication between the Overseer computer and ICE is in the form of commands to the ICE. Most Overseer computer commands to the ICE require no response from the ICE. In the interest of time, there is no handshaking between the Overseer computer and the ICE in time-critical situations (since the Overseer computer can generally establish independently whether a command to the ICE was properly executed). The only situation where a handshaking by is used is to inform the Overseer computer that a command which takes substantial time to execute (such as the slewing of the sidereal drive) has been completed.

7.2.2. Overseer Computer-Trigger Processor Communications

Communications from the Overseer computer to a Trigger processor are divided into two types: those during the analysis of image data by the Trigger processors ("sift" mode -- see section 8.3.1) and those after the analysis, where the Overseer computer can make requests for data from the Trigger processors ("query" mode).

Sift mode in the Trigger processors is initiated by a command from the Overseer computer immediately before reading out the CCDs. In sift mode, the Trigger processors analyze incoming data and report any events to the Overseer computer via serial link. The Overseer computer's job during sift mode is to listen and wait for "candidate
Figure 7.1: Schematic summary of Overseer communications with the ICE and Trigger Processors.
reports" and the analysis-ending "sift-termination report". (For more detail on Trigger processor analysis procedures, see Chapter 8). A single handshaking byte is returned to the Trigger processor after every candidate report for efficiency.

After transmitting the sift-termination report, which marks the end of sift mode in the Trigger processors, the Trigger processors are automatically in query mode and are open to requests from the Overseer computer. The commands from the Overseer computer now take the form of requests for data or organizational commands. The requests for data include:

1) Send an image subarray of any size from any part of the CCD.
2) Send 9x9 image subarrays from a predefined location corresponding to either a flash or a standard star.
3) Send an entire image.
4) Send the row and column vectors whose vector sum make up the threshold image (cf. section 8.3.2.1 and Appendix E).

All image data is transferred to the Overseer computer over the high-speed serial link at 2 Mbits/second.

The organizational commands from the Overseer computer to the Trigger processors include the following:

1) Assign to a given location (x,y) on the CCD a number corresponding to a standard star image located at that spot on the CCD.
2) Calculate a threshold frame from the frame presently in memory (cf. section 8.3.2.1 and Appendix E).
3) Set the parameters used in the analysis of image data by the Triggers: sift time, threshold offset, and brightened
7.3. Overseer Computer Utility Routines

Overseer computer utility routines are larger subsections of the Overseer computer software which are too detailed to be included in the general overview of chapter 4, yet play a major role in the operation of the Overseer computer during observations. A description of these routines follows.

7.3.1. Timekeeping

The Overseer computer has two system clocks. The first clock, associated with the CPU board of the Overseer computer, keeps year, month, day and time to a precision of 5 milliseconds. This clock is used for recording the time of events. It can be set by the user or, via software, with the GC-1000 WWV receiver. The GC-1000 clock receives WWV time signals and can transmit the precise date and time over RS-232 serial link on command. Thus, the Overseer computer can set the day-date clock from information received from the GC-1000 WWV clock.

The second system clock is used for timing CCD exposures: the Overseer computer's day-date clock is inadequate for this task because of low precision and its long access time. For these reasons, the 10 MHz clock from the Overseer computer's parallel I/O board (the OB68K230) is used for exposure timing. The OB68K230's timer has a precision of 4 microseconds and can act as a countdown timer, so that once the timer is set (for example, at the beginning
of an exposure), it will count down until the time has expired, and then restart itself. The Overseer computer only needs to watch the timer and read out the next exposure when the frame timer expires.

Time in the ETC is stored in "modified Julian day" format for simplicity: a four-byte modified Julian day gives the year, date and time to a precision of one second. The modified Julian day is the Julian day minus 2400000.5 (Nautical Almanac, 1985): for reference, 12.00 UT on 1 June 1985 is modified Julian day 46216.5.

7.3.2. Astrometry

The Overseer computer must have a precise knowledge of the mapping of pixel location to celestial coordinates in each CCD in order to judge the validity of an optical flash based on whether it has the same celestial coordinates in two cameras. The Overseer computer can make a reasonably good estimate of this mapping given the details of the optical system: the focal length of the lens, the pixel size on the CCD, any first-order distortions in the lens and the angle of the rows and columns of the CCD with respect to lines of right ascension and declination. The sub-pixel precision necessary for the ETC is achieved by calculating the parameters of the astrometric mapping using the centroided location of several (~20) SAO standard stars in the field of the CCD and the celestial coordinates of those stars.

The mapping used in the ETC is a simple linear relation between pixel coordinates \((x,y)\) and the reduced coordinates \((\xi,\eta)\). The reduced coordinates are calculated from the celestial coordinates
\((a,b)\) in the tangent-plane approximation, given the celestial coordinates of the field center \((a_0, b_0)\), as in Podobed (1965, p. 180).

\[
\xi = \frac{\cos(b_0)\sin(a-a_0)}{\sin(b)\sin(b_0)+\cos(b)\cos(b_0)\cos(a-a_0)} \quad (7.1a)
\]

\[
\eta = \frac{\sin(b)\cos(b_0) - \cos(b)\sin(b_0)\cos(a-a_0)}{\sin(b)\sin(b_0)+\cos(b)\cos(b_0)\cos(a-a_0)} \quad (7.1b)
\]

The mapping of pixel coordinates \((x,y)\) to reduced coordinates \((\xi,\eta)\) is

\[
x = a_1 + b_1 \xi + c_1 \eta \quad (7.2a)
\]

\[
y = a_2 + b_2 \xi + c_2 \eta \quad (7.2b)
\]

The values of the parameters of the mapping \((a_1,2,3, b_1,2,3)\) can be roughly estimated from details of the CCD optical system. For example, if a CCD has one axis oriented east-west, then \((a_1, a_2)\) are the coordinates of the center of the CCD, \(b_1\) and \(c_2\) are the focal-plane scale and \(b_2 = c_1 = 0\).

The Overseer computer calculates the precise values of the astrometric parameters from the estimated values of these parameters in the following way. The Overseer computer has a rough idea of the celestial coordinates of the center of each CCD, since the declination of each camera is specified in software and the right ascension can be calculated from the hour angle of the camera and the sidereal time. The hour angle of the camera is easily calculated from the value of the synchro shaft encoder mounted on the polar axis of the
drive: the mapping of shaft encoder units to hour angle is specified in software and does not change. Given these celestial coordinates of the center of a CCD, the rough calculation of the parameters of the astrometric mapping is sufficient to allow Overseer computer to estimate the locations on the CCD of several (~20) SAO stars to an accuracy of ~10'. In order to determine the locations of these stars on the CCD more precisely, the Overseer computer then acquires small image subarrays of each SAO star from the appropriate Trigger processor. The Overseer computer can then determine the location of each standard star on the CCD to a precision of ~0.1 pixels with a simple interpolation algorithm (see Appendix D). The Overseer computer is now able to calculate the values of the astrometric parameters from a least-squares fit of the precise locations of the SAO stars to the celestial coordinates of the SAO stars. If the least-squares fit yields an rms deviation of predicted stellar coordinates from actual stellar coordinates of more than 0.5 pixels, the star with the highest residual to the fit is discarded and the least-squares fit is repeated. After iterating the above procedure until the rms deviation is below 0.5 pixels, the celestial coordinates of any stellar image on the CCD can be calculated by 1) determining the location of the image on the CCD precisely and 2) inverting equations 7.1 and 7.2 to map the precise image location onto its celestial coordinates with a final precision of better than 0.5 pixel.

The celestial coordinates of the SAO stars used in the above procedure are found in a file on disk which contains a portion of the
SAO catalog. The celestial coordinates in the SAO catalog file are epoch 1950.0; therefore, all celestial coordinates used in the ETC are epoch 1950.0 for simplicity. The effects of stellar proper motions of the stars and differential precession across the field are insignificant when compared to 3' pixels. The effect of precession is ignored, since precession only introduces a rotation into the astrometric mapping, which is accounted for by the cross terms in the fit parameters.

7.3.3. Data Storage by the ETC

The data from flashes detected by the ETC is stored in an extensive data storage structure, diagrammed in table 7.1. The stored data includes:

1) 9x9 pixel subarrays about the flash event and local photometric standards from immediately before the detection and for the course of the flash.

2) Flash information, including the time of the flash, which CCD cameras detected it, the precise location of the flash on the detecting CCDs and the calculated flash coordinates.

During observations, the ETC stores data onto magnetic tape at thirty-minute intervals, including any data taken from detected flashes in that thirty-minute period. In addition, the ETC will always store data indicative of the operating conditions of the ETC in that period, regardless of whether flashes were detected (see table 7.2).
FLASH EVENT DATA STORED BY THE ETC

<table>
<thead>
<tr>
<th></th>
<th>Number of flashes detected. Data from all detected flashes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data stored from each</td>
<td>Number of exposures taken of the flash. Data from the flash.</td>
</tr>
<tr>
<td>detected flash</td>
<td></td>
</tr>
<tr>
<td>Data stored from each</td>
<td>Subarrays from standard stars from old, recent and current frames. Coordinates and position of flash in both cameras detecting. Threshold frame information (see Appendix E). Data from all exposures.</td>
</tr>
<tr>
<td>flash</td>
<td></td>
</tr>
<tr>
<td>Data stored from each</td>
<td>Time of the exposure Data from each camera</td>
</tr>
<tr>
<td>exposure</td>
<td></td>
</tr>
<tr>
<td>Data stored from each</td>
<td>Subarrays around flash location. Subarrays around locations of standard stars.</td>
</tr>
<tr>
<td>camera</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: List of data stored from a flash event detected by the ETC.
### ETC OPERATIONAL DATA STORED AT REGULAR INTERVALS

| Archives                                    | All system constants (see section 7.1.5.3.10)  
|                                            | Start and end times of cycle  
|                                            | Coordinates, positions and magnitudes  
|                                            | of all SAO standards used in each FOV  
|                                            | Subarrays about all SAO standards at start  
|                                            | and end of cycle  
| Camera data                                 | Right ascension, declination of field center  
|                                            | Focal length, f-number, area of lens  
|                                            | Angle of CCD, angle of camera to north-south  
|                                            | CCD gain, noise, bias level, total noise  
|                                            | Trigger processor constants  
|                                            | CCD temperature  
| Trigger processor data                      | Sift time  
| Mount data                                  | Synchro units to hour angle  
|                                            | calibration information  
|                                            | Right ascension of mount  
|                                            | Hour angle at start and end of observations  
| System data                                 | Exposure time  
|                                            | Interval at which to slew the mounts  
|                                            | Interval at which to redo astrometry

**Table 7.2: Instrument Status Data Stored by ETC**

This archived information includes all pertinent information about the configuration of the CCD cameras, readout and sift times, and throughput and vignetting information. The ETC also regularly stores subarrays around five standard stars in the field of each CCD at the beginning and end of each thirty-minute observing period as an
after-the-fact check of sky conditions.

7.3.4. The ETC Overseer Computer Software

The Overseer computer software is completely written in C. It is divided into 19 modules, each restricted to a particular aspect of the Overseer computer software. Some overlap of functions is, of course, unavoidable. Nonetheless, the modulization of the code is an attempt to enhance the organization, readability and maintainability of the code.

This section will describe each module in some detail: further details can be found in the code and documentation found in Appendix A. All module names begin with two capital letters in order so that subroutines can be identified with the modules they are found in. (The "c" suffix indicates a module written in C).

The modules are broken up into control, I/O and utility routines. Control routines are primarily concerned with the actual operation of the instrument and the flow of the data, and use utility routines to assist them. I/O routines are used to communicate with other processors.

7.3.4.1. Control Routines

The ETC control routines are listed and briefly described below:

e) STart.c is the execution module: it simply executes ENtry, which is the real entry-level module.

e) ENtry.c is the primary user-interation module. Here the user chooses whether to start automatic observations (through OPERations) or human-aided observations (through USEr), or to
do some preliminary image location work (IMage).

e) OPerations.c is the node at which the Overseer computer software is directed into its various day-to-day modes. For example, during the day OPerations will execute the "wait until dark" routine. When it gets dark, OPerations will first execute the "set up the mounts" code, followed by the observation code. Any panic or failure flags in the observation code will return control of the code to OPerations.

e) OBservations.c is the module which controls all aspects of observations. It coordinates exposures, sets up and reads from Trigger processors, watches the and looks for peripheral failures. At the end of an observation cycle, it returns control to the module from which it was entered.

e) USer.c is the semi-automatic observation control code used in observations with the ETC test unit. It executes the observations automatically, but requires user assistance during setup to position the mount, to line up the cameras properly and to determine observation thresholds.

e) IMage.c is a collection of user-interactive code used in determining the celestial coordinates of a CCD field center, in focussing a camera and in testing certain sift parameters.

7.3.4.2. I/O Routines

The three I/O modules in the ETC Overseer computer software are:

e) COsmac_IO.c contains all software used in communication with the ICE, including downloading software.

e) TRigger_IO.c contains all software used in communication with the Trigger processors, including downloading software.

e) EXternal_IO.c contains code to communicate with external instruments, such as the RMT.

7.3.4.3. Utility Routines

The ETC Overseer software's utility routines are a collection of useful subprograms which are used during the execution of ETC operations.

e) AStrometry.c consists of all astrometry routines for the ETC, including code to calculate the astrometric mapping parameters, given precise locations and coordinates of astrometric
standards, as well as code to calculate celestial coordinates of a stellar given the precise location on a CCD, and vice versa (see section 7.1.4.2).

e) DAytime.c contains code to determine whether it is day or night and code to wait for night if it is day.

f) DEclarations.c contains look-up tables regarding the cameras and their operating parameters.

g) DOme.c contains code to move and read out the location of the ETC building's dome.

h) JPD.c is a module devoted to the software contributions of John Doty. These routines include faster algorithms for time-critical ETC operations, such as a fast median-filtering routine and a fast standard star choosing algorithm.

i) PHotometry.c contains the system's photometry routine, which calculates the brightness above background of any array passed to it. It returns the sum over all pixels in the array of the difference between the brightness of the pixel and the sky+bias level in the same pixel.

j) SAo.c contains all code pertaining to the SAO stars used as standards by the ETC. It is primarily used to select SAO stars for use as standards in a given field.

k) UTility.c is a collection of various routines with general applicability. The reader is referred to Appendix A for more detail of its many features.

l) WEather.c contains weather-detection software and is presently not implemented.

m) WWv.c is a module designed to read the precise time from the GC-1000 clock into the Overseer computer's day-date timer.

n) The modules Globals.c and extern.h contain all initial declarations of system global variables.

o) The module constants.h contains all system-wide fixed constants.
CHAPTER 8

The ETC Trigger Processor

Introduction

The ETC Trigger processors are responsible for real-time data reduction of in ETC. The Trigger processors are powerful microprocessors dedicated to the quick, efficient analysis of CCD image data taken by the ETC. Each Trigger processor is responsible for the analysis of the image data from one CCD camera: the job of a Trigger processor is to cull through the CCD image data and report significant brightenings to the Overseer computer. Through the Trigger processors, the rate of data leaving the ICE (100 kbytes/second/camera) is reduced by four to six orders of magnitude before entering the Overseer computer. This efficient, dedicated method of data analysis is what makes the ETC a viable instrument for real-time astronomy at the one-second timescale.

8.1. Trigger Processor Hardware

A Trigger processor in the ETC test unit consists of three Multibus (reference) boards in its own enclosure:

1) An Omnibyte OB68K1A 10 MHz microprocessor.

2) A Texas Instruments TMM40020-04 512 kbytes memory board.

3) A custom Direct Memory Access (DMA) board with a High Speed Serial data receiver/transmitter Link (HSSL).

The data from each ETC camera is transferred to Trigger processor
memory over the HSSL receiver at ~250 kbytes/second. Image data from the CCDs is processed as it streams in. Information about an event detected by a Trigger processor is sent to the Overseer computer via an RS-232 serial link on the OB68K1A board.

8.1.1. The Omnibyte OB68K1A

The Omnibyte OB68K1A single-board processor is a Motorola 68000-based microprocessor and supports code written in C and 68000 assembler. Two on-board EPROMs provide the OB68K1A with a power-up operating program, known as ETRM (ETC Trigger ROM Monitor), accessible by the Overseer computer via RS-232 serial link. Through ETRM, memory locations and I/O registers in the Trigger processor can be accessed by the Overseer computer or a human user. The Trigger software is stored as a file in the Overseer computer and is downloaded using ETRM: the executable Trigger software is stored in RAM on the OB68K1A board via RS-232 serial link.

The Omnibyte OB68K1A board also includes a timer chip with which the Trigger processor times the data analysis process (the sift timer: see section 8.3.1). The precision of the on-board timer is better than 0.01 seconds.

8.1.2. The Multibus DMA/HSSL board

The Direct Memory Access/High Speed Serial Link receiver/transmitter board consists of discrete circuitry linked to DMA circuitry which interacts through the Multibus with the Omnibyte OB68K1A board. The DMA/HSSL board is used for fast, uninterrupted
transfer of data to and from the Trigger processor's memory. Direct Memory Access (DMA) circuitry makes possible the hardware-controlled storage of data without wait states or software control. The High-Speed Serial Link (HSSL) allows for transfer of image data at a rate of 2 Mbits/second. The Trigger processor HSSL receiver circuitry accepts image data from the ICE, while the transmitter sends requested image data to the Overseer computer. Together, the DMA and HSSL are an efficient method of mass data transfer and storage which does not require any software intervention. This leaves the Trigger software free to analyze CCD image data as it streams in, making it possible to detect events before all the data has been received.

The DMA controller is based on the Motorola 68450 DMA chip. The controller has four transmitter/receiver channels and is controlled by bitwise manipulation of the 68450 registers. In the case of incoming data, the DMA circuitry is responsible for the control and organization of data and address bytes, making sure that each incoming byte goes where it is supposed to be in RAM. For outgoing data, the DMA circuitry must acquire the data from the requested memory locations and feed them to the data transmitter circuit in the correct order.

8.1.3. Trigger Processor Memory

The Trigger processor memory is broken up into the following sections:

1) Eight kbytes of read-only-memory (ROM) on two ROM chips on the Omnibyte board. ETRM firmware (section 8.1.1) is located in this ROM, so that the the Overseer computer can communicate
with the Trigger processors through ETRM immediately after power-up.

2) 32 kbytes of RAM on the Omnibyte OB68K1A board. The executable Trigger software is stored in this memory.

3) 512 kbytes of RAM on the TMM40020-04 Multibus memory board. This memory is used for frame and data storage and is large enough to hold four full 400x292 byte ETC frames.

8.2. Overview of Trigger Software

The software for the Trigger processors was written as part of a senior thesis project by Steven Rosenthal of MIT. It includes the code setting up the OB68K1A environment, the DMA control code, the ETRM firmware (the ETC Trigger ROM Module, described in section 8.1), as well as all of the real-time analysis software.

8.2.1. Trigger processor memory allocation

The Trigger processor memory is large enough for four 400x292 CCD images. The Trigger processor 512 kbyte memory is broken up into four image regions, with the remaining 45 kbytes of memory used for data storage. The four image areas are named the current, recent, old and threshold frames. In the current frame is stored the most recent image to enter the Trigger processor (data flowing into the Trigger processor is always stored in the current frame). The recent frame generally contains the image taken immediately prior to the current frame. During sifting, the current frame is compared to the recent frame. After sifting, as is explained below, the current frame becomes the recent frame, and the recent frame is overwritten with an incoming image.
The old frame is a frame of data taken at some point and set aside in memory, expressly designated to be the old, or archival, frame. It can be used as the comparison frame during sifting, but it generally is viewed as archival information. The threshold frame is a construction which is used in the first level of sifting (see section 8.3.2.1). It consists of a sky frame with the stars artificially removed. Frames being analyzed are compared to the threshold frame. The method of calculating the threshold frame is explained in detail below and in Appendix E.

The frame labels (current, recent, old and threshold) are not fixed to a memory location in the Trigger processor, but rather are names which can be dynamically attached to any of the four areas of memory. For example, in the sifting process, the current frame is always being compared to the recent frame. When the analysis of the current frame is over and a new image is to be read in, the current frame becomes the recent frame. Rather than move the data from the current frame into the memory locations associated with the recent frame, the names of the memory locations are simply swapped: the next image read in goes into the current frame, overwriting the old recent frame, while the old current frame is the new recent frame. In this fashion, the analysis of consecutive images can proceed quickly and without lengthy transfers of data within Trigger processor memory.

8.3. Trigger Processor Operations

The Trigger software runs in one of two states. In **sift mode**, the Trigger processor waits for CCD image data to come in across the
HSSL from the ICE. As soon as the first byte of data enters the Trigger processor, analysis of the data begins. If, during the analysis, the Trigger processor discovers a candidate flash event, it will report the event (in the form of a candidate report) to the Overseer computer over the RS-232 serial link. The analysis continues until the entire image has been analyzed or the time allotted for analysis has been exceeded. In either case, the Trigger processor issues a sift-termination report to the Overseer computer: the sift-termination report indicates any errors that occurred in the analysis.

After the issuance of the sift-termination report, the Trigger processor automatically enters query mode, its other mode of operation. During sift mode, no communication is possible between the Overseer computer and Trigger processors, except for candidate and sift reports, since any communication can slow the processing of incoming data by the Trigger processor. The only exception to this is a handshaking byte sent from the Overseer computer to the Trigger processor after a candidate report. In query mode, the Trigger processor awaits commands or requests for data from the Overseer computer. During this period, the Overseer computer can check or change the operational parameters of the Trigger processor, as well as request transmission over the HSSL of arrays of data from any of the three data frames stored in Trigger processor memory. When appropriate (usually immediately before the next CCD image is to be read into the Trigger processor), the Overseer computer returns the Trigger
processor to sift mode with the issuance of an "end queries" command.

If for some reason the software in a Trigger processor halts, the Trigger processor is left in a non-communicative, non-productive state: the Trigger processor will have to be reset, the Trigger software re-downloaded and re-executed. A signal on the RS-232 serial link acts as an Overseer computer-activatable reset. The reset is generated on the parallel I/O board of the Overseer Computer (see section 7.1), and is sent to the halted Trigger processor over an RS-232 serial link, followed by the reloading of the Trigger software.

8.3.1. Sift Mode

In sift mode, the Trigger processor single-mindedly devotes itself to the analysis of incoming CCD data. As soon as the Trigger software recognizes the reception of the first byte of data from the CCD, it begins the process of data analysis; the DMA hardware in the HSSL stores the data automatically as it streams in. At the same time, the Trigger processor starts an internal countdown timer (the *sift timer*) which indicates the time at which sifting will be terminated, whether all the data has been analyzed or not. The sift timer, in conjunction with the Overseer computer's frame timer (section 7.3.1), assures that a known amount of time is spent in both sift and query modes, and that these times will not change from exposure to exposure.
At the onset of sifting, the Trigger processor allows data from several rows of the CCD image to stream into memory before beginning actual data analysis, to permit the sift algorithm to reference data in adjacent rows. The analysis of CCD data then begins. Because the HSSL transfers CCD data by DMA, the continuing storage of CCD image data proceeds without interrupting the analysis processes.

The analysis of a CCD image consists of sequentially passing each pixel of the image through the "sifter", which is described in the following section. The analysis of this image, known as the current frame, requires a threshold frame (see section 8.3.2.1 and Appendix E), as well as an image against which to compare the current frame to detect any brightening. This image, the recent frame, is the image taken immediately before the current frame.

If any portion of a CCD image is considered an event by the Trigger processor, the Trigger processor issues a candidate report to the Overseer computer over RS-232 serial link. This report consists of seven bytes and communicates the column and row numbers of the event, interpolated to a within a fraction of a pixel and reported in centipixels (.01 pixels), as well as the value of the triggering pixel in both the current and recent frames. The format of this report is included in Table 8.1. Upon receipt of the candidate report, the Overseer computer transmits a handshaking byte to the Trigger processor. The Trigger processor must receive this byte before it can issue any further reports. The handshaking byte is implemented to avoid having so many candidate events report to the
Overseer computer that the Overseer computer falls behind in receiving incoming bytes and, as a result, does not receive all the report data.

When the Trigger processor's internal sift timer expires, the Trigger processor issues a **sift-termination report** to the Overseer computer over the RS-232 link and enters query mode, regardless of whether the sift has finished. If the sift is still in progress, it is terminated immediately. Use of DMA by the HSSL assures that any data still coming in over the HSSL are stored correctly in the Overseer computer memory. A sift-termination report consists of three bytes which indicate the type of sift and the status of the sift when the sift timer expired. The format of the sift-termination report is included in Table 8.1.

The first byte of the sift-termination report includes a three-bit binary error code which indicates any errors in the sift. Each of these bits, the lowest three bits in the byte, indicates an error in the sift when it is transmitted in the high state. The information passed is 1) the sift was aborted before completion (bit 0), 2) too few bytes were received (bit 1) and/or 3) no sifting was done (bit 2). (See Appendix B for more detail). The possible errors are:

0: Sift completed, no errors. The Trigger processor received all the image bytes and successfully completed the sift procedure in the allotted time.

1: Sift incomplete. The sift process was interrupted by the sift timer. The last two bytes of the sift-termination report indicate the number of rows successfully sifted.

2: Not enough data. The number of bytes received by the Trigger processor is not the same as the number of bytes
expected. This error usually indicates a problem with the ICE or, more likely, the HSSL.

3: Sift incomplete and not enough data (see 1 and 2).

4: No sift. The Trigger processor had no image against which to compare the current image during the sift. Sifting was not attempted, but the image was placed in memory for use in future sifts.

5: Not possible.

6: No sift and not enough data (see 2 and 4).

7: Not possible.

8.3.2. Sifting

The real-time analysis, or sifting, of the CCD images is done with a series of sift algorithms which, together, represent all of the criteria for recognizing an event in the ETC. Each algorithm, or level of sifting, is a test of whether an image pixel meets a certain criterion. A pixel meeting the criterion of a given level is passed

<table>
<thead>
<tr>
<th>Format of Trigger Processor Reports to the Overseer Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Candidate Report</strong></td>
</tr>
<tr>
<td>Column number of event in centipixels</td>
</tr>
<tr>
<td>Row number of event in centipixels</td>
</tr>
<tr>
<td>Value of event pixel in recent frame</td>
</tr>
<tr>
<td>Value of event pixel in current frame</td>
</tr>
<tr>
<td>Value of threshold offset</td>
</tr>
<tr>
<td><strong>Sift-termination Report</strong></td>
</tr>
<tr>
<td>Three-bit error flag (see section 8.3.1)</td>
</tr>
<tr>
<td>Row number of sift termination</td>
</tr>
<tr>
<td>(0 if sift completed)</td>
</tr>
</tbody>
</table>

Table 8.1: Format of Trigger processor Reports to the Overseer computer
to the next level for further testing; a pixel rejected by any level is not analyzed further and the sifter proceeds to examine the next pixel. Any pixel passing all sift levels is considered by the Trigger processor to be an event candidate and becomes the subject of a candidate report to the Overseer computer.

Since an ETC image of $10^5$ bytes must be analyzed in about one second, an average of 5-10 microseconds of sifting time is allowed per pixel. Because this time is so short, the first level of sifting is a quick, simple algorithm which rejects most pixels in an ETC image. The small fraction of the image pixels which pass the first level of sifting are analyzed in increasing detail (at the cost of a higher analysis time per pixel) by the next levels of the sifter.

The following subsections describe each level of sifting in detail, proceeding in the order of levels encountered by a triggering pixel.

8.3.2.1. Level 1: Threshold Comparison

The first level of sifting is a check of whether the brightness of a pixel exceeds the expected sky brightness in a statistically significant way. This level is passed if the value of the current image pixel is greater than the value of the same pixel in the threshold frame, which is located in a specific area in Trigger processor memory. The value of each pixel of the threshold frame is the sum of the sky-plus-bias level in the image frame (in the absence of stars) and a constant integer offset, the threshold offset (specified by the
Overseer computer).

The threshold frame is constructed at the request of the Overseer computer by the Trigger processor before the beginning of an observation cycle from a normal ETC image through the process of median filtering. This method, described in detail in Appendix E, removes all high-spatial-frequency images (such as stars) from the CCD image, leaving an image of the sky without the stars. The value of the threshold offset reflects a statistical criterion programmed into the ETC software. The value of threshold offset is determined by the level of statistical significance: the Overseer computer calculates the value of threshold offset from the total noise per pixel of the CCD image. The value of threshold offset is added to the value of each pixel of the median filtered image to create the threshold frame.

Use of a threshold frame permits high speed processing of incoming CCD image data. Because the threshold offset is already added to the median filtered image, the Level 1 test consists of a single comparison of single-byte values. The Level 1 test requires a processing time of 4.8 microseconds per pixel.

The pixels which pass Level 1 have a value exceeding the starless sky value by a few ADU. These pixels are usually a part of a stellar image, a "hot" CCD column or a cosmic ray. The fraction of a full image's pixels passing Level 1 and thereby reaching Level 2 is dependent on the value of threshold offset and the number of bright stars in the field of the CCD: the fraction is of the order of ~0.01
8.3.2.2. Level 2: Brightening Test

The second level test checks whether a pixel is significantly brighter than the same pixel in the recent frame. The second level is passed by those pixels whose value in the current frame exceeds their value in the comparison frame by an amount greater than a constant known as brightened delta, which is specified by the Overseer computer. The value of brightened delta is calculated in the same way as the value of threshold offset (see above). The pixels that pass Level 2 are potentially very interesting to the ETC: they consist of cosmic rays, hot pixels and, hopefully, flash events. The fraction of incoming pixels passing Level 2 depends very much on the value of brightened delta and the total noise associated with the image: it will generally be $<0.005$.

8.3.2.3. Level 3: Centering Test

A flash event may cause several adjacent pixels to brighten enough to pass Level 2. Since the sifter should only issue one candidate report per event and report the center of the event, the Level 3 test checks the pixels neighboring the current pixel to determine whether the current pixel is the center of the event.

The third level of sifting is passed if the current pixelbrightened more than any of its immediate neighbors. The fraction of incoming pixels passing Level 3 is $0.5$. 
8.3.2.4. **Level 4: Anti-streak Algorithm**

The fourth level of sifting is a test of whether the current pixel is part of a streaked image. This tests for meteor trails and perhaps fast satellite or airplane trails. The Level 4 sift tests all pixels on the perimeter of a seven-by-seven pixel box centered on the pixel under consideration: if any of these pixels pass the Level 2 brightening test (section 8.3.2.2), the image is considered a streak and the pixel being analyzed is rejected. If any part of the box is off the edge of the CCD, the pixel is rejected.

Estimates of the rate of meteors which have trails short enough (in angular size) to falsely satisfy the anti-streak algorithm are given in Chapter 10.

8.3.2.5. **Level 5: Bright Star Test**

This level was devised when it was noticed that pixels surrounding bright stars were frequently the subjects of candidate reports. These brightenings were interpreted as being due to a shifting of the stellar image on the CCD, perhaps due to of local atmospheric effects or the effects of high shot noise near bright stars in the CCD. In order to recognize this effect, Level 5 checks whether the value of any of a pixel's immediate neighbors exceeds the value of the pixel by more than three times brightened delta (generally ~20 ADU).

8.3.3. **Sift Parameters**

The performance of the Trigger software can be controlled by the Overseer computer by changing one of several sift parameters during
query mode. These adjustable parameters include:

1) The variable **threshold offset** is the criterion used in the first level of sifting (see section 8.3.2.1).

2) The variable **brightened delta** is used in the second and fifth sifting levels (see sections 8.3.2.2 and 8.3.2.5).

3) The **sift time** is the upper limit on the amount of time allowed for sifting. The amount of sift time needed is dependent on the values of threshold offset and brightened delta, and should be sufficiently shorter than the exposure time to allow time for requests for data by the Overseer computer.

8.3.4. Query mode

Query mode is the free time between the termination of sifting and the next "end-queries" report. This time is available to the Overseer computer for 1) changing Trigger processor parameters, 2) making requests for data, or 3) requesting calculation of a threshold frame. A list of possible requests was given in section 7.2.2 and Figure 7.1.
The Instrument Control Electronics

Introduction

The Instrument Control Electronics (ICE) are a set of custom-built analog and digital electronics which control the instrumentation of the ETC. Through the ICE, the Overseer computer or a human user can interface with and control the ETC instrument (e.g., read out the CCDs, measure and adjust the CCD temperature, slew the telescope drive and read the hour angle of the mount). Such software manipulation of the instrument hardware is essential to the automated ETC.

9.1. Instrument Control Electronics Hardware

The ICE are controlled directly by a single-board microcontroller, known as the "COSMAC" board, which is based on the RCA 1802 "COSMAC" microprocessor. Overseer computer communication with the ICE occurs over an RS-232 serial port on the COSMAC board.

The ICE are divided into two groups: the "digital" and "analog" electronics. "Digital" electronics consist of all those boards sharing a common bus with the COSMAC microprocessor board. The COSMAC bus is a well-defined standard, allowing for 16-bit wide memory addresses, 8-bit data and eight I/O lines for communications with other boards sharing the bus. All boards sharing this bus are
directly controlled by the COSMAC.

The "analog" electronics are primarily concerned with CCD clock generation and CCD signal processing and transmission. The CCD clock generation ("driver") and CCD signal processing ("analog") boards are very noise-sensitive, since any noise present on these boards reduces the quality and consistency of the CCD images. For this reason, the analog electronics are physically separated and RF-shielded from the digital electronics to reduce the extent of infiltration of digital noise. The output and input signals from the Analog Electronics to the CCDs are wired directly from the analog backplane to the cable connectors leading to the cameras.

The sections that follow give a general description of the function and layout of each board in the Instrument Control Electronics. More attention will be paid to the Analog Electronics, since they are more specific to the operation of low-noise CCD cameras and generally the more intricate boards of the ICE. A schematic diagram of the boards and the data and signal flow in the ICE is given in Figure 9.1.

9.1.1. Digital Electronics

The ICE digital electronics consist of several boards which are necessary for primary computer and CCD control (the COSMAC, memory, sequencer and DAC boards). In addition, the digital electronics include boards which control details of the ETC instrumentation (CCD temperature control, shaft encoder readout, clutch control, stepper
Figure 9.1: Layout of the Instrument Control Electronics into Analog and Digital electronics
motor control and dome control boards).

9.1.1.1. COSMAC Board

As mentioned above, the COSMAC controller board is based on the RCA 1802 "COSMAC" microprocessor. The COSMAC is an eight-bit, bit-level controller; higher-level control is made possible by a FORTH-like language developed for the COSMAC. This code is stored in a set of four "boot ROMs", so the COSMAC is in a user-accessible state immediately after power-up.

The COSMAC board contains a UART (Universal Asynchronous Receiver/Transmitter) for serial communication at data rates from 300 to 9600 baud. Higher level instruction code can be created for the COSMAC and loaded into the COSMAC from another processor (e.g. the Overseer computer) over the RS-232 serial link. The collection of higher-level code for the ETC's Instrument Control Electronics, known as the operating system, contains intermediate level building blocks, the Sequel compiler (see section 9.1.1.3) and upper-level instrument control code. A copy of the basic operating system and command definitions can be found in Appendix C.

9.1.1.2. COSMAC Memory Board

The COSMAC memory board provides 16K of RAM to the COSMAC board through 8 2K RAM chips. The COSMAC memory board also houses the four 2K "boot ROMs" which store the COSMAC start-up instructions. The memory available to the COSMAC is allocated in part to certain peripheral boards of the Digital Electronics: the major portions of the
COSMAC memory are reserved for the boot ROMs and the Sequencer compiler (see section 9.1.1.3). In addition, many small control functions of the Digital Electronics are accessed through memory locations, rather than through the available I/O lines. A map of the functions of memory locations of the ETC four-camera test unit is shown in Table 9.1 (see the appropriate subsections for details of the control functions).

### 9.1.1.3. Programmable Sequencer

The Sequencer is responsible for the creation of the timing of the CCD clocks (see discussion of the CCD clocking in sections 9.1.2 and Figure 9.2). The sequencer, based on the Signetics 8x300 bipolar microcontroller, is run at 8 MHz, producing pulses 250 ns wide. The sequencer can produce up to 16 distinct streams of programmable timed pulses. The streams are generated by the sequential reading and execution (in 250ns steps) of 16-bit memory locations. The state of each of the 16 bits reflects the state of the appropriate stream in any given 250 ns time period: if a bit is high, the stream level is

<table>
<thead>
<tr>
<th>Memory location (hex)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 1FFF</td>
<td>Boot ROM</td>
</tr>
<tr>
<td>7300 - 73F7</td>
<td>DACs</td>
</tr>
<tr>
<td>E000 - EFFF</td>
<td>Sequencer compiler</td>
</tr>
<tr>
<td>FF00 - FFFF</td>
<td>Device control</td>
</tr>
</tbody>
</table>

Table 9.1: COSMAC Memory Map
The programming and sequential execution of the sequencer memory locations is done through a high-level language called Sequel, developed at MIT by John Doty, which runs in the COSMAC environment. The Sequel program as well as the stream execution memory are stored in fast-access RAM on the sequencer board.

In addition to the creation of the three clocks used to drive the CCDs, the sequencer is also responsible for timing operations in the analog processing of the CCD signal. The timing of the hold pulse in the analog processing chain is controlled by a sequencer stream (see Figure 9.2).

The signals from the Sequencer are fed to the Analog Electronics through a 26-pin flat ribbon cable to any of the driver boards. Since all the CCDs in the ETC test unit are timed identically, the sequencer signals are bussed along the Analog backplane to all the driver boards. Sequencer signals to the Analog boards are fed from the drivers along the backplane.

9.1.1.4. DAC Board

The DAC board produces the analog voltages used in clocking out the CCDs (see section 9.1.2 and Figure 9.2). It is filled with 24 Digital to Analog Converters (DACs), each of which will produce a 0 to 10 Volt analog DC output based on an eight-bit digital input. The 24 DACs provide voltages to two cameras (three DACs per each of three tri-level CCD clocks and two bias voltages per CCD). The DAC signals
are brought to each driver board on 26-pin flat ribbon cable connectors.

9.1.1.5. CCD Temperature control

The CCD is cooled to ~-90°C by a closed-cycle refrigerator. The temperature of the CCD can be held constant by passing a controlled amount of current through a power resistor mounted on the CCD cold sink. The temperature of the CCD is read via a thin film detector (a temperature-dependent resistor). The temperature of the CCD is controlled by an analog feedback loop which regulates the current through the power resistor to within ±1°C.

9.1.1.6. Stepper motor driver board

The slewing of the ETC telescope mount is done with a 0.88 amp stepper motor mounted on the telescope drive. The stepper motor driver board is designed to provide power to any winding or any combination of windings of a stepper motor given the appropriate low-level software command. Thus, the motor can be stepped by a series of commands powering the windings of the stepper in the correct sequence. The stepper motor in the ETC is stepped as quickly as the software will allow (~500 steps/second), which is sufficient to slew the mount at a rate in excess of 40 degrees per minute.

9.1.1.7. Synchro readout board

The synchro readout board services the synchro shaft-angle encoder on the ETC tracking mount. It utilizes a CMOS synchro-to-
digital converter, which converts the three 400 Hz signals from the synchro shaft-angle encoder on the polar axis of the telescope mount to a single 14-bit absolute shaft angle. This angle can be calibrated to give the absolute hour angle of the telescope mount. A 14-bit S/D gives an angular resolution of 1.3 arc-minutes, which is comparable to the angular size of an ETC pixel.

9.1.1.8. Clutch control board

The clutch control board controls the motor clutches in the telescope drive. The drive has a slewing motor, a tracking motor and a clutch associated with each. In order to prevent both clutches from being engaged at the same time, they are controlled by a single double-pull, single-throw relay on the clutch control board.

9.1.1.9. Dome Control Board

The dome at ETC Site 1 is equipped with an incremental encoder designed and built by Gerard Luppino of MIT. The output of the incremental encoder is monitored by circuitry on the dome control board. The dome encoding circuitry allows the determination of the location of the dome to 10-bit precision (~21 arc-minutes). The dome control card also includes circuitry which controls the two relays controlling the motion and direction of motion of the dome.

9.1.2. Analog Electronics

The primary function of the analog electronics is the acquisition, processing and transmission of image data from the CCDs. The
clock signals used in reading out the CCD are created on the **driver** board using the timing signals from the **sequencer** and the voltages from the **DACs**. Each clock requires three DAC signals and two sequencer streams. When the CCDs are read out, the CCD output signal is amplified by a preamplifier inside the camera housing and sent to the **analog processing board**. Here the signal is sent through a delay line and several stages of amplification and digitized to 12 bits. These 12 parallel bits are sent to the High Speed Serial Link (HSSL) where they are converted to eight serial bits and transmitted to the appropriate Trigger Processor. A schematic layout of the analog processing circuitry can be found in Figure 9.2a.

### 9.1.2.1. Driver Board

The driver board creates CCD clocking signals by combining voltage levels from the DAC board according to timing signals from the sequencer. The CCD clocking signal is created by combining three analog voltage levels. One voltage level is always present in the clocking signals; the other two voltage levels are added to the clocking signal depending on the states of the two sequencer streams. The combination of the sequencer streams' timing and the three analog voltages allows for the production of the tri-level clocks used in reading out the CCDs. (See Figure 9.2c for a simplified schematic illustration).

Each driver board includes drivers for three separate CCD clocks and a bias voltage for the on-chip FET circuit. DAC and sequencer signals travel to the board over two flat ribbon cables. The CCD
Figure 9.2a: The CCD output analog processing chain. The output of the CCD is preamplified inside the camera housing and sent to the analog processing board. There the signal is separated from the background (via correlated double sampling (see below)), passed through several stages of amplification (one stage being a programmable amplifier (PRAM) controlled by the COSMAC), and digitized to 12 bits. The 12 bits are mapped to 8 on the HSSL board and converted to a serial stream, which is transmitted over coaxial cable at 250 kbytes/second. (see section 9.1.2 for details)

Figure 9.2b: Correlated Double Sampling. The delay line converts the time dimension of the CCD output signal to a spatial dimension along the delay line, allowing the signal and reset level to be input simultaneously to the balanced-difference amplifier.

Figure 9.2c: Simplified sketch diagramming the creation of tri-level clocking signals for the CCD. The voltages $V_a$, $V_b$ and offset are supplied by the DAC board. The timing signals $A$ (a and b) are two bits in a sequencer stream (see section 9.1.2.1 for details). The voltages $V(a)$ and $V(b)$ are added to the always-present $V(offset)$ at the input of the high-speed inverting amplifier depending on the states of $A(a)$ and $A(b)$. 
clocking signals are taken directly from the analog backplane to the cable connectors leading to the CCD cameras.

9.1.2.2. Analog Board

When a CCD is read out, the output signal is passed through a preamplifier in the CCD camera housing. The preamplified CCD signal is then fed directly to the analog processing board. The CCD image information in the signal is separated from the signal background by an analog delay line and a balanced difference amplifier via correlated double sampling (see Figure 9.2b). The image signal is then passed through a software-controlled programmable variable-gain amplifier (PRAM), the gain of which is controlled by a sequencer stream, which is controlled by the COSMAC. The signal out of the PRAM is re-amplified, digitized to 12 bits and passed to the High Speed Serial Link.

9.1.2.3. High Speed Serial Link (HSSL) board

The HSSL board accepts the 12-bit digital output from the Analog processor and converts it to an eight-bit serial stream readable by the HSSL receiver in the Trigger processor (see section 8.1.2). The twelve-to-eight-bit conversion is done through UV-erasable Programmable Read-Only Memory chip (EPROM) on the HSSL board, which maps its 12-bit input onto its 8-bit output based on a lookup table stored in its memory.

The high-speed serial link is based on the National Semiconductor DP8342/3 serial transmitter/receiver pair. The transmitter
accepts 8-bit parallel data and transmits in a serial form readable by the receiver in the Trigger Processor. The receiver then translates the serial data back into parallel data. The transmitter/receiver pair in the ETC are run at 20 MHz, resulting in a 2 Mbit/second data transfer rate.

9.2. ICE Software

The Instrument Control Electronics are entirely controlled by low-level commands to the COSMAC microprocessor. A bit-level operating language was developed for the COSMAC microprocessor by Robert Goeke of MIT. This FORTH-like language was built upon to form a mid-level language, whose commands form the basis of the ICE software operating system.

The ICE control software consists of about four dozen mid-level commands to other boards on the COSMAC bus. Most commands involve the movement of data to and from locations in the ICE memory. The rest are commands to the sequencer, either to read out the CCDs or to change the CCD signal creation and analysis environment.

The ICE operating system and working software can be found in Appendix C. The bulk of the code defines the COSMAC software environment and the Sequel compiler: ETC control code is located near the back of the file.
CHAPTER 10

Expected Results

Introduction

Historically, it has generally been true that important scientific discoveries are made after the introduction of new instrumentation that opens the door to the investigation of an as-yet-unexplored physical regime. As a detector of optical flashes with risetimes of the order of one second, the ETC is investigating a relatively untouched region of parameter space in astronomy. The ETC has the potential to discover an entirely new class of astrophysical phenomena, characterized by brightness changes on the timescale of one second.

While the potential new discoveries of the ETC cannot be predicted, the detection of fast optical transients from astrophysical objects can be foreseen. The rate and brightness of optical flashes from GRBs and flare stars can be estimated from previous theories for expected transient optical radiation from the sources. By the same token, the rate and brightness of optical flashes from terrestrial sources, such as meteors and satellites, can be estimated.

As a bank of solid-state, electronic detectors, the ETC is also susceptible to local events which mimic optical flashes, such as cosmic rays and shot noise in the CCD. This chapter includes a
117
discussion of the sources of events as seen by the ETC and estimates their expected rates. The results are graphed in Figures 10.3-10.7.

10.1. Rate of Optical Flashes from Gamma-Ray Burst Sources

The expected rate of optical events from GRB sources is very uncertain at present, due to the lack of solid data on optical flashes from GRBs. A rough estimate can be made by convolving 1) the logN-logS curve for detected GRBs (Jennings, 1982), 2) the ratio of $F_\gamma/F_B = 750$ (for short (~5 second) optical bursts; Rappaport and Joss, 1985), 3) an estimate of the B-V for such a burst (Rappaport and Joss, 1985) and 4) an assumed fraction of GRBs which emit optical light.

The convolution of the log N(>S)-log S curve for GRBs (see section 2.1.4) and the ratio of gamma-ray to optical fluence is straightforward, as it is a simple recalibration of the axes of the log N(>S)-log S curve, based on the duration of the optical flash. Since the burst duration is rather loosely constrained by observation ($\tau_{\text{burst}}<500$ seconds), the optical event rate was calculated for burst times of 1, 5 and 30 seconds. The calculated rates of optical transient events from GRBs, assuming B-V = 0.0, are plotted in Figure 10.1.

An estimate of the B-V value of an optical burst from a GRB source requires an assumption of the mechanism for the production of optical light from GRBs. If the optical radiation is gamma-radiation reprocessed on the surface of a companion star, an assumption of the
burst temperature of the companion is sufficient to give a value for B-V. Rappaport and Joss' (1985) value for a typical burst temperature, $T_b = 8500$ degrees K, yields the value for B-V of -0.15; the resulting curve is not significantly different from the B-V = 0.0 curve which is plotted in Figure 10.1.

Finally, since no observer can make a statement at present about the fraction of GRBs which emit optical light, the plots in Figure 10.1 assume that all GRBs produce optical light.

10.2. Rate of Optical Flashes from Comet Impacts onto White Dwarfs

Tremaine and Zyktow (1985; see Chapter 3) have proposed that optical burst events could be created by the impact of a comet onto a white dwarf. The comets, which are bound in a cloud about the white dwarf, impact the white dwarf at an average rate of $1.2 \times 10^{-3}$ yr$^{-1}$. If, as proposed in a scenario by Tremaine and Zytkow, the impact of a comet (mass = $10^{16}$ g) heats the surface of a typical ($M = 0.5M_{\text{Solar}}$) white dwarf to 30000°K, resulting in a burst duration of ~10s, the absolute magnitude of the burst would be $M_V = 9.9$ (Lang (1980), pp. 564-5). Given that the space density of white dwarfs is 0.03-0.1 pc$^{-3}$ (say 0.05 pc$^{-3}$), the rate of optical flashes with $m_V < 9.9$ is $2.3 \times 10^{-6}$ sr$^{-1}$ hr$^{-1}$. Assuming the white dwarfs are isotropically distributed, the log $N(>S)$ - log $S$ relation for white dwarfs will follow an $S^{-3/2}$ dependence. The rate $R(m<m_0)$ of optical flashes brighter than magnitude $m_0$ from white dwarfs is included in the plots in Figures 10.7, 10.8 and 10.9.
Figure 10.1: The expected rate of optical transient events associated with GRBs, based on a ratio of gamma-ray to optical fluence of 750 and optical burst times of 1, 5 and 30s.
The rate of optical flashes expected from white dwarfs increases if one assumes that the missing mass in the Galaxy ($\sim 0.1 \, M_{\odot} \, \text{pc}^{-3}$) consists solely of $0.5M_{\odot}$ white dwarfs. In this case, the space density of white dwarfs is $0.2 \, \text{pc}^{-3}$, and the rate of optical flashes with $m_V < 9.9$ is $\sim 10^{-5}$.

10.3. Rate of Optical Flashes from Flare Stars

Flare stars are a class of red (primarily M) stars which exhibit frequent brightenings with risetimes of from a few to hundreds of seconds. These flares are brightest (in magnitudes) in the UV (where the star is not very bright), decreasing in brightness with increasing wavelength, until they are virtually undetectable in the red. The frequency of the flares has been empirically determined to be an exponential function of the quiescent absolute magnitude of the star, with fainter flare stars flaring more frequently (Gurzadyan (1980)). Observed flare stars have absolute magnitudes ranging from 8 to 15 and are generally found within 50 pc of the Sun. (A complete review of flare star observations and theory can be found in Gurzadyan (1980)).

Observed flare stars have the following properties, based on information given in Gurzadyan (1980):

1) The rate of flares from a given star is a function of its absolute magnitude $M_V$:
   $$ \log f_B = 1.78 + 0.148 M_V $$
   flares/hours.

2) The probability of a flare star flaring by more than an amount $\Delta V$ can be roughly modelled as
   $$ \ln P(\Delta V) = -\alpha_p \Delta V, $$
   where $1.4 < \alpha_p < 2.2$.

3) The rate of flares of a given $\Delta m$ in $V$-band is roughly
one-quarter of the rate in B-band.

4) Most flare stars are of spectral type M.

5) About 40% of all M stars within 6.5 pc of the Sun have been observed to flare; about 3% of all M stars within 20 pc have been observed to flare. This is evidently a selection effect.

Given these observational constraints, an estimate of the rate of flares versus peak flare magnitude can be made with a simple Monte Carlo analysis. In this procedure, the volume of space inside 20 pc is randomly filled with 1000 M stars, according to the density distribution of M stars in Allen (1976). Each star is assigned an absolute magnitude $M$, distance $r$, apparent magnitude $m$ (based on $M, r$) and a flare probability (per hour) based on the absolute magnitude $M$. In the Monte Carlo simulation, all stars are checked at regular time intervals (small compared to the smallest flare period). It is established whether each star flared in the preceding interval by comparing a random number to the individual flare probability. If so, the change in visual magnitude during the flare is determined from the rough probability distribution quoted above. The magnitude, distance, flare magnitude and time of each flare are recorded: after the Monte Carlo simulation has ended, these data are used to determine the rate of flares (in $\text{sr}^{-1}\text{hr}^{-1}$) above a threshold magnitude $\mu$.

Such a Monte Carlo simulation of flare stars was executed, based on the observed flare star characteristics listed above. The value of $\alpha_p$ was assumed to be 1.8, and the fraction of M stars that flare was assumed to be 0.5. The simulation was executed in shells of volume about the Sun in order to observe the dependence of flare rate
on distance from the Sun. The shells used were 0-20 pc, 20-50 pc, 50-100 pc and 100-200 pc, with each shell being observed for 100,000 flares. The flare rates from the various shells were summed to give flare star rates for populations of stars within 20, 50, 100 and 200 pc of the Sun. The results of these simulations are included in Figure 10.2. The straight line in Figure 10.2 represents an extrapolation of these curves to an infinite spherical distribution about the Sun. It is interesting to note how quickly the various curves converge to the infinite-distribution \( S^{-3/2} \) rate: for the sensitivity of the ETC \( m_v = 11-12 \), the majority of flare star events will come from stars within 100 pc of the Sun.

In order to account for instrument sensitivity, the Monte Carlo simulation was modified to calculate the rate of flare events exceeding \( 400 e^- \) (a sensitivity of \( m^-10.8 \)). This curve is displayed in Figure 10.3.

The uncertainty in the flare star event rate reflects the uncertainty in the observed flare star characteristics. The fact that the value of \( \alpha_p \) is seen to lie between 1.4 and 2.2 affects the event rate by a factor of \( e^{0.4} = 1.2 \). An additional element of uncertainty is introduced with the assumption that 50% of all M stars exhibit flares: the fraction is seen to be at least 0.4, yet could be as high as 1.0, so the flare rate calculated by the Monte Carlo simulation could be low by a factor of 2 because of this assumption.

A factor which has not been included in these calculations is the fact that the ETC is sensitive to changes in brightness in of the
Figure 10.2: The results of the Monte Carlo simulation of flare star event rates discussed in section 10.3. The curves are the flare event rates from populations of flare stars located within 20 pc, 50 pc, 100 pc and 200 pc of the Sun. The straight line is the extrapolation of these curves to an infinite spherical distribution of flare stars about the Sun ($\log N \sim (-3/2)\log S$).
Figure 10.3: The effect of considering a detection threshold on the rate of detection from the 200 pc flare star population of Figure 10.2. The lower curve is of flare stars events by the ETC. The upper curve is the rate of events the convolution of the upper curve with a detection threshold of 400 electrons ($m = 10.4$).
order of 1-4s. If the risetime of an optical burst is significantly longer than this, the sensitivity to the burst will drop, since only a fraction of the increase in brightness will be observed in a single exposure. If the rise of brightness to peak is roughly linear, the effective magnitude of the burst (as seen by the ETC) is increased by 

$$-2.5 \log(t_{\text{exp}}/t_{\text{rise}}),$$ 

where $t_{\text{exp}}$ and $t_{\text{rise}}$ are the exposure and rise-times, respectively.

This effect is potentially greater for flare stars, which have typical risetimes of 10-60 seconds, than for optical flashes from GRBs, which may have risetimes of the order of seconds. If we take a typical flare star risetime to be 30 seconds, the decrease in detection sensitivity of a one-second exposure is 3.7 magnitudes. The effects of the typically long flare star brightening times is taken into account in Figure 10.4, which assumes a flare star risetime of 30 seconds. (It is important to note that 30 seconds is a rough median risetime and that actual event risetimes vary over almost two orders of magnitude (Moffett, 1974)).

This effect is, in principle, the same for optical bursts from GRB sources; yet, because of the absence of reliable data on GRB-related optical transient risetimes, the magnitude of the effect is difficult to estimate. This effect is taken into account in Figure 10.7, 10.8 and 10.9, assuming an optical burst time of five seconds.
Figure 10.4: The effect of a long flare star risetime on the rate of detected events in the ETC. The lines are the event rates from an infinite distribution of flare stars (Figure 10.2), adjusted for the amount of brightening detected in one exposure time. The flare star risetime is assumed to be 30 seconds: the three curves plotted are detection rates in 1s, 4s and 10s exposures, assuming a linear rise to peak.
10.4. Terrestrial Sources of Optical Flashes

The primary known sources of terrestrial optical flashes are satellites and meteors. These flashes would be a particular problem for a single-site ETC, which would not be able to use parallax to discern the terrestrial origin of optical flashes from these sources. The streak-rejection algorithm in the Trigger software (section 8.3.2.4) is capable of rejecting fast satellites and most meteors. However, any source of optical transient radiation with an angular speed small enough to pass the streak-rejection test, such as slower satellites or head-on meteors, would be considered real events by a single-site ETC.

This section describes the rough calculation of event rates due to these sources.

10.4.1. Meteors

A meteoroid is a particle or group of particles, ranging in size from 0.001 cm (micrometeoroids) up to tens of kilometers (asteroids). A meteor is produced when such a particle enters the Earth's atmosphere and is partially or totally vaporized, creating a luminous trail sometimes visible with the naked eye. Upon entry into the Earth's atmosphere, the meteoroid is heated by contact with air molecules and begins to melt and evaporate at a rate dependent on the meteoroid temperature and mass. The rate of emission of meteoroid vapors determines the luminosity of the meteor trail.
A meteor can create a false trigger in the ETC if the angular size of the portion of the trail bright enough to trigger the ETC is smaller than the streak criterion in the Trigger software (section 8.3.2.4). The angular length depends on the angle of incidence of the meteor into the atmosphere, the peak brightness of the meteor, the meteor velocity and the image exposure time. The rate of triggering meteors is a convolution of the probability of a meteor of a given brightness creating a false event in the ETC and the flux of meteors of that brightness entering the atmosphere.

10.4.1.1. Meteor light curve

The "typical" meteor light curve has been modelled by Oepik (1958). (Since meteors come in a variety of types, sizes and brightnesses, no one type can be considered "typical"; however, most of the meteors in the magnitude range of ~2 - 10 are considered to be created by the steady ablation of matter from a melted meteoroid, the rate of which was roughly modelled by Oepik). From his model, the meteor light curve has the form \( j \propto X(1-X)^2 \), where \( j \) is the brightness of the meteor and \( X = e^{-\Delta h/a} \), where \( \Delta h \) is the change in altitude of the meteor and \( a \) is the scale height of the atmosphere at the altitude of ablation. From this light curve, Oepik calculates the relative intensity of the meteor to be \( j/j_{\text{max}} = \frac{27}{4}X(1-X)^2 \).

Expressed in magnitudes, \( \Delta m = m - m_{\text{peak}} = 2.5\log(X(1-X)^2) + 0.83 \).

Observations of meteor light curves have substantiated this model (Bronshten, 1983).
From the above relation, given a detector sensitivity $\mu$, and the peak effective magnitude, $m_{\text{eff}}$, of the meteor as seen by the detector (taking image exposure time into account), one can calculate the length of the trail (in scale heights) that has a magnitude brighter than $\mu$; i.e., the length, $\Delta h/a$, of the trail with $\Delta m < m_{\text{eff}} - \mu$.

10.4.1.2. Meteor Geometry

The geometry of the incident meteor is shown in Figure 10.5. A meteor which enters the atmosphere with zenith angle $Z$ which creates a visible trail over a range of altitudes $\Delta h$ at a mean altitude $H$ will create a trail of angular size $\theta = \tan^{-1}(\Delta h \tan(Z)/H)$. The probability of a meteor entering the atmosphere with a zenith angle $Z$ is proportional to $\cos(Z)\sin(Z)$ (Oepik (1958), p. 67): the probability of the meteor entering the atmosphere with zenith angle less than $Z$ is $\sin^2(Z)$.

Given a detection threshold magnitude, $\mu$, some vertical length, $\Delta h$, of the trail of a meteor of peak effective magnitude $m_{\text{eff}}$ will be brighter than $\mu$. In this situation, there will be a range of zenith angle $Z$ which creates a visible trail length $\theta < \theta_0$. For a given threshold magnitude $\mu$, the rate of meteors with trails brighter than $\mu$ yet with an angular size $\theta$ less than a threshold $\theta_0$ is determined by convolving the rate of meteors of effective magnitude $m_{\text{eff}}$ with the probability associated with the zenith angle $Z$. 
Figure 10.5: The geometry of a typical meteor trail. A meteor entering the atmosphere at altitude \( H \) with a zenith angle \( Z \) will subtend an angle \( \theta = \arctan(L \sin Z / H) \), where \( L \) is the length of the meteor trail. If the angle is less than the anti-streak criterion for the ETC (presently 13 arc-minutes), the meteor will trigger a single-site ETC as an event. A two-site ETC will be able to reject the meteor by virtue of its parallax over the 1.4 km baseline between the sites.
10.4.1.3. The Effective Magnitude of a Meteor Trail

The effective peak magnitude of a meteor of actual peak magnitude $m_{\text{peak}}$ is given by $m_{\text{eff}} = m_{\text{peak}} - 2.5\log(t/\tau)$, where $t$ is the time the meteor spends in a single pixel and $\tau$ is the image exposure time. The time spent by a meteor in a single pixel depends on the velocity $V$ of the meteor: the value of $t$ is $V\sin(Z)/H\alpha$, where $\alpha$ is the angular size of a pixel. The value of meteor velocity $V$ is typically ~40 km/s; the value of $H$ ranges between 80 km and 100 km.

10.4.1.3.1. Meteor flux

The flux of meteors in the magnitude range -2.4 to +12 has been measured by Cook, et al. (1981). The result of the observations is consistent with those of Hawkins and Upton (1958) and is

$$\log f_B (\text{cm}^{-1}\text{s}^{-1}) = -17.89 + 0.534m_B$$

(This equation is consistent to within a factor of ~2 with values found in Allen (1976)). This value can be converted to meteors/frame/second by assuming a typical altitude of 90 km:

$$f_B (\text{frame}^{-1}\text{s}^{-1}) = 8.35 \times 10^{-6} 10^{0.534m_B}$$

Given the typical colors of a typical meteor ($B-V = -1.41$ (Allen, 1976)),

$$f_V (\text{frame}^{-1}\text{s}^{-1}) = 1.47 \times 10^{-6} 10^{0.534m_V}$$

For the ETC, one frame has a solid angle of .086 steradians, so the calibration of frame$^{-1}$s$^{-1}$ to sr$^{-1}$hr$^{-1}$ is a multiplicative factor
of 41900. The flux of triggering meteors in sr⁻¹hr⁻¹ are

\[ f_B (\text{sr}^{-1}\text{hr}^{-1}) = 0.350 \times 10^{0.534m_B} \]
\[ f_V (\text{sr}^{-1}\text{hr}^{-1}) = 0.062 \times 10^{0.534m_V} \]

**10.4.1.4. The Net Rate of Head-on Meteors**

The total flux of triggering meteors was calculated by considering the threshold magnitude, \( \mu \), and effective magnitude of a meteor, \( m_{eff} \), based on the zenith angle \( Z \) and its associated probability. The results for a typical meteor velocity and several exposure times are graphed in Figure 10.6.

It should be noted that a meteor can create a trigger not only if it comes into the atmosphere with a low zenith angle \( Z \), but also at larger values of \( Z \), when the section of the meteor trail brighter than the threshold magnitude is small enough to trigger the ETC. This effect is noticeable as an increase of 20-30% in the event rate at fainter threshold magnitudes (\( \mu > 10 \)).

**10.4.2. Satellites**

The impact of satellites on the ETC could be very large, yet the magnitude of their effect is difficult to calculate. A tumbling satellite can create a false optical flash in the ETC by momentarily reflecting sunlight and appearing to the ETC as an optical glint. The ETC would also consider a moving, non-glinting satellite to be an event if the length of the satellite trail in a single exposure is less than the ETC anti-streak criterion.
Figure 10.6: The calculated rate of false events due to head-on meteors (see section 10.4.1). The rates assume a meteor velocity of 40 km/sec and exposure times of 1s, 4s and 10s. The criterion for being detected as a head-on meteor by the ETC is an angular extent of less than 13 arc-minutes.
The problem of glinting or moving low-Earth-orbit satellites will presumably disappear once the ETC has spread to two sites at Kitt Peak. (The 1.5 km baseline between the two ETC sites on Kitt Peak allows the recognition of terrestrial sources of optical flashes at altitudes of up to 3000 km). A single-site ETC, however, would possibly have to contend with a major fraction of satellite-induced events without the benefit of parallax. Presently, a non-glinting, moving satellite will be detected as an optical flash by a single-site ETC if its angular motion in a single exposure is less than 13 arc-minutes. This means that faster satellites (those in lower orbits) are more likely to be rejected by the anti-streak algorithm.

One method of reducing the event rate due to moving satellites might be to increase the exposure time, so that low-Earth-orbit satellites would always be rejected. Another possibility is the a posteriori rejection of satellite events based on three or more collinear glints in a single frame. Nevertheless, the problem of glinting satellites remains a problem until the second ETC site is ready, since, to a single-site ETC, a short glint looks like a point flash of light.

10.4.3. Instrument-related Spurious Events

There is a finite probability of the detection of spurious events due to non-optical effects in the CCD detectors. One source of such false triggers is statistical fluctuations due to shot noise in the CCD: a thorough discussion of this effect is given in Chapter 11. A second source of non-optical false triggers is the interaction
of cosmic-rays with the CCDs in the ETC cameras. A cosmic ray passing through a CCD ionizes the silicon it encounters, thereby freeing typically ~400 electrons (for a 3μm path length in silicon). A cosmic-ray interaction can mimic an optical flash in a single CCD camera (400 electrons corresponds to a star of m = 10.j): properly placed cosmic rays in two CCD cameras can therefore create a false event in the ETC.

The calculation of the rate of false events due to coincident cosmic rays in two cameras is straightforward. The flux of cosmic rays on Kitt Peak is 5-10 per cm² per minute. The rate per 22.3μm pixel per hour is 1.49x10⁻⁵. The probability per CCD (of 10⁵ pixels) of a false trigger is 2.23x10⁻⁵/hour. At 0.086 steradians per CCD, the cosmic-ray spurious event rate for the plenary ETC will be 2.6x10⁻⁴/hr/sr, independent of sensitivity. This rate is included in Figures 10.7, 10.8 and 10.9.

10.5. The Rate of Events Due to All Considerations

The total rate of events due to celestial and terrestrial sources of optical flashes is presented in Figures 10.7, 10.8 and 10.9. The exposure times assumed in these figures are one, four and ten seconds. The duration of optical flashes from GRB sources is assumed to be five seconds in all three figures. Superimposed on each of these graphs is the sensitivity of the plenary ETC to optical flashes in one year of observations. Also included in each figure is a curve representing the sensitivity of the observations made by Schaefer, Vanderspek, Bradt and Ricker (1984), which had been the
most sensitive wide-field search for optical flashes before the ETC.
Caption to Figures 10.7, 10.8 and 10.9

Figures 10.7, 10.8 and 10.9 display the expected rate of optical transients from known terrestrial and celestial sources in one-, four- and ten-second exposures in the plenary ETC, as described in Chapters 4 and 10. Included are the rates of optical transients due to GRBs (as discussed in section 10.1), assuming both one-second and five-second burst durations, flare stars (section 10.2), assuming a thirty-second rise to peak, and comet impacts onto solitary white dwarfs (section 10.3). In addition, the rates of "false" optical transients due to head-on meteors (section 10.4.1) and cosmic rays (section 10.4.2) are included. The three-sigma Poisson sensitivity of the plenary ETC in one year of observations is indicated on each plot (see Chapter 4), as is the sensitivity of the Two-Schmidt survey of Schaefer, Vanderspek, Bradt and Ricker (1984).
Figure 10.7: The expected event rates in one-second exposures in the plenary ETC.
Figure 10.8: The expected event rate in four-second exposures in the plenary ETC.
Figure 10.9: Expected event rate in ten-second exposures in the plenary ETC.
CHAPTER 11

Results of Observations with the ETC

Introduction

The ETC test instrument was set up at Kitt Peak in October and November of 1984. During the following December, January, and early March, the Overseer software was tested and refined. In late March and mid-May, 1985, observations of portions of the night sky were made with the ETC test unit. The goals of the observations with the ETC test unit were 1) to test the concept and operation of the Trigger and Overseer software, 2) to explore the limits of operation of the ETC software package, 3) to test the durability and reliability of the test instrument itself, and, of greatest astronomical interest, 4) to carry out the most sensitive wide-field optical flash search to date. These goals were achieved during the observation period. The following sections describe the tests of the ETC hardware and software; in addition, the observations with the ETC test instrument and the results of the observations are discussed.

11.1. Observations with the ETC Test Unit

The "shakedown" observations with the ETC test unit were made in December, January, March and May, 1985. The quality of the observations software progressed from "very crude" in December to "working and useable" in March of 1985, to "quite satisfactory" in May of 1985. Observations made after 23 March 1985 are particularly useful
in the sense that the software was working well enough that events could be detected and all data would be stored. Before 23 March 1985, the quality of the data stored made its astronomical use very difficult.

11.1.1. Description of the ETC Test Unit Hardware

The test observations were made with just two of the four available cameras (cameras B and C of A, B, C and D) due to a malfunction in camera D's support hardware. Camera A, though useable, did not have a coincidence "mate", so that it could not be used to survey an independent part of the sky. Each camera was equipped with a 25 mm lens and a filter covering either the V-, R- or I-band. The filters were necessary because the chromatic aberration of the lenses made wide-band focus impractical (even with the filter, the point-spread-function of the lens was such that a stellar image was smeared over three or four pixels).

The hardware (including Overseer computer, Trigger processors and ICE) in the test unit were as described in chapters 6 through 9. The HSSL transmitter in the ICE was equipped with an EPROM which mapped the 12-bit output of the analog processing board to an 8-bit format according to a half linear, half logarithmic mapping known as the "linlog" mapping (see section 9.1.2.3). The linlog function maps the values 0-150 on the input to 0-150 on the output (linearly) and the values 151-4095 on the input to 151-255 on the output in logarithmic steps, such that each output step above 150 ADU corresponds to 0.0341 magnitudes. (A plot of this mapping can be found in Figure
11.1.1. ETC Camera Sensitivity and Gain

The sensitivity of the ETC test unit to detected events can be calculated from equation 4.2. However, in order to have a good understanding of the system sensitivity, the gain (in ADU/e⁻) and readout noise (in e⁻) of each CCD camera must be well determined. Therefore, the gain and readout noise of each CCD camera was measured in three different ways: through a light-transfer curve (LTC), from the brightness of stars of known magnitude in the CCD, and from the dark-sky brightness.

The light-transfer curve (LTC) method requires the measurement of the total noise in the CCD at various light levels (the noise is the standard deviation of the values of the difference between two consecutive frames at the same light level). Since the total noise is given by \( \sigma_{\text{total}}^2 = \frac{\text{Signal (ADU)}}{g} + \sigma_{\text{readout}}^2 \), several data points allow for an estimate of the system gain \( g \) and readout noise \( \sigma_{\text{readout}} \).
Figure 11.1: The mapping of the 12-bit output of the analog processing board to the 8-bit image data used in the ETC. The mapping is stored on a UV-eraseable programmable read-only memory chip (EPROM). The 12-bit input to the EPROM (in units of linear ADUs) is mapped onto its 8-bit output (in linlog units) with the linlog mapping. Input values between 0 and 150 are mapped linearly onto output values 0 to 150. Input values between 151 and 255 are mapped logarithmically onto 151 through 255, so that each output step corresponds to 0.0341 magnitudes.
The second method involves calculating the relation of the brightness of stellar images (in ADU) to their visual magnitudes. This was done with SAO standard star images on a full CCD image recorded at Kitt Peak for both cameras B and C, and both analyses yielded mappings of the form \( \log(\text{ADU}) = \text{constant} - m_v \times (0.4 \pm 0.02) \), as was to be expected. Given the transmissivities of the atmosphere and the CCD camera optical system, and the efficiency of the CCD, the system gain can be calculated from this mapping.

The third method requires measuring the brightness of a moonless sky, also on a stored image. From a comparison of the measured dark sky brightness (in ADU/pixel/second) with the actual sky brightness (in photons/pixel/seconds) the gain can be calculated, given the transmissivity of the optical system and the CCD efficiency.

These three methods were used in the calculation of the gain and readout noise of each CCD camera. The results are summarized in Table 11.1. From the analysis, the values of 62 e^-/ADU and 51 e^-/ADU were adopted for cameras B and C, respectively. The readout noise was determined from the LTC to be 70 e^- in camera B and 52 e^- in camera C.

The values of the various parameters of the ETC optical system implicitly used in the calculation of the sensitivity of the ETC test unit (equation 4.2) are listed in Table 11.2. The values of I-band lens throughput and CCD efficiency are estimates, due to incomplete data in that wavelength band. The values of atmospheric characteristics (transmission and sky brightness) were taken from Allen (1983):
<table>
<thead>
<tr>
<th>Method</th>
<th>Camera B gain</th>
<th>Camera C gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$62 \pm 5 \text{ e}^-/\text{ADU}$ ($\sigma_{\text{readout}} = 1.15 \text{ ADU}$)</td>
<td>$46 \pm 7 \text{ e}^-/\text{ADU}$ ($\sigma_{\text{readout}} = 1.01 \text{ ADU}$)</td>
</tr>
<tr>
<td>Stellar brightnesses</td>
<td>$67 \pm 15 \text{ e}^-/\text{ADU}$</td>
<td>$48 \pm 18 \text{ e}^-/\text{ADU}$</td>
</tr>
<tr>
<td>Dark sky brightness</td>
<td>$63 \pm 6 \text{ e}^-/\text{ADU}$</td>
<td>$58 \pm 6 \text{ e}^-/\text{ADU}$</td>
</tr>
</tbody>
</table>

Table 11.1: Results of Gain Calibration of the ETC Cameras

these values are for sea level and differ somewhat from the values at Kitt Peak. The window transmissivity was calculated from the Fresnel equations, which give the reflection losses at each surface of the window. The values for the sensitivity to a detected event in each of the cameras at a signal-to-noise level of 10 are also listed in Table 11.2.

11.2. Testing and Operation of the ETC Software

During testing of the ETC test unit, the Overseer software was run in "user" mode, which allows for automatic observations and data storage after human-aided setup. User mode allows the observer to change observational parameters in order to test the performance of the instrument. The following sections describe "user" mode in detail: the methods used in the testing of the limits of the
Optical Characteristics of the ETC Test Unit

<table>
<thead>
<tr>
<th></th>
<th>V-band</th>
<th>R-band</th>
<th>I-band</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focal length</td>
<td>25 mm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f-number</td>
<td>0.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area of lens</td>
<td>6.8 cm²</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pixel angular size</td>
<td>3.09 arc-minutes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera C gain</td>
<td>51 e⁻/adu</td>
<td>52 e⁻</td>
<td></td>
</tr>
<tr>
<td>Camera C readout noise</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera B gain</td>
<td>62 e⁻/adu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera B readout noise</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sky rate (e⁻/pixel/second)</td>
<td>138</td>
<td>602</td>
<td>1046</td>
</tr>
<tr>
<td>Filter bandpass</td>
<td>1160 A</td>
<td>1200 A</td>
<td>1600 A</td>
</tr>
<tr>
<td>Atmospheric transmission</td>
<td>.82</td>
<td>.92</td>
<td>.95</td>
</tr>
<tr>
<td>Filter transmission</td>
<td>.71</td>
<td>.80</td>
<td>.77</td>
</tr>
<tr>
<td>Lens transmission</td>
<td>.85</td>
<td>.80</td>
<td>.70</td>
</tr>
<tr>
<td>Window transmission</td>
<td>.91</td>
<td>.91</td>
<td>.91</td>
</tr>
<tr>
<td>CCD efficiency</td>
<td>.45</td>
<td>.60</td>
<td>.38</td>
</tr>
<tr>
<td>Image splitting</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
</tr>
<tr>
<td>One-second event sensitivity</td>
<td>7.5</td>
<td>8.4</td>
<td>8.0</td>
</tr>
<tr>
<td>Four-second event sensitivity</td>
<td>9.0</td>
<td>9.7</td>
<td>9.3</td>
</tr>
<tr>
<td>Ten-second event sensitivity</td>
<td>9.9</td>
<td>10.5</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Table 11.2: ETC Prototype Optical Characteristics

Overseer and Trigger software are described in section 11.2.2.

11.2.1. User Mode

The "user" mode of ETC observations was developed as a preliminary to full instrument automation. User mode allows the observer to specify every aspect of the ETC observations, in order to test the quality and speed of the Overseer and Trigger software, whereupon the
ETC observes the sky without further interaction by the observer. An example of the observer's interactions with the ETC software's user mode is shown in Figure 11.2: the details of user mode are given below.

Upon entering user mode, the software asks the observer to move the telescope drive (manually or through the slewing motor) to the right ascension appropriate to the observations. Once the right ascension is correct, the software asks the observer to input the exposure time, sift time and the statistical criteria used in the calculation of the thresholds used in the Trigger software (see section 8.3.2). The Overseer computer then measures the sky+bias level of each CCD from images taken by each camera: the bias level is adjusted (if necessary) until the sky+bias level is between 25 and 60.

Once the sky+bias level is correct, the observer has the opportunity to save full images from all cameras onto disk files. The observer is then asked to rotate the dome slit until the cameras view the slit center. The relationship between the dome azimuth and the dome encoder value can then be calibrated. Next, the Overseer computer then calculates the values of threshold offset and brightened delta (see Chapters 7 and 8) from the observer-specified statistical criteria and a measurement of the total noise per pixel in each camera.

At this point, the observer participates in precisely calibrating "pixel address" to celestial coordinates for each CCD camera in
Preparation of the observer-computer interaction during the setup phase of "user" mode, described in section 11.2.1.

- Setting the necessary trigger parameters
- Attaching the observer-computer interaction during the setup phase of "user" mode, described in section 11.2.1.

- Reading the entire page to SC-MGR.
- Reading the entire page to SC-MGR.
- Reading the entire page to SC-MGR.
- Reading the entire page to SC-MGR.
- This line is 8:40 UT.
- Checking for edge calibration...
- Reading a dirty frame from last trigger memory
- B complete
- C complete
- D complete
- Reading the entire page to SC-MGR.
- The time is 15:48 UT.
turn. The Overseer computer roughly calculates the coordinates of
the center of a camera field from its knowledge of the date, time and
location of the site, the hour angle of the sidereal drive (through
the absolute synchro shaft encoder mounted on the drive) and the
declination of the camera (specified in the software). The Overseer
computer then calculates the expected locations of 16 SAO standard
stars in the field based on these rough coordinates: these locations
are circled over an image of the sky taken by that CCD camera and
displayed on the color graphics monitor. The observer is requested
to tell the Overseer computer how far the circles are from the actual
locations of the SAO standard stars. This is an iterative process,
which ends when the observer reports that the SAO stars are all cir-
cled. The Overseer computer then calculates the precise position of
the stellar images on the CCD and calculates the precise astrometric
mapping of the CCD location to celestial coordinates (see section
7.1.4.2).

Once all the CCD fields are calibrated, the threshold frames are
calculated by the Trigger processors from actual image frames. The
Trigger processors are then initialized: at this point, the sequence
of observation cycles can begin. The observation cycles run com-
pletely automatically, so no observer need be present. An observa-
tion cycle consists of twenty minutes of observations, followed by
data storage (see section 7.4.2), a check of the sky+bias level, a
recalculation of the astrometric mapping and a recalculation of the
threshold frames. If the sky+bias level in a CCD has changed signi-
significantly, the bias level in the CCD is adjusted automatically. The recalculation of the astrometric parameters is necessary due to a slight (~30') misalignment of the polar axis of the telescope drive; however, the misalignment is small enough that the Overseer computer can recalibrate the mapping without the assistance of an observer. This periodic recalibration is also necessary because of the effects of atmospheric refraction, which vary with hour angle.

The sequence of observation cycles continues indefinitely until 1) sunrise, or 2) the hour angle of the mount exceed six hours. In all, the net efficiency of ETC observations, including startup and recalibration pauses, is 75%: i.e., three-quarters of the time the observation software is running in the Overseer computer, the ETC is taking and analyzing images of the night sky. An individual observation cycle is halted when data has been taken from six prospective candidate events: the data are then stored and the observation cycle is immediately restarted.

11.2.2. The Testing of the ETC Software

During testing of the ETC software, the values of threshold offset and brightened delta were varied to determine their effect on the performance of the software. No attempt was made to unduly minimize the rate of "false" (statistical) events: the values of threshold offset and brightened delta were set as low as possible to maximize the chance of detecting a real event. The lower limit on the values of these two variables (which were generally set equal to one another) was the point where the sift algorithm was unable to
complete the sifting of a typical frame in the time allotted, because of the large number of pixels passing the first level of sifting (see section 8.3.2.1). The sift time was then increased to allow further decrease in the values of the detection variables until the amount of time left to the Overseer computer for data transfer (data transfer time = exposure time - sift time) was too small and exposure time limits were exceeded when the Overseer computer attempted to collect data from the Trigger processors. It was determined that the present Overseer software requires 0.6 seconds per flash to acquire data from two Trigger processors. The bulk of actual observations was performed with ten-second integrations, with sift times of four to six seconds. The values of threshold offset and brightened delta were adjusted until the rate of single events from a given camera was between 10 and 20 per exposure. These observation settings assured that >90% of all exposures were "good" (an exposure was considered "good" if the sifting completed with no errors and the Overseer computer had sufficient time to collect all the image data).

11.3. Details of Observations with the Test Instrument

The observations with the ETC test instrument were conducted between 24 March and 1 April 1985 and again between 12 May and 28 May, 1985. The observations were made primarily in V-band, with some bright-time observations in R- and I-band. A list of the times and filters of observation can be found in Table 11.3.

Cameras B and C were used in coincidence with a total field overlap of 236 square degrees (after accounting for imaging area lost
Table 11.3: Dates and Times Observed with ETC Prototype

<table>
<thead>
<tr>
<th>UT date</th>
<th>Filter</th>
<th>Time observed (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 March 1985</td>
<td>V</td>
<td>2.67</td>
</tr>
<tr>
<td>25 March 1985</td>
<td>V</td>
<td>5.17</td>
</tr>
<tr>
<td>26 March 1985</td>
<td>V</td>
<td>2.31</td>
</tr>
<tr>
<td>27 March 1985</td>
<td>V</td>
<td>3.08</td>
</tr>
<tr>
<td>28 March 1985</td>
<td>V</td>
<td>1.18</td>
</tr>
<tr>
<td>30 March 1985</td>
<td>R</td>
<td>2.46</td>
</tr>
<tr>
<td>31 March 1985</td>
<td>I</td>
<td>3.49</td>
</tr>
<tr>
<td>1 April 1985</td>
<td>I</td>
<td>3.36</td>
</tr>
<tr>
<td>12 May 1985</td>
<td>R</td>
<td>1.44</td>
</tr>
<tr>
<td>13 May 1985</td>
<td>V</td>
<td>3.45</td>
</tr>
<tr>
<td>14 May 1985</td>
<td>V</td>
<td>4.75</td>
</tr>
<tr>
<td>15 May 1985</td>
<td>V</td>
<td>3.85</td>
</tr>
<tr>
<td>18 May 1985</td>
<td>V</td>
<td>3.04</td>
</tr>
<tr>
<td>19 May 1985</td>
<td>V</td>
<td>4.26</td>
</tr>
<tr>
<td>20 May 1985</td>
<td>V</td>
<td>1.35</td>
</tr>
<tr>
<td>21 May 1985</td>
<td>V</td>
<td>1.22</td>
</tr>
<tr>
<td>24 May 1985</td>
<td>V</td>
<td>1.77</td>
</tr>
<tr>
<td>25 May 1985</td>
<td>V</td>
<td>3.45</td>
</tr>
<tr>
<td>26 May 1985</td>
<td>R</td>
<td>3.08</td>
</tr>
<tr>
<td>28 May 1985</td>
<td>R</td>
<td>1.34</td>
</tr>
</tbody>
</table>

to CCD defects, pixels disregarded by the Trigger processors, a small degree of misalignment between the cameras and an average of 5% of the imaging area lost to dome occultations. A total of 41.1 hours were observed in V-band, 8.3 hours in R-band and 6.9 hours in I-band. The solid-angle-time product observed, then, was 2.95 steradian-hours in V-band, 0.60 steradian-hours in R-band and 0.50 steradian-hours in I-band.

The choice of fields observed by the ETC test unit was affected in part by the potential for detection of possible sources of optical
transients. The declination of the center of each camera was $+25^\circ$; 2$^\circ$: the right ascensions of the center of the fields observed were within a few minutes 10$^h$, 12$^h$, 15$^h$ and 17$^h$. Generally, the fields were chosen to be near the meridian at certain hours of the night, (in order to reduce the effects of occultation by the dome and extinction by the atmosphere) but some effort was made to include potential sources of optical flashes. One of these fields (12$^h$) contained the error circles of three gamma-ray bursters (GRB1200+21 (24 November 1978), GRB1152+20 (1 January 1979) and GRB1140+20 (2 May 1979); Baity, et al., 1984), while another (17$^h$) contained the flare stars V475 Her, Ross 867 and Ross 868 (Gurzadyan, 1980).

Observations were made primarily during dark or grey time (roughly six days on either side of new moon). The sky brightness during bright time was found to be so high that the vignetting of the lens (see section 6.1.1.2) becomes significant. If the sky brightness at the center of the CCD is 200 ADU per pixel (a typical value during bright time), the sky brightness at the edge of the CCD is 140 ADU, due to the 30% vignetting of the lens. Since the Overseer Computer will set the bias level of the CCD to a point where the sky+bias level at the center of the CCD is ~40, a large area near the edge of the CCD will have sky+bias levels less than zero, rendering it useless for observations. Raising the bias level to bring the values of the edge pixels above zero would bring the values of most of the CCD pixels close to the logarithmic section of the linlog mapping, thereby reducing the sensitivity of the CCD as a whole. In
addition, the increased sky noise during bright time significantly reduced the overall sensitivity.

11.4. Results of the Observations with the ETC Test Unit

A total of 725 coincident events were reported during the observations with the ETC test unit. These events were categorized by hand, primarily through the analysis of the 9x9 pixel image subarrays of the event location from before and during the flash event. The data fell into several categories, based on the event profiles. The event categories are:

1) Satellite events: a satellite passing through the field-of-view created the event. A satellite is confirmed by second and third triggers from its trajectory. (see section 11.6 for a further discussion)

2) Inconsistent coordinate events: the events in the two cameras were obviously unrelated. This is usually determined by the relative positions of bright objects in the 9x9 image subarrays of the event from the detecting cameras.

3) Cloud triggers events: the events (always in groups of five or six, because the maximum allowed number of events per observation cycle is six) were due to patchy clouds. These events were recognized a) because of the low number of "good" integrations in that observation cycle and b) from notes made at the time in the observations log.

4) Occultation events: the event occurred in a portion of camera which was partially obscured by the dome.

5) General brightening events: the event was induced by a sudden increase in the background light level in the CCD, due to such causes as a flashlight in the dome, a passing car with its lights on, or distant lightening.

6) Streak events: the image has a streak-like appearance, but no second trigger to confirm it to be a satellite.

7) No trigger events: the data stored show no evidence of an event having occurred. This is currently an unsolved bug in either the Overseer or Trigger software.
8) Statistical events: the event was created by statistical variations in the brightness of a corresponding pixels in both detecting cameras.

A breakdown of the frequency of these types of event is given in Table 11.4, based on 16,700 exposures taken during the test observations.

11.4.1. Statistical Events

After the events of types 1-7 had been identified in the ETC data bank, a total of 355 statistical events remained. These events are subclassified into four groups, based on their morphology:

1) Class I events occurred when the brightest pixel in a stellar image brightened in both detecting cameras.

2) Class II events occurred when the brightest pixel of a stellar image brightened in one detecting camera while an immediate neighbor pixel brightened in the other.

3) Class III events occurred when a pixel neighboring the brightest pixel of a stellar image brightened in both cameras.

<table>
<thead>
<tr>
<th>Type of event</th>
<th>Number detected</th>
<th>% of total</th>
<th>Number/exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satellite</td>
<td>159</td>
<td>21.9</td>
<td>0.0095</td>
</tr>
<tr>
<td>Inconsistent coordinates</td>
<td>34</td>
<td>4.7</td>
<td>0.0020</td>
</tr>
<tr>
<td>Cloud trigger</td>
<td>110</td>
<td>15.2</td>
<td>0.0066</td>
</tr>
<tr>
<td>Occultation</td>
<td>5</td>
<td>0.7</td>
<td>0.0003</td>
</tr>
<tr>
<td>General brightening</td>
<td>13</td>
<td>1.8</td>
<td>0.0008</td>
</tr>
<tr>
<td>Streak</td>
<td>17</td>
<td>2.3</td>
<td>0.0010</td>
</tr>
<tr>
<td>No trigger</td>
<td>14</td>
<td>1.9</td>
<td>0.0008</td>
</tr>
<tr>
<td>Statistical Class I</td>
<td>194</td>
<td>26.8</td>
<td>0.0116</td>
</tr>
<tr>
<td>Statistical Class II</td>
<td>46</td>
<td>6.3</td>
<td>0.0028</td>
</tr>
<tr>
<td>Statistical Class III</td>
<td>92</td>
<td>12.7</td>
<td>0.0055</td>
</tr>
<tr>
<td>Statistical Class IV</td>
<td>43</td>
<td>5.9</td>
<td>0.0026</td>
</tr>
<tr>
<td>Total</td>
<td>725</td>
<td>100.0</td>
<td>0.0434</td>
</tr>
</tbody>
</table>

Table 11.4: Breakdown of Events Detected by the ETC Test Unit
4) Class IV events occurred when background pixels brightened in both cameras, or when a pixel near, but not immediately neighboring a very bright star triggered in both cameras. Most of these events barely exceeded the event thresholds, and all but a few of these events showed no appreciable light curve in both cameras. Because of the high CCD readout noise (50-70 $e^-$/pixel), it is probable that most, if not all of these events were caused by statistical fluctuations in the CCDs. The rate of these events (355 events in 16,700 exposures, or ~.021 events per exposure) are shown below to be comparable to the expected rate of correlated events due to statistical fluctuations in the CCD. The calculation of the expected false event rate is based on the fact that the ETC software does not detect sudden brightenings of image pixels, per se, but rather large differences in the value of image pixels from one image to the next.

11.4.1.1. The Difference-Sensitivity of the ETC

When discussing the statistics of an ETC event, one must take into account that the ETC instrument is not level-sensitive, but difference-sensitive. The ETC is not an instrument monitoring a sky of static brightness levels with a noiseless CCD, waiting for a part of the sky to increase its brightness by more than a certain amount in a single exposure time; rather, the ETC is monitoring a sky where the brightness levels vary from frame to frame due to shot noise and readout noise, waiting for the difference between two frames to exceed a certain amount in a single exposure time. The difference-sensitivity of the ETC has significant implications for the
sensitivity and expected statistical trigger rate of the ETC.

The first implication of the ETC's difference-sensitivity is the fact that the event sensitivity is lower than the sensitivity of a camera to the detection of a stellar image by a factor of \( \sqrt{2} \), as briefly mentioned in chapter 4. This is due to the fact that the readout and shot noises must be taken into account twice in the calculation of the total noise associated with an event, because two frames are involved in the event detection. The equation giving the level of significance, \( q_d \), of a detected event is

\[
q = \frac{S_{\text{after}} - S_{\text{before}}}{(S_{\text{before}} + S_{\text{after}} + 2\sigma_R)^{1/2}}
\]

where \( S_{\text{before}} \) and \( S_{\text{after}} \) are the total signal above the bias level (in electrons) in a given pixel in two consecutive images. Substituting \( S_{\text{after}} = S_{\text{before}} + \Delta S \), where \( \Delta S \) is the amount of brightening in the event, one gets

\[
q = \frac{\Delta S}{(2(S_{\text{before}} + \sigma_R^2 + \Delta S)^{1/2}}
\]  \hspace{1cm} (11.1)

The second and more important implication of the ETC's difference-sensitivity is that the significance of a detected event is not simply that determined by Gaussian statistics: that is, an event that showed a brightening corresponding to a statistical significance of \( N\sigma \), where \( \sigma \) is the total noise (the denominator of equation 11.1), cannot be assigned a confidence level associated with the Gaussian probability of an \( N\sigma \) event occurring randomly. The problem,
again, is the fact that the ETC detects an event when the value of a pixel increases by a certain value from one exposure to the next. This means that an \( N_\sigma \) event can be created if, for example, the pre-event pixel value is \( N_\sigma/2 \) below the mean value and the event pixel is \( N_\sigma/2 \) above the mean value. The probability of this situation occurring randomly is \( (p_G(N/2))^2 \) (where \( p_G(x) \) is the probability of an event of \( >x_\sigma \) occurring according to Gaussian statistics), which is significantly greater than \( p_G(N) \) for most values of \( N \). In assigning a confidence level to an event, one must therefore not use the Gaussian probability function \( p_G(x) \), but rather the probability function \( p_\Delta(x) \), which is defined as

\[
p_\Delta(x) = \int_{-\infty}^{\infty} \text{prob}(<y)\text{prob}(>(x+y))dy
\]

where \( \text{prob}(<y) \) is the Gaussian probability of an event being less than \( y_\sigma \) (including the probability of an event being below average) and \( \text{prob}(>(x+y)) \) is the Gaussian probability of an event being greater than \( (x+y)\sigma \). The function \( p_\Delta(x) \) gives the probability that the difference between a pixel's value in consecutive images will have a significance greater than \( x_\sigma \). The value of \( p_\Delta(x) \) exceeds the value of \( p_G(x) \) significantly for more values of \( x \), as can be seen in Figure 11.3.

11.4.1.2. Calculation of the expected statistical event rate

The expected rate of false ETC events due to Gaussian variations of the values of image pixels can be estimated, given the function \( p_\Delta(x) \) and the distribution of the pixel values in a typical ETC.
Figure 11.3: The relative probability of a random event in the ETC. The curve marked "Gaussian probability" indicates the probability of a pixel value being $N$ sigma above its mean value. The curve marked "Difference probability" indicates the probability of the difference in the value of a pixel in two consecutive measurements being greater than $N$ sigma (see section 11.4.1.1).
image. The distribution of pixel values in a CCD image can be divided into three populations, each with a different rate of statistical brightening. The pixels with values between a few ADU above the median of all pixel values in an image and ~140 ADU are parts of stellar images and, because of their larger shot noise, are more likely to create a statistical event than pixels with values near the median value. However, 90% of the image pixels have values within 5 ADU of the median, so the net rate of statistical events from these two populations is balanced somewhat. (Empirically, it was determined that the majority of statistical events occurred near stellar images; cf. Table 11.4). Those pixels with values greater than 150 are in the range of pixel values where each step is logarithmic, and therefore significantly larger than those in the linear range: statistical events with values greater than 150 are therefore very rare.

The expected rate of events was calculated with a computer program which, using a full ETC frame from either camera, calculates the probability of detecting a correlated event in the two cameras, based on the values of the total noise and the detection threshold in each camera. The program reads the values of the image pixels in order, calculating for each the total noise, \( \sigma \), (based on twice the signal shot noise and readout noise) in both cameras (after converting the pixel value in the one camera to the equivalent value in the other camera by multiplying by the ratio of the camera gains). From the ratio of the detection threshold value to the total noise in a pixel the probability of a random fluctuation in that pixel leading to an
event in a single camera is calculated, using the \( p(\Delta x) \) function described above. The probability of a correlated event due to random fluctuations is just the product of the two single-camera probabilities. The probability is calculated for all pixels and summed to give the probability of statistical triggers per exposure.

A direct comparison of the results of this computer analysis with the observed rate of correlated events is complicated for several reasons. First, the observations were made under many different sky conditions (dark time, grey time, bright time) and at various values of detection thresholds in both cameras (typical values were 7 ADU in camera B and 6 or 7 ADU in camera C). Those observations made under bright sky conditions (partial moon) suffer from a higher sky noise and therefore would be expected to have a higher rate of statistical events. Observations made under brightening sky conditions (during sun- or moonrise) are also expected to have a higher event rate because the sky brightness is increasing slowly during observations, thereby increasing the fraction of pixels able to pass Level 1 of sifting (see chapter 8 and below).

In addition, the rate of statistical events from pixels with values near the median value is significantly reduced by the fact that there is a probability of ~50% that the pre-event pixel value is below its mean value, in which case a typical brightening will not pass Level 1 of sifting.

Nonetheless, in order to show that statistical fluctuations are sufficient to account for the rate of correlated events observed in
the prototype ETC, the program described above was run with a dark-time image with both detection thresholds set at 7 ADU (so that a $\Delta S > 8$ ADU will trigger --- a typical situation during test observations). The rate of correlated events based on camera B and converting to camera C was 0.063 per exposure and the rate based on camera C was 0.048 per exposure, averaging to 0.05 per exposure. Taking into account that about half of the pixels with values near the median value will not trigger because they are not able to pass Level 1 of sifting (as discussed above) reduces the expected event rate by ~20%, to 0.045 events per exposure. This is more than sufficient to account for the bulk of the 355 "statistical", "stellar" and "other" events, which occurred at a rate of ~0.02 events/exposure.

A final complication of this analysis process is that the Overseer software allowed fluctuations to be considered a correlated event even if the celestial coordinates differed by up to two pixels. This means that a pixel in one camera can trigger not only with its direct counterpart in the other camera, but also with neighbors of the counterpart. The program described above was modified to deliver the probability of a correlated flare with the nine nearest pixels in the other camera: the result was an increase in the expected event rate by a factor of 2-3. Because the actual software allowed correlated events to occur not from the nearest nine pixels, but rather from an area of $<2\pi$ pixels, the actual effect is more modest. The expected increase in event rate is ~75%, raising the expected rate to ~0.08 events per exposure.
11.4.2. Satellite Detections by the ETC Test Unit

The high frequency of detection of satellites by the ETC was somewhat surprising. It had been assumed that low-Earth-orbit satellites would be moving too fast to be accepted as ETC events, due to the fact that their long trails on the CCD would be noticed by the Trigger processor streak-rejection algorithm (section 8.3.2.5). Generally, the detected satellite trails were indeed long enough to have tripped the streak rejection algorithm, yet none of the perimeter pixels had brightened sufficiently. This may be attributable to statistical fluctuations in the perimeter pixels. Indeed, the bulk of the satellite trail across the CCD very likely was rejected as a streak; only those few images of the satellite trail where the perimeter pixel was lower than it should have been due to statistical fluctuations escaped rejection. (A typical satellite triggered the ETC an average of six times during its traversal of a field: during this time, roughly 200 exposures were made of the field).

A total of 29 satellite trails were detected by the ETC during prototype observations. The satellites were divided into those with a roughly east-west trajectory and those with trajectories making an angle of roughly 60° with east-west. Judging by the times of detection and angles of the paths of some of the trails, several trails may have been made by the same satellite. Data from two typical satellite detections are shown in Figure 11.4.

The satellite trails detected were made by satellites of visual magnitudes from 6 to 9. The angular speeds of the satellites at the
Figure 11.4: Two views of ETC data collected during March and May, 1985. Figure 11.4a is a display of all image data collected after the detection of an optical flash event. The data displayed are 9x9 pixel image subarrays about the locations of the flash event and five SAO standard stars in each camera. In Figure 11.4a, the left-hand six rows contain data from camera B and the right-hand six rows contain image data from camera C. Of each set of six rows, the upper five rows contain images of SAO standard stars and the lowest row, set apart from the others, contains data from the flash location. The images are order chronologically from the left. The far left column contains the 'old' images, taken at the beginning of the observation cycle. The next column contains the 'recent' images, taken immediately before the detection of the event. The remaining columns contain data taken at and after the flash event. All exposure times are ten seconds. Figure 11.4a shows the detection of an east-west-moving satellite; Figure 11.4b, a close-up of another set of flash data, shows the detection of a satellite with an inclination of \( \sim 60 \) degrees. Both satellites are of roughly 6th to 8th magnitude.
time of detection were consistent with circular orbital periods of 4 to 12 hours: it should be noted, however, that no estimate of the orbital eccentricity was made from these detections, so that estimates of orbital periods are uncertain.

The rate of expected satellite events (~0.01 per ten-second exposure) will pose a problem for the plenary ETC, since the associated event rate is very large compared to celestial event rates. (The rate is one event per 15 minutes, on the average, but the events are clumped in groups and trigger at a rate of one event per one or two minutes in these groups). One solution is to lower the streak-rejection criterion from 7 to 5 pixels (cf. section 8.3.2.5): this criterion would have rejected a significant number of the satellites detected during the observations with the test instrument. Another possible means of rejecting satellite events is for the Overseer computer to monitor the coordinates of reported events: if any three are shown to be collinear, they would be considered having been due to a satellite crossing the field.

11.5. Interpretation of Results

The observations with the ETC test instrument yielded no optical flashes which could not be explained by local, terrestrial or statistical sources of optical transient events. The total solid-angle-time product observed by the ETC in the visual band is 2.9 steradian-hours. Given the ten-second sensitivity of the ETC test unit to detected events ($m_v = 9.9$: a 10σ criterion), the test observations were expected to have detected 1.5 flare stars and 0.008
optical transients from GRB sources (see Figure 11.5). The test results define a $3\sigma$ Poisson upper limit of 2.2 optical flashes per hour per steradian at $m_V = 10$ (Gehrels, 1985). This result is superimposed on the graph in Figure 11.5.

The results of the observations with the ETC test unit have no significant impact on differentiating between the models of optical transient events described in Chapter 3. However, the observations made with the ETC test unit make up by far the most substantial wide-field survey of the night sky for short-timescale optical transients. The $3\sigma$ Poisson upper limit of <2.2 flashes per hour per steradian is an order of magnitude lower than the $3\sigma$ upper limit of <22 flashes per hour per steradian determined by the previous best effort of Schaefer, Vanderspek, Bradt and Ricker (1984). It should be noted that this upper limit result was determined with one-sixteenth of the plenary ETC in five weeks of observation! The results achievable by the plenary ETC in one year of observation should be a factor of 160 better than the results presented here, and a factor of 1600 better than the efforts of Schaefer, Vanderspek, Bradt and Ricker (1984)!

The observations with the test unit indicate that the flash background rate (the rate of flashes from non-cosmic sources) is dominated by satellite events. The plenary ETC, with its capability of recognizing a satellite by its trail across a CCD as well as by the use of parallax, will be able to reduce significantly the event rate due to satellites. The detection of these satellites indicates that
Figure 11.5: The expected rates of optical flashes in ten-second exposures in the ETC test unit. The curves represent the expected event rates due to GRBs (section 10.1), flare stars (section 10.2), comet impacts onto white dwarfs (section 10.3), and head-on meteors (section 10.4.1) and the upper limit on the event rate due to cosmic rays. The three-sigma Poisson sensitivity of ETC observations to date are indicated, as well as the sensitivity of the Two-Schmidt survey, conducted by Schaefer, Vanderspek, Bradt and Ricker (1984).
the events detected by Pedersen, et al. (1985), may have been created by a satellite. The angular speed of a typical satellite (detected by the ETC) is "70"/second. The longest dimension of the photometer aperture used by Pedersen was "80": therefore, optical events with durations less than 1.2 seconds could be created in Pedersen's photometer by the passage of a typical satellite.
CHAPTER 12

Future Work

Introduction

Although the four-camera ETC test instrument was able to operate successfully in a semi-automatic mode during the test phase, there are a number of weak links in the ETC instrumentation. These weak links must be corrected before the ETC can be expected to operate for long periods without an observer on site. The general problem with the ETC instrumentation is that several of its components are not reliable: under certain situations, the state of the instrument hardware cannot always be consistently predicted. Plans for the correction of the problems. In addition, the weak spots in the ETC software, which all center around its operational speed, will be mentioned.

12.1. General Plans for the Improvement of the ETC

The observations made with the ETC test instrument revealed several areas for general improvement in the ETC. These improvements include:

1) The Overseer computer disk memory will be upgraded by at least 20 Mbytes to allow for more disk data storage, so that the interval between successive data storage onto magnetic tape is longer.

2) The Overseer computer core memory will be upgraded to at least 2 Mbytes to increase the speed of the Overseer software.
3) A Cipher Floppy Tape unit will be introduced as the medium for mass data storage. The Floppy Tape unit stores data onto standard 1/2" cartridge tapes.

4) The Trigger processors will be replaced by Heurikon HK68 single-board microcomputers with built-in DMA. The Heurikon board will include 1 Mbyte of hard memory to allow for substantial data storage in the Trigger processors.

5) The HSSL will be replaced by a new, more reliable fiber-optic link.

6) The ETC CCD cameras will be replaced by thermoelectrically-cooled CCD cameras designed by Gerard Luppino of MIT. The use of these cameras will eliminate the problem of the bulky coolant hose associated with the closed-cycle refrigerator. In addition, since each camera is cooled individually, a single camera can be warmed and brought to atmospheric pressure without affecting the operation of the other CCD cameras. In addition, the preamplifiers, which had been housed inside the CCD cameras, will be moved outside of the CCD housing, removing both a source of heat and a source of noise from inside the CCD housing.

7) The ETC CCD cameras will be outfitted with new, better lenses. These custom-built lenses will allow one- or two-pixel focus over a broad band (3000 A) of wavelengths, which will significantly improve the signal-to-noise performance of the ETC.

The following sections will discuss specific problems encountered during the test phase of the ETC instrument.

12.2. The ETC Electronics

The majority of the problems with the ETC electronics are related to reliability. Specifically,

1) Some parts of the Instrument Control Electronics, which are located in the second floor of the ETC dome, tend to stop working correctly when the ambient temperature drops below 0°C. The symptoms indicate that the problems are not construction-related (e.g., a cold solder joint on a circuit board), but rather are related to design flaws or the choice of chips used in the boards. The intermittent nature of these problems has hindered their exact localization to date.
2) The High Speed Serial Link (HSSL) receivers in the Trigger processors occasionally lose data. This problem is due to improper ground return wiring of the receiver chips in the HSSL receiver. Although the rate of data loss has been reduced (less than .1% of all frames are affected, and the presence of the problem is always apparent), it still should be eliminated completely.

3) One of the Digital-to-Analog converter (DAC) boards has a tendency to drop out of operation altogether every few weeks. This problem has not yet been definitively located, but the application of pressure on one or more chips on the board generally solves the problem. This problem is most likely due to the use of a prototype construction technique in fabricating the DAC boards, a method which has occasionally resulted in reliability problems in the past.

4) The ICE in general have a problem with the loss of static-sensitive CMOS chips because of electrostatic discharges. Although the loss rate due to static electricity is low, it must be reduced to zero in the plenary ETC.

The solutions to these problems which have been proposed to date are:

1) Put the ICE in a warm place. The locations of the sources of the problems of cold-sensitivity to date have not yet been found. Even if they were, the problem could occur in other locations in the ICE. The best solution is to create a warm area in the dome for the ICE.

2) The problems with the HSSL receivers is not so severe as to warrant major revisions of its construction. However, the new fiber-optic HSSL should eliminate all the problems associated with the present HSSL.

3) Recently, printed-circuit DAC boards were constructed for use at MIT. These boards do not have the problem of loose sockets and should prove to be error-free in use in the ETC.

4) Much effort must be yet put into static protection of the ICE. The carpeting in the second floor of the ETC dome must be covered with static-free matting to guard against accidental static discharges to the electronics.

In addition to the above reliability problems, two specific aspects of the ETC electronics should be improved. First, the effective CCD readout noise is higher than specified for the CCDs. This
increased noise is due to improper "tuning" of the CCD analog electronics. Future ETC CCD camera systems should be thoroughly "tuned" before they are sent out into the field. The latest generation of CCD electronics has a much better noise performance than the present ETC cameras: system noises of less than 20 e⁻ with good charge transfer efficiency are being achieved regularly.

Second, the dome encoding scheme should be improved. Presently, it is a relative encoder, with no absolute zero point. This situation makes it necessary to rotate the dome to a specific position regularly in order to zero the encoder. This problem will be corrected in the Fall of 1985 by encoding the dome absolutely.

12.3. The ETC Instrument Mechanical Hardware

The current ETC mechanical hardware also has a number of problems which will hamper progress towards automation. The major problems are:

1) The vacuum cavity which is shared by the four ETC cameras and the manifold is not leak-free. The system maintains a good vacuum (with the closed-cycle refrigerator on) for ten days to two weeks at a time, but then can spontaneously return to near atmospheric pressure with little warning. Evidently, the cooling probe acts as a cryopump when cold, adsorbing molecules leaked into the system onto its surface. When the surface of the cooler is saturated, the pressure increases slowly due to a leak in the system, until the ion pump shuts off, resulting in a rapid pressure rise.

2) The telescope mount is misbalanced, primarily due to the weight of the coolant hose wrapped around the axis of the manifold. The result is that it is very difficult to slew the telescope mount at hour angles of more than ~ 3 hours.

Proposed solutions to these problems are:
1) A new camera/manifold system will be used in future ETC cameras. The present system relies heavily on rubber O-rings between flat aluminum faces. The improved system will utilize steel knife-edge vacuum interfaces with rubber gaskets, so the vacuum integrity of the system will be much improved.

2) The cooling system in future ETC cameras will not require stiff cooling hoses. The new cameras, designed by Gerard Luppino, are cooled by thermoelectric coolers and will not impede the slewing of the mount. Without the coolant hose, which tended to accumulate ice when cold, balancing the telescope will also not be as difficult a problem as it is now.

Another advantage to the new ETC camera/manifold design is that access to the camera interiors will be much simpler than it is now. The method of cooling individual cameras will allow one camera to be taken off line without warming the others.

12.4. ETC Software

The main problem with the ETC software is that it is not fast enough to support one- to three-second integration times. There are several bottlenecks in the software which slow it down substantially:

1) Whenever a candidate report is received by the Overseer computer, the location of the event is converted into its celestial coordinates. The conversion of event location to celestial coordinates involves calculating of \( \sim 10 \) trigonometric functions, with each function requiring 10 milliseconds. Thus, each candidate report requires \( \sim 0.1 \) seconds to be fielded.

2) The transfer of event image data between the Trigger processors and the Overseer computer takes \( \sim 0.6 \) seconds per flash event per exposure.

The next revision of the ETC software will transfer more of the computational burden from the Overseer computer to the Trigger processors, which should improve overall system speed. Specifically,
1) Each Trigger processor will be given the parameters of the astrometric mapping of its associated camera, so that it may calculate the celestial coordinates of each candidate event itself. In this way, the Overseer computer will act as a bookkeeper of candidate events, and not as a calculator. In addition, the coordinates will be calculated in the form of three-vectors, which avoids the calculation of slow trigonometric functions.

2) Each Trigger processor will store image data in its own memory structure during observations. At the end of an observation cycle, each Trigger processor can then transfer its image data to the Overseer computer without time pressures.

Some general areas for the improvement of the ETC software are:

1) The sifting algorithms do not reject signal fluctuations in bright stars. Some method of recognizing the higher shot noise in bright stars must be devised.

2) The criteria for coincidence of celestial coordinates must be tighter than in the test observations, in order to reduce the rate of statistical false triggers.

3) The calculation of the precise celestial coordinates of the center of each CCD, presently observer-aided in "user" mode (see Chapter 11), must be made automatic. A field-finding algorithm, used by the third Small Astronomical Satellite (SAS-3), will be altered for use in the ETC.
CHAPTER 13

Acknowledgements

The number of people who were involved in getting the ETC test instrument working is large. The ETC test instrument is intricate and complex, and without the help of these people, the test unit could easily have incurred delays of a year or more. The people primarily responsible for the successful construction of the ETC test unit are John Vallerga, Gerard Luppino, George Ricker and John Doty.

George Ricker, as my advisor, was responsible for the development of the concept and realization of the ETC. His choice of me to design and construct the test ETC was, for me, an excellent one, as it allowed me to develop as an experimental and observational astrophysicist in addition to working on one of the most exciting astronomical projects of the decade. George's experience in observational astronomy and his training of me have stood me in good stead during the construction of the ETC. George also provided useful suggestions and guidance during the writing and implementation of the Overseer software.

John Doty's contributions to the ETC were in many ways more subtle. The bulk of the CCD electronics, the concept of the Overseer computer and the HSSL were all his brainchild, and the refinement of these aspects of the ETC is being assisted by him. The concept of how the Trigger software should work was also developed by John Doty.
His further contributions have been in the form of streamlining the Trigger and Overseer software: the five days he spent at the ETC site at Kitt Peak should be counted as some of the most critical time spent with the ETC by anyone, since he used the time to remove an intricate system problem that would have made efficient execution of the ETC software very difficult.

John Vallerga and Gerry Luppino were indescribably helpful in teaching me how to build an instrument. With their guidance, many mistakes in the construction and testing of the ETC electronics were avoided or caught before they became critical. In addition, their ongoing efforts to improve the CCD electronics proved very useful for the ETC ICE, which was developed in parallel with their efforts.

In addition to the above four, several undergraduates at MIT have made significant contributions to the ETC instrument. Steven Rosenthal was responsible for the development and implementation of the DMA/HSSL transmitter/receiver boards (with Charles Kimball working some on the ICE HSSL transmitter). Steve also wrote and tested the Trigger software as his senior thesis.

Duane Thresher was the primary builder and tester of the ETC's clock driver generation and analog processing boards. His knowledge, patience, good humor and invariable presence at odd hours made the construction of the ETC much more bearable.

Howard Stearns developed and built the ETC frame timer and the Trigger hardware reset. His hardware implementations and Overseer driver code changes have worked almost flawlessly.
George Mitsuoka was significant in contributions to the writing of several drivers, not least noticeably the HSSL driver. Erica Ellingson wrote her senior thesis on the ETC optical system and the effects of cosmic rays on ETC observations.

Kip Dee Kuntz's efforts to observe with an underdeveloped ETC in December, 1984, are noted with admiration and gratitude. His presence at Kitt Peak will be long remember by many.

Among the "cast of thousands" of undergraduates at MIT who worked on the ETC as UROP students in a sometimes more, sometimes less effective way were Anna Franco, Carlos Montero-Luque, Nancy Ellman and Geoff Engelstein.

Several non-MIT persons were deeply involved in the ETC project. Dr. Frank Melsheimer of DFM Engineering, Inc., was the primary designer of the ETC camera/manifold system and the interface with the MFC-100 cooler. Cindy Reiter and Doug Fraser of FTS Systems, Inc., were very helpful in the selection and testing of the MFC-100 cooler for the ETC test unit.

I would also like to thank Sidney Wolff, Buddy Powell, Bob Barnes, Dick Doane, John Settlemyre, John Africano and the TA support staff at KPNO for their patience and help during the construction of the ETC.

I would also like to thank Cheryl Simmons and Dan Calileo of MIT's Center for Space Research for their patience and guidance in the management of the ETC budget.
I thank Claude Canizares and Paul Joss for a careful reading of this thesis and a bloodless thesis defense. Saul Rappaport also provided useful comments and suggested the Monte Carlo approach to the flare star calculations.

Finally, I'd like to thank the countless people who supported the ETC effort and my family and friends for supporting me as a person during the sometimes very trying moments during the construction and testing of the ETC.
References


Lipunov, V.M., Moskalenko, E.I., and Shakura, N.I. 1982, Astrophys...


Nautical Almanac, 1984, U.S. Naval Observatory.


Rappaport, S.A, and Joss, P.C. 1985, preprint, to be published in
Nature.


APPENDIX A

The Overseer Computer Software

Introduction

This Appendix contains the entire ETC Overseer Software code as it was used in the test observing runs of March and May, 1985. There is no separate documentation for the Overseer software: instead, individual subroutines are documented in the code as necessary. The reader is referred to Chapter 7 for an overview of the functions of the different modules in the Overseer software.
```c
#define SUBARRAY_SIZE 9
#define S_SQUARED 1
#define COINCIDENCE 1
#define NUM_of_CAMERAS 2
#define NUM_COSMACS 1
#define SQRT_ROM 1
#define NUM_STANDARDS_per_FLASH 5
#define NUM_of_STANDARDS_per_FOV 16
#define MAX_EXPOSURES_per_FOV 10
#define MAX_FLASHES_per_NIGHT 6
#define MAX_NUM_of_ACTIVE_FLASHES 40
#define NROWS 292
#define NCOLUMNS 400
#define PIXEL_SIZE 22300000 /* picometers */
#define NUM_PIXEL_DEFECTS 10
#define EAST_LONGITUDE 4335398 /* microradians at KPNO */
#define LATITUDE 557865 /* microradians at KPNO */
#define PI 3.1415926535
#define TWO_PI 6.2831853070
#define PiE6 3141593
#define TWO_PIe6 6283185
#define NUM_of_HOUSE_DATA 10
#define MAX_TRIES 150 /* adu */
#define MAX_TEMPERATURE 891 /* two 50 mm pixels */
#define MAX_RMS 100
#define MAX_HOURANGLE 1178000 /* 4.5 hours */
#define PATIENCE 0.001389 /* two minutes */
#define MAX_BYTES 1000000 /* after which data is store */
#define MAXBYTES 1 /* on tape */
#define INIT_REQUEST 2
#define START_TIMER 3
#define ZERO_DETECT 4
```

#include "constants.h"

struct Position_str
{
    int x, y;
};

struct Coordinate_str
{
    int x, y;
};

struct Double_pair_str
{
    double a, b;
};

struct Subarray_str
{
    unsigned char array[S_SQUARED];
};
struct Per_camera_str
{
    struct Subarray_str Flash_array, Stand_array[NUM_STANDARDS_per_FLASH];
    int Flash_photometry, Stand_photometry[NUM_STANDARDS_per_FLASH];
};

struct Per_exposure_str
{
    struct Double_pair_str Exposure_time;
    struct Per_camera_str Data_from_exposure[COINCIDENCE+1];
};

struct Per_flash_data_str
{
    struct Subarray_str R_array[COINCIDENCE+1];
    struct Subarray_str
        Rec_array_standard[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    struct Subarray_str O_array[COINCIDENCE+1];
    struct Subarray_str Old_arr_standard[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    int Rec_photometry[COINCIDENCE+1];
    int Rec_standard_photometry[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    int Old_photometry[COINCIDENCE+1];
    int Old_standard_photometry[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    struct Coordinate_str Flash_coordinates[COINCIDENCE+1];
    struct Coordinate_str Stand_coordinates[NUM_STANDARDS_per_FLASH];
    struct Position_str Flash_position[COINCIDENCE+1];
    struct Position_str Stand_position[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    unsigned char Stand_number[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    unsigned char Flash_number[COINCIDENCE+1];
    int Camera_number[COINCIDENCE+1];
    unsigned char Row_threshold[(COINCIDENCE+1)*NCOLUMNS];
    char Column_threshold[(COINCIDENCE+1)*NROWS];
    struct Per_exposure_str Single_exposure_data[MAX_EXPOSURES_per_FLASH];
};

struct Storage_str
{
    int Number_of_exposures;
    struct Per_flash_data_str Single_flash_data;
    Data_storage[MAX_FLASHES_per_NIGHT];
};

int Num_stored_flashes;
struct Active_per_camera_str
{
  int Camera_number;
  struct Position_str Flash_position;
  struct Coordinate_str Flash_coordinates;
  struct Position_str Stand_position[NUM_STANDARDS_per_FLASH];
  unsigned char Trigger_flash_number;
  unsigned char Stand_trigger_number[NUM_STANDARDS_per_FLASH];
};

struct Per_active_entry_str
{
  struct Coordinate_str Stand_coordinates[NUM_STANDARDS_per_FLASH];
  double Update_time;
  int Storage_number, Number_of_exposures;
  struct Active_per_camera_str Live_camera_entry[COINCIDENCE+1];
  } Live_flash_file[MAX_NUM_of_ACTIVE_FLASHES];
  int Num_of_active_entries;

struct Standard_archive_str
{
  struct Coordinate_str sao_coords[NUM_of_STANDARDS_per_FOV];
  struct Position_str sao_pos[NUM_of_STANDARDS_per_FOV];
  short sao_mag[NUM_of_STANDARDS_per_FOV];
  int sao_brightness[NUM_of_STANDARDS_per_FOV];
  int sao_number[NUM_of_STANDARDS_per_FOV];

  struct Subarray_str sao_start_array[NUM_of_STANDARDS_per_FOV];
  struct Subarray_str sao_end_array[NUM_of_STANDARDS_per_FOV];
};

struct archive_str
{
  double start_time_of_run, end_time_of_run;
  int num_stored_flashes, num_of_cameras, num_cosmacs, coincidence;
  int max_exposures_per_flash, max_flashes_per_night, num_of_standards_per_fo;
  int num_standards_per_flash, nrows, ncolumns, subarray_size;
  int d_str_length, c_str_length, m_str_length, t_str_length, s_str_length;
  int good_exposure_counter[NUM_of_CAMERAS], counter_camera[NUM_of_CAMERAS];

  struct Standard_archive_str standard_archives[NUM_of_CAMERAS];
} Archives;
struct C_option_str
{
  int declination;                       /* microradians */
  int right_ascension;                  /* microradians */
  int ra_difference;                    /* difference from M_data[].RA */
  int dec_difference;                   /* difference from M_data[].RA */

  int focal_length;                     /* microns */
  float f_number;
  float fudge_factor;
  int ccd_angle;                        /* microradians */
  int area_of_lens;                     /* square microns */
  int tilt_angle;                       /* zero */

  double c_to_p_parameters[6];
  double p_to_c_parameters[6];
  int radians_per_pixel;

  float gain;
  float readout_noise;
  float total_noise;

  int quiescent_difference;
  int bias_level;
  unsigned char threshold_offset;      /* number of sigma */
  unsigned char adu_offset;            /* number of sigma */
  unsigned char brightened_delta;
  unsigned char adu_delta;

  float linear_term, quadratic_term;   /* vignetting data */
  float peak_efficiency;
  float fifteenth_mag;

  unsigned char temperature;           /* adu */
} C_data[NUM_OF_CAMERAS];

struct M_option_str
{
  int synchro_angle_of_meridian;
  int synch_units_per_radian;
  int right_ascension;
  int start_ha, end_ha;                /* start and end hourangles for */
  /* normal operation in microradians */
} M_data[NUM_OF_CAMERAS];

struct T_option_str
{
  unsigned char cosmic_sensitivity;
  short sift_time;
} T_data;

struct S_option_str
{
  short exposure_time;
  double astrometry_interval;
  double slew_time;
}
double time_after_subsidal;
double dome_interval;
} S_data;

struct jpdfitxy
{
  double x0,y0,z0,fx0,fx1,fx2,fx3,fy0,fy1,fy2,fy3;
} Jpdc_to_p[NUM_of_CAMERAS];

struct jpdxyfit
{
  double f1,f2,f3,fx1,fx2,fx3,fy1,fy2,fy3;
} Jpdp_to_c[NUM_of_CAMERAS];
struct SAO_entry_str  /* the trigger number is the sequence number */  
{
  struct Coordinate_str SAO_coordinates[NUM_of_STANDARDs_per_FOV];
  struct Position_str SAO_position[NUM_of_STANDARDs_per_FOV];
  short SAO_magnitude[NUM_of_STANDARDs_per_FOV];
  int SAO_brightness[NUM_of_STANDARDs_per_FOV];
  int SAO_star_number[NUM_of_STANDARDs_per_FOV];
  } SAO_file[NUM_of_CAMERAS];

double Sunrise, Sunset, Slew_time, Astrometry_time, Dome_adjust_time;
double Start_time_of_run, End_time_of_run, AM_twilight, PM_twilight;

int Cosmac_port[NUM_COSMACS];
int Cos_status[NUM_COSMACS];
int Trig_port[NUM_of_CAMERAS];
int Trig_status[NUM_of_CAMERAS];
int Overseer_status;
int Timer_port;
int Extr_port;
int RMT_port;
int Automatic;
int Total_number_of_exposures;

struct House_str  
{
  unsigned char temp[4], current[4], t_setting[4], c_setting;
  } Housekeeping[NUM_COSMACS];

struct Subarray_str HSSL_array[3];
struct Double_pair_str Image_exposure_time;
char Used_trigger_number[NUM_of_CAMERAS*64];

struct Astr_data_str  
{
  int ra, dec, x, y, expected_brightness, real_brightness;
  } Ast_data[NUM_of_STANDARDs_per_FOV];

char Column_threshold[NUM_of_CAMERAS*NROWS];
unsigned char Row_threshold[NUM_of_CAMERAS*NCOLUMNS];

int C_str_length, M_str_length, T_str_length, S_str_length, D_str_length;
int A_str_length;

int Amplitude_of_synchro_error[NUM_COSMACS];
int Zero_point_of_synchro_error[NUM_COSMACS];
int Phase_of_synchro_error[NUM_COSMACS];

struct C_defect_str  
{
  int defective_column[NCOLUMNS];
  } Col_defect_table[NUM_of_CAMERAS];

struct P_defect_str  
{
  struct Position_str location[NUM_PIXEL_DEFECTS];
}
} Pixel_defect_table[NUM_of_CAMERAS];

short rom_mapping[256];
char letters[NUM_of_CAMERAS];

int Offset[NUM_of_CAMERAS];

float Sky_brightness[NUM_of_CAMERAS];
float Sky_sigma, Global_brightness; /* a UTsky_brightness kluge

int Cam_azimuth, Dome_azimuth;

int DAYTIME, WEATHER_BAD, MOUNTS_LOCKED, TRACKING, DOME_OPEN, PANICKING, CLOUDS;
int OBSERVED, ENCODER_WORKED, ENC_WORKING;
#include "constants.h"
#include "extern.h"

#define CAMA 0
#define CAMB 0
#define CAMC 1
#define CAMD 3

DEclarations()
{
    /*
    Routine contains simple declarations of system constants, such as
    gain, noise, and so on.
    
    Present configuration is set for [CAMA] corresponding to camera A,
    [CAMB] corresponding to camera B,
    [CAMC] corresponding to camera C.
    */
    int i,j,chan;

    /*chan = open("linlog_table",0);*/
    chan = open("sqrt_ROM",0);
    if (chan == -1) printf("Cannot open sqrt_ROM\n");
    read(chan,rom_mapping,512);
    close(chan);

    letters[CAMA] = 'A';
    letters[CAMB] = 'B';
    letters[CAMC] = 'C';

    A_str_length = sizeof(struct archive_str);
    C_str_length = sizeof(struct C_options_str);
    M_str_length = sizeof(struct M_options_str);
    S_str_length = sizeof(struct S_options_str);
    D_str_length = sizeof(struct Storage_str);
    T_str_length = sizeof(struct T_options_str);

    Archives.max_flashes_per_night = MAX_FLASHES_per_NIGHT;
    Archives.num_of_standards_per_fov = NUM_OF_STANDARDS_per_FOV;
    Archives.num_standards_per_flash = NUM_STANDARDS_per_FLASH;
    Archives.num_of_cameras = NUM_OF_CAMERAS;
    Archives.num_cosmacs = NUM_COSMACS;
    Archives.coincidence = COINCIDENCE;
    Archives.max_exposures_per_flash = MAX_EXPOSURES_per_FLASH;
    Archives.nrows = NROWS;
    Archives.ncolumns = NCOLUMNs;
    Archives.subarray_size = SUBARRAY_SIZE;
    Archives.d_str_length = D_str_length;
    Archives.c_str_length = C_str_length;
    Archives.m_str_length = M_str_length;
    Archives.t_str_length = T_str_length;
    Archives.s_str_length = S_str_length;

    T_data.cosmic_sensitivity = 0;
    T_data.sift_time = 220;}
S_data.exposure_time = 500;
S_data.time_after_subsidal = 0.000347; /* 30 seconds */
S_data.astrometry_interval = 0.0138889; /* 20 minutes */
S_data.dome_interval = 0.0138889;

M_data[0].synchro_angle_of_meridian = 9400;
M_data[0].synch_units_per_radian = 2608;

C_data[CAMA].focal_length = 25000;
C_data[CAMB].focal_length = 55000;
C_data[CAMC].focal_length = 50000;

C_data[CAMA].f_number = 0.85;
C_data[CAMB].f_number = 1.2;
C_data[CAMC].f_number = 1.4;

C_data[CAMA].fudge_factor = 1.01;
C_data[CAMB].fudge_factor = 1.005;
C_data[CAMC].fudge_factor = 1.00;

C_data[CAMA].ccd_angle = TWO_PIe6/4;
C_data[CAMB].ccd_angle = -10700;
C_data[CAMC].ccd_angle = -3000;

C_data[CAMA].declination = 312100;
C_data[CAMB].declination = 549400;
C_data[CAMC].declination = 541100;

/* difference[i] = value[i] - value[CAMA] */

C_data[CAMA].ra_difference = 0;
C_data[CAMB].ra_difference = 0;
C_data[CAMC].ra_difference = -8000;

C_data[CAMA].dec_difference = 0;
C_data[CAMB].dec_difference = 0;
C_data[CAMC].dec_difference = 3700;

C_data[CAMA].gain = 1.0/66.0;
C_data[CAMB].gain = 1.0/62.0;
C_data[CAMC].gain = 1.0/51.0;

C_data[CAMA].readout_noise = 100.0;
C_data[CAMB].readout_noise = 70.0;
C_data[CAMC].readout_noise = 52.0;

C_data[CAMA].brightened_delta = 25;
C_data[CAMB].brightened_delta = 8;
C_data[CAMC].brightened_delta = 8;

C_data[CAMA].threshold_offset = 25;
C_data[CAMB].threshold_offset = 8;
C_data[CAMC].threshold_offset = 8;

Amplitude_of_synchro_error[0] = 19000;
Zero_point_of_synchro_error[0] = -8500;
Phase_of_synchro_error[0] = 9000;

OBSERVED = 0;
PANICKING = 0;

for (j=0; j<NUM_of_CAMERAS; j++)
{
    for (i=0; i<NUMPIXEL_DEFECTS; i++)
    {
        Pixel_defect_table[j].location[i].x = 0;
        Pixel_defect_table[j].location[i].y = 0;
    }
    for (i=0; i<NCOLUMNS; i++) Col_defect_table[j].defective_column[i] = 0;
}

for (i=99; i<105; i++) Col_defect_table[CAMB].defective_column[i] = 1;
for (i=93; i<108; i++) Col_defect_table[CAMC].defective_column[i] = 1;

/* These declarations are set so that the value of */
/* Archives.good_exposure_number is incremented correctly */

Archives.counter_camera[CAMA] = CAMA;
Archives.counter_camera[CAMB] = CAMB;
Archives.counter_camera[CAMC] = CAMB;

/* General calculative phase */

for (i=0; i<NUM_of_CAMERAS; i++)
{
    C_data[i].radians_per_pixel = PIXEL_SIZE/C_data[i].focal_length;
    C_data[i].area_of_lens = PI*(C_data[i].focal_length/C_data[i].f_number)
    *(C_data[i].focal_length/C_data[i].f_number)/4.0;
}
/* Declarations */
#include "constants.h"

extern struct Position_str
  {
    int x,y;
  };

extern struct Coordinate_str
  {
    int x,y;
  };

extern struct Double_pair_str
  {
    double a,b;
  };

extern struct Subarray_str
  {
    unsigned char array[S_SQUARED];
  };

extern struct Per_camera_str
{
    struct Subarray_str Flash_array, Stand_array[NUM_STANDARDS_per_FLASH];
    int Flash_photometry, Stand_photometry[NUM_STANDARDS_per_FLASH];
};

extern struct Per_exposure_str
{
    struct Double_pair_str Exposure_time;
    struct Per_camera_str Data_from_exposure[COINCIDENCE+1];
};

extern struct Per_flash_data_str
{
    struct Subarray_str R_array[COINCIDENCE+1];
    struct Subarray_str
        Rec_array_standard[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    struct Subarray_str O_array[COINCIDENCE+1];
    struct Subarray_str Old_arr_standard[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    int Rec_photometry[COINCIDENCE+1];
    int Rec_standard_photometry[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    int Old_photometry[COINCIDENCE+1];
    int Old_standard_photometry[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    struct Coordinate_str Flash_coordinates[COINCIDENCE+1];
    struct Coordinate_str Stand_coordinates[NUM_STANDARDS_per_FLASH];
    struct Position_str Flash_position[COINCIDENCE+1];
    struct Position_str Stand_position[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    unsigned char Stand_number[(COINCIDENCE+1)*NUM_STANDARDS_per_FLASH];
    unsigned char Flash_number[COINCIDENCE+1];
    int Camera_number[COINCIDENCE+1];
    unsigned char Row_threshold[(COINCIDENCE+1)*NCOLUMNS];
    char Column_threshold[(COINCIDENCE+1)*NROWS];
    struct Per_exposure_str Single_exposure_data[MAX_EXPOSURES_per_FLASH];
};

extern struct Storage_str
{
    int Number_of_exposures;
    struct Per_flash_data_str Single_flash_data;
    } Data_storage[MAX_FLASHES_per_NIGHT];

extern int Num_stored_flashes;
extern struct Active_per_camera_str
{
    int Camera_number;
    struct Position_str Flash_position;
    struct Coordinate_str Flash_coordinates;
    struct Position_str Stand_position[NUM_STANDARDS_per_FLASH];
    unsigned char Trigger_flash_number;
    unsigned char Stand_trigger_number[NUM_STANDARDS_per_FLASH];
};

extern struct Per_active_entry_str
{
    struct Coordinate_str Stand_coordinates[NUM_STANDARDS_per_FLASH];
    double Update_time;
    int Storage_number, Number_of_exposures;
    struct Active_per_camera_str Live_camera_entry[COINCIDENCE+1];
} Live_flash_file[MAX_NUM_of_ACTIVE_FLASHES];

extern int Num_of_active_entries;

extern struct Standard_archive_str
{
    struct Coordinate_str sao_coords[NUM_of_STANDARDS_per_FOV];
    struct Position_str sao_pos[NUM_of_STANDARDS_per_FOV];
    short sao_mag[NUM_of_STANDARDS_per_FOV];
    int sao_brightness[NUM_of_STANDARDS_per_FOV];
    int sao_number[NUM_of_STANDARDS_per_FOV];

    struct Subarray_str sao_start_array[NUM_of_STANDARDS_per_FOV];
    struct Subarray_str sao_end_array[NUM_of_STANDARDS_per_FOV];
};

extern struct archive_str
{
    double start_time_of_run, end_time_of_run;
    int num_stored_flashes, num_of_cameras, num_cosmacs, coincidence;
    int max_exposures_per_flash, max_flashes_per_night, num_of_standards_per_fo;
    int num_weather, num_blend, num_merge, subarray_size;
    int d_str_length, c_str_length, m_str_length, t_str_length, s_str_length;
    int good_exposure_counter[NUM_of_CAMERAS], counter_camera[NUM_of_CAMERAS];

    struct Standard_archive_str standard_archives[NUM_of_CAMERAS];
} Archives;
extern struct C_option_str
{
    int declination;        /* microradians */
    int right_ascension;    /* microradians */
    int ra_difference;      /* difference from M_data[].RA */
    int dec_difference;     /* difference from M_data[].RA */
    int focal_length;       /* microns */
    float f_number;
    float fudge_factor;
    int ccd_angle;
    int area_of_lens;       /* square microns */
    int tilt_angle;         /* zero */
}

double c_to_p_parameters[6];
double p_to_c_parameters[6];
int radians_per_pixel;
float gain;
float readout_noise;
float total_noise;
int quiescent_difference;
int bias_level;
unsigned char threshold_offset;
unsigned char adu_offset;
unsigned char brightened_delta;
unsigned char adu_delta;
float linear_term;         /* vignetting data */
float quadratic_term;      /* e/photon */
float peak_efficiency;
float fifteenth_mag;
unsigned char temperature; /* adu */
} C_data[NUM_of_CAMERAS];

extern struct M_option_str
{
    int synchro_angle_of_meridian;
    int synch_units_per_radian;
    int right_ascension;
    int start_ha, end_ha;      /* start and end hourangles for */
} M_data[NUM_COSMACS];

extern struct T_option_str
{
    unsigned char cosmic_sensitivity;
    short sift_time;
} T_data;

extern struct S_option_str
{
    short exposure_time;
    double astrometry_interval;
    double slew_time;
}
double time_after_subsidal;
double dome_interval;
} S_data;

extern struct jpdfitxy
{
  double x0,y0,z0,fx0,fx1,fx2,fx3,fy0,fy1,fy2,fy3;
} Jpdc_to_p[NUM_of_CAMERAS];

extern struct jpdxylfit
{
  double f1,f2,f3,fx1,fx2,fx3,fy1,fy2,fy3;
} Jpdf_to_c[NUM_of_CAMERAS];
extern struct SAO_entry_str /* the trigger number is the sequence number:
{
    struct Coordinate_str SAO_coordinates[NUM_of_STANDARDS_per_FOV];
    struct Position_str SAO_position[NUM_of_STANDARDS_per_FOV];
    short SAO_magnitude[NUM_of_STANDARDS_per_FOV];
    int SAO_brightness[NUM_of_STANDARDS_per_FOV];
    int SAO_star_number[NUM_of_STANDARDS_per_FOV];
} SAO_file[NUM_of_CAMERAS];

extern double Sunrise, Sunset, Slew_time, Astrometry_time, Dome_adjust_time;
extern double Start_time_of_run, End_time_of_run, AM_twilight, PM_twilight;

extern int Cosmac_port[NUM_CAMERAS];
extern int Cos_status[NUM_CAMERAS];
extern int Trig_port[NUM_of_CAMERAS];
extern int Trig_status[NUM_of_CAMERAS];
extern int Overseer_status;
extern int Timer_port;
extern int Extr_port;
extern int RMT_port;
extern int Automatic;
extern int Total_number_of_exposures;

extern struct House_str
{
    unsigned char temp[4], current[4], t_setting[4], c_setting;
} Housekeeping[NUM_CAMERAS];

extern struct Subarray_str HSSL_array[3];
extern struct Double_pair_str Image_exposure_time;
extern char Used_trigger_number[NUM_of_CAMERAS*64];

extern struct Astr_data_str
{
    int ra, dec, x, y, expected_brightness, real_brightness;
} Ast_data[NUM_of_STANDARDS_per_FOV];

extern char Column_threshold[NUM_of_CAMERAS*NROWS];
extern unsigned char Row_threshold[NUM_of_CAMERAS*NCOLUMNS];

extern int C_str_length, M_str_length, T_str_length, S_str_length, D_str_length
extern int A_str_length;

extern int Amplitude_of_synchro_error[NUM_CAMERAS];
extern int Zero_point_of_synchro_error[NUM_CAMERAS];
extern int Phase_of_synchro_error[NUM_CAMERAS];

extern struct C_defect_str
{
    int defective_column[NCOLUMNS];
} Col_defect_table[NUM_of_CAMERAS];

extern struct P_defect_str
{
    struct Position_str location[NUM_PIXEL_DEFECTS];
}
extern short rom_mapping[256];
extern char letters[NUM_of_CAMERAS];
extern int Offset[NUM_of_CAMERAS];
extern float Sky_brightness[NUM_of_CAMERAS];
extern float Sky_sigma, Global_brightness; /* a UTsky_brightness kluge */
extern int Cam_azimuth, Dome_azimuth;
extern int DAYTIME, WEATHER_BAD, MOUNTS_LOCKED, TRACKING, DOME_OPEN, PANICKING;
extern int CLOUDS, OBSERVED, ENCODER_WORKED, ENC_WORKING;
/* Frame file header for ccphot */
/* 10/27/83 jpd */

struct ccheader {
    char xframe[20];    /* File name */
    short nim;          /* # of images in file */
    short nphi;         /* # columns */
    short npvi;         /* # rows */
    short p0;           /* first column # */
    short lo;           /* first row # */
    short iseq;         /* sequence # */
    short texp;         /* exposure in seconds */
    short tmsec;        /* additional millisecond */
    char idt[3];        /* month, day, year */
    char itm0[3];       /* start hour, min, sec */
    char itm1[3];       /* end hour, min, sec */
    char iwp;           /* write protect switch */
    char ids;           /* 0 = direct, 1 = spectrometer */
    char ib;            /* 1 if bias subtracted */
    char iff;           /* 1 if flat field corrected */
    char id;            /* 1 if dark subtracted */
    char iov;           /* 1 if overclock subtracted */
    char nbpx;          /* bytes per pixel */
    short l0act;        /* first active row */
    short l1act;        /* last active row */
    short p0act;        /* first active column */
    short plact;        /* last active column */
    char icom[80];      /* comment */
};
This module contains the entry-level modules of the ETC, including the setting of programmed values (via DEclarations()), opening all serial ports to peripherals, and testing whether peripheral computers (Triggers and COSMACs) are alive.

#include "constants.h"
#include "extern.h"
#include <stdio.h>
#include <sgtty.h>

/*** DECLARE EXTERNAL ROUTINES ***/

extern OPsetup_operations(); OPsetup_triggers();

extern COdownload(), COslew_clutch(), COtrack_clutch(), COSinit();
extern COenter_hostcom(), CORouse_cosmac(), COexit_hostcom();
extern COcheck_full_load(), COSend_break();

extern EXphone_home();

extern WWu();

extern TRsetup(), TRreset_trigger(), TRload_trigger();
extern char TRnull_query();
extern TRgo_trigger(), TRset_flash_subarray_size();
extern TRset__standard_subarray_size();

extern UTperipheral_check(), UTinitialize_variables();

extern DEclarations();

extern USEr();

extern IMage();
struct sgttyb our_port_settings;

ENTRY()

/* The program where all the overseer routines start. Assumes system was just powered up. Checks to see whether Cosmacs and triggers are downloaded and downloads to them if necessary. Also checks to see whether the Cosmacs and triggers are working correctly (if they respond to a prompt). Sets Overseer_status to 3 (daytime) and runs operations().
 */

char command[];
int i, flag = 1;

ENTRY();
/* opens trigger and cosmacs ports */
DECLARATIONS();
/* WWv(); */

/* check trigger stati and download if necessary */
ENTRY_TRIGGER_CHECK();
/* routine determines trigger status and records it
 */
ENTRY_COSMAC_CHECK();
/* routine determines cosmac status and records it */
COENTER_HOSTCOM(Cosmac_port[0]);

for (i = 0; i < NUM_COSMACS; i++) COTRACK_CLUTCH(i);

UTPERIPHERAL_CHECK();
/* checks peripherals */
UTINITIALIZE_VARIABLES();

while (flag)
{
    printf("ETC Entry Menu\nCommand? ");
    fflush(stdout);
    scanf("%s", command);
    switch (command[0])
    {
    case 'n':
        for (i = 0; i < NUM_COSMACS; i++) COTRACK_CLUTCH(i);
        OPSETUP_OPERATIONS();
        break;
    }
case 'u':
    for (i=0; i<NUM_COSMACS; i++)
        COtrack_clutch(i);
    USER();
    break;

case 'i':
    IMAGE();
    break;

case 'x':
    for (i=0; i<NUM_COSMACS; i++)
    {
        COSlew_clutch(i);
        COexit_hostcom(Cosmac_port[i]);
    }
    exit(0);
    flag = 0;
    break;

default:
    printf("\n\nETC entry routine menu:\n\n");
    printf("n Normal automatic operation (out of order)\n");
    printf("u User-guided observing software\n");
    printf("i User image I/O software\n");
    printf("x Stop tracking and exit\n");
    break;
}
} /* while */
} /* ENtry */
**ENcosmac_check()**

Routine checks COSMACs, first by determining their status (whether up, i.e., whether loaded), downloading the operating system if necessary, then downloading the sequel routine for reading out the CCDs. Finally, the COSMAC’s dac values are initialized.

```c
int i, response, load_flag;
char space[];

for (i = 0; i < NUM_COSMACS; i++)
{
    response = COroute_cosmac(i);
    switch (response)
    {
        case 0:
        case 2:
            printf("Please reset COSMAC #%d a few times\n", i);
            printf("\nHit any character and return when ready: ");
            fflush(stdout);
            scanf("%s", space);

            if (CODownload(i, 0) == 0)
            {
                CODownload(i, 1);
                CODownload(i, 2);
            }
            else
            {
                Cos_status[i] = -1;
                printf("COSMAC #%d not loading properly\n", i);
            }
            break;
        case 3:
            CODownload(i, 0);
            CODownload(i, 1);
            CODownload(i, 2);
            break;
        case 1:
            load_flag = COnext_load(i);
            if (load_flag == 3)
            {
                CODownload(i, 1);
                CODownload(i, 2);
            }
            if (load_flag == 0)
            {
                COSend_break(i);
                CODownload(i, 0);
                CODownload(i, 0);
                CODownload(i, 0);
            }
            break;
    }
}
```
} }
} /* ENcosmac_check */
ENtrigger_check()
{
    /*
    Routine loops through and checks the status of each trigger
    */
    int i;

    for(i=0;i<NUM_of_CAMERAS;i++) trigger_check(i);
} /* ENtrigger_check */

trigger_check(trigger)
int trigger;
{
    /*
    Routine prepares trigger for running trigger software by first checking
    whether trigger.bin is already loaded and running -- if not, the
    trigger is reset and trigger.bin is loaded and executed. If the trigger
    still does not respond to a null query, it is considered dead.
    */
    char status;

    if (Trig_status[trigger] != -1)  
    {
        status = TRnull_query(trigger);
        printf("Null query (%c) responds with %d\n",letters[trigger],status);
        if (status != 0)  /* then trigger.bin is not running */
        {
            printf("Trigger %c does not respond to a null query\n",letters[trigger]);
            TRreset_trigger(trigger);  /* put the trigger into ETRM */
            TRload_trigger(trigger);  /* attempt to download trigger.bin */
            TRgo_trigger(trigger);  /* execute trigger.bin */
            status = TRnull_query(trigger);
            printf("Null query responds with %d\n",status);
            if (status != 0)  /* then trigger is dead */
            {
                printf("Trigger %c is dead\n",letters[trigger]);
                Trig_status[trigger] = -1;
            }
        }
        else  
        {
            TRset_flash_subarray_size();
            TRset__standard_subarray_size();
        }
    }
} /* trigger_check */
ENopen_ports()
{
    /*
    Routine opens ports for timer on parallel board, all COSMACs and
    triggers as well as for the RMT and other external devices.
    Port settings for all COSMACs and triggers are set to raw, no echo.
    */

    int i;

    Cosmac_port[0] = open("/dev/tty7",2);
    Trig_port[0] = open("/dev/tty0",2);
    Trig_port[1] = open("/dev/tty4",2);
    /*Trig_port[2] = open("/dev/tty9",2);*/
    /*Trig_port[3] = open("/dev/tty10",2);*/
    Timer_port = open("/dev/tmr",0);
    RMT_port = open("/dev/null",2);

    for(i=0; i<NUM_COSMACS; i++)
    {
        if (Cosmac_port[i] != -1)
        {
            gtty(Cosmac_port[i],&our_port_settings);
            our_port_settings.sg_flags = (our_port_settings.sg_flags | RAW) & ~EC
            stty(Cosmac_port[i],&our_port_settings);
            Cos_status[i] = 1;
        }
    }

    for(i=0; i<NUM_OF_CAMERAS; i++)
    {
        if (Trig_port[i] != -1)
        {
            gtty(Trig_port[i],&our_port_settings);
            our_port_settings.sg_flags = (our_port_settings.sg_flags | RAW) & ~EC
            stty(Trig_port[i],&our_port_settings);
            Trig_status[i] = 1;
        }
        else Trig_status[i] = -1;
    }
} /* ENopen_ports */
This module allows the user direct interaction with the ETC camera fields. It will display the field of the current camera when read out. In addition, when requested, it will list the pixels in a subarray at any part of the chip, locate the centroid and give a FWHM. Exposure times and offset levels can be varied.

```
#include "constants.h"
#include "extern.h"
#include "ccphot.h"

#include <stdio.h>
#include <sgtty.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#include <sys/time.h>

#define SKY_SIZE 40 /* Used in calculation of sky noise */

int list_array_on = 0, FWHM_on = 0, show_array_on = 0; /* toggle switches */
int camera_number = 0, x_array = 0, y_array = 0; /* basic data */
int image_in_buffer = 0; /* image flag */
int high_pixel, x_centroid, y_centroid, array_median;
float x_fwhm, y_fwhm;
short array[SKY_SIZE];
unsigned char ucarray[SKY_SIZE];
unsigned char first_array[SKY_SIZE], second_array[SKY_SIZE];
int diff_array[SKY_SIZE];

extern JPDsnth();
extern TRsnd_entire_image(), TRsend_queries(), TRdc_current_frame();
extern TRrandom_array();

extern UTreadout_one_frame(), UTadjust_offsets(), UTsky_brightness();
extern UTslew_to(), UTra_from_ra(), UThour_angle(), UTra_of_ha();

extern COflush_CCDs(), CObtrack_clutch(), COset_offset();
extern unsigned char COsett_offset();
exern DOnmecheck();

extern double TMjd_time();

struct tm *localtime(), *gmtime();

Image()
{
    /* Routine is the node to all Image() routines. */

    int ra, flag = 1;
    char response[1], help;
}
T_data.sift_time = 90;
C0track_clutch(0);

while(flag)
{
    printf("\n\nImage I/O Routine: command? ");
    fflush(stdout);
    scanf("%ht",response);

    help = response[0];
    switch(help)
    {
    case 'c': /* change camera number */
        change_camera();
        break;
    case 'd': /* display image stored in trigger */
        disp();
        break;
    case 'D': /* display image stored in trigger */
        Disp();
        break;
    case 'n': /* input new exposure time */
        new_exposure_time();
        break;
    case 'r': /* read new image from CCDs */
        read_new_image();
        break;
    case 'R': /* read new image from CCDs */
        Read_new_image();
        break;
    case 'a': /* enter array location */
        get_array_location();
        break;
    case 't': /* modify toggle status */
        get_toggles(response);
        break;
    case 'o':
        UTadjust_offsets();
        break;
    case 'l':
        long_exposures();
        break;
    case 'b':
        offset();
        break;
    case 's':
        printf("The mount is at a right ascension of %d\n", 
            UTra_of_ra(UTha_from_ra(ra),0));
        printf("Enter ra to slew to: ");
        fflush(stdout);
        scanf("%d",&ra);
        UTslew_to(UTha_from_ra(ra),0);
        printf("The mount is at a right ascension of %d\n", 
            UTra_of_ra(UTha_from_ra(ra),0));
        break;
    case 'q':
noise();
break;
case 'h':
DOmecheck();
break;
case 'x':
flag = 0;
break;
default:
printf("c\tChange camera number\n");
printf("d\tDisplay image presently in trigger memory\n");
printf("b\tBrute-force display of image presently in trigger memory\n");
printf("n\tEnter new exposure time\n");
printf("r\tRead in new image\n");
printf("R\tRead in new image and display w/o frame_buf\n");
printf("a\tInput array location\n");
printf("o\tAutomatic bias level adjust\n");
printf("b\tAdjust individual bias levels\n");
printf("s\tSlew to any right ascension\n");
printf("q\tCalculate noise in a small subarray\n");
printf("t\tToggle FWHM display switch\n");
printf("s\tToggle show-array switch\n");
printf("l\tToggle list-array switch\n");
printf("l\tLong exposure images\n");
printf("x\tExit to entry level\n");
fflush(stdout);
bday;
}
]
}

change_camera()
[
/* Routine changes camera viewed */
printf("Present camera number is %d\nEnter new camera number: ",
camera_number);
fflush(stdout);
scafef("%d", &camera_number);
]

IMcheck_toggles()
[
/* Routine executes any subroutine indicated by a high toggle value */
if (!image_in_buffer) return;
collect_array();
if (list_array_on) list();
if (show_array_on) aed_array();
if (FWHM_on) get_fwhm();
]

new_exposure_time()
[ /* Routine changes the image exposure time */

    int value;
    printf("Present exposure time is \%d centiseconds\n", S_data.exposure_time);
    printf("Enter new exposure time in centiseconds: ");
    fflush(stdout);
    scanf("\%d", &value);
    if (value >= 100) S_data.exposure_time = value;
    else printf("Exposure time is too low\n");
]

read_new_image()
[
    /* Routine reads in and displays on the AED an image from a given camera:
    UTreadout_one_frame();
    disp();
]

Read_new_image()
[
    /*
    Routine is the same as read_new_image(), except it avoids frame_buf
    in the transfer of the CCD image to the AED
    */
    UTreadout_one_frame();
    Disp();
]

get_array_location()
[
    /* Routine allows input of array location */
    printf("Enter array location in pixels: ");
    fflush(stdout);
    scanf("\%d \%d", &x_array, &y_array);
    if((x_array<0) || (x_array>=NCOLUMNS) || (y_array<0) || (y_array>=NROWS))
        {
        printf("Bad chip location!!\n");
        return;
        }
    if(!image_in_buffer) return;
    surround_array();
    IMcheck_toggles();
]

collect_array()
[
    /* Routine grabs a subarray from the specified location out of frame_buf

int i,j,chan,counter=0;

chan = open("/dev/frame_buf",0);

for(j=y_array-SUBARRAY_SIZE/2;j<=y_array+SUBARRAY_SIZE/2;j++)
{
  lseek(chan,NCOLUMNS*j+x_array-SUBARRAY_SIZE/2,0);
  read(chan,&ucarray[counter*SUBARRAY_SIZE],SUBARRAY_SIZE);
  for(i=counter*SUBARRAY_SIZE;i<(counter+1)*SUBARRAY_SIZE;i++)
    array[i] = rom_mapping((int)ucarray[i]);
  counter++;
}
close(chan);
}

list()
{
  /* Routine lists a subarray from the specified location out of frame_buf.
   */
  int i,j;

  printf("\n\nArray located at (%d,%d)\n\n",x_array,y_array);
  j = SUBARRAY_SIZE;
  while(--j != -1)
  {
    for(i=0;i<SUBARRAY_SIZE;i++)
      printf("%d",array[j*SUBARRAY_SIZE+i]);
    printf("\n");
  }

  aprint(number)
int number;
{
  /* Generic number printing routine */

  printf("%d",number);
  if (number>999) printf(" ");
  if((number<1000) && (number>99)) printf(" ");
  if((number<100) && (number>9)) printf(" ");
  if (number<10) printf(" ");
}

aed_array()
{
  /* Routine displays specified subarray to AED */

  int i,j,aed;

  aed = open("/dev/aed",1);
  for(j=0;j<SUBARRAY_SIZE;j++)

{    
    lseek(aed, (350+j)*512,0);  
    write(aed, &ucarray[SUBARRAY_SIZE*j], SUBARRAY_SIZE);  
}  
close(aed);  
}  

aed_image()  
{  
/* Routine transfers a full image from frame_buf to the AED */  
    int j,aed,frame_buf;  
    unsigned char buf[NCOLUMNS];  
    aed = open("/dev/aed",1);  
    frame_buf = open("/dev/frame_buf",0);  
    for(j=0;j<NROWS;j++)  
    {  
        read(frame_buf,buf,NCOLUMNS);  
        lseek(aed,j*512,0);  
        write(aed,buf,NCOLUMNS);  
    }  
    close(aed);  
    close(frame_buf);  
}  

median_value()  
{  
/* Routine calculates the median value of a given subarray */  
    array_median = JP0snth(array,S_SQUARED,S_SQUARED/2);  
}  

get_fwhm()  
{  
/* Routine ostensibly returns the centroided value of the image in array.  
   This is done by 1) finding the highest pixel and  
   then performing a dotyfit centroiding operation on that pixel  
   */  
    int xh,yh;  
    high_pixel = IMget_high_pixel();  
    /* returns value in unit pixels */  
    yh = high_pixel/SUBARRAY_SIZE;  
    xh = high_pixel - yh*SUBARRAY_SIZE;  
    IMdotyfit(&xh,&yh);  
    /* returns values in centipixels */  
    if ((xh == -1) && (yh == -1)) x_centroid = y_centroid = -1;  
    else  
    {  
    }
IMget_hish_pixel()
{
    /*
    Routine returns high pixel from the array array
    */

    unsigned char high;
    int i,j,high_j,spot;
    high = 0;
    for(i=1;i<SUBARRAY_SIZE-1;i++)
    {
        for(j=1;j<SUBARRAY_SIZE-1;j++)
        {
            spot = SUBARRAY_SIZE*i+j;
            if (array[spot]>high)
            {
                high = array[spot];
                high_j = spot;
            }
        }
    }
    return(high_j);
} /* IMget_hish_pixel */

IMdotyfit(x,y)
int *x,*y;
{
    /*
    Routine returns the centroided location of a stellar image.
    Centroiding algorithm is a fit to a parabola, based on an
    algorithm developed by John Doty
    */

    int r,c,l,t,b;
    median_value();
    if ((*x == 0) || (*x == SUBARRAY_SIZE) || (*y == 0) || (*y == SUBARRAY_SIZE))
    {
        *x = 100 * (*x);
        *y = 100 * (*y);
    }
    else
    {
        r = IMvalue(*x+1,*y) - array_median;
        l = IMvalue(*x-1,*y) - array_median;
        c = IMvalue(*x,*y) - array_median;
        t = IMvalue(*x,*y+1) - array_median;
        x_centroid += xh - 100*(SUBARRAY_SIZE/2);
        y_centroid += yh - 100*(SUBARRAY_SIZE/2);
    } /* return_centroid */
}
b = IM_value((x),(y)-1) - array_median;
printf("%d,%d,%d,%d,%d\n",r,l,c,t,b);

if ((r+l==2*c) || (t+b==2*c)) *x = *y = -1;
else {
    *x = (int)(100.0 * (float)(*x) + 50.0 * ((float)(l-r)/(float)(r+l-2*c);  
    *y = (int)(100.0 * (float)(*y) + 50.0 * ((float)(b-t)/(float)(t+b-2*c;
}

/* now calculate the FWHM in both directions */
fwhm(r,l,c,t,b);
}

/* IMdotyfit */

IM_value(x,y)
int x,y;
{
    /* Routine returns value of subarray at location (x,y) */

    printf("v%d ",array[SUBARRAY_SIZE*y+x]);
    return((int)array[SUBARRAY_SIZE*y+x]);
} /* value */

fwhm(yr,yl,yc,yt,yb)
int yr,yl,yc,yt,yb;
{
    /* Routine returns FWHM of a stellar image. Presently not working */

    float a,b,c,x1,x2,y1,y2,xmax,ymax,insq;

    /* first, the x-direction */

    a = 0.5*(yr+yl-2.0*yc);
    b = 0.5*(yr-yl);
    c = yc;

    xmax = -0.5*b/a;
    ymax = a*xmax*xmax+b*xmax+c;

    insq = b*b-4*a*(c-ymax/2);
    printf("insq is %f\n",insq);

    x1 = (-b + sqrt(b*b-4*a*(c-ymax/2)))/2.0/a;
    x2 = (-b - sqrt(b*b-4*a*(c-ymax/2)))/2.0/a;

    x_fwhm = x1-x2;

    /* now, do y-direction */

    a = 0.5*(yt+yb-2.0*yc);
    b = 0.5*(yt-yb);
\[ \text{xmax} = -0.5*b/a; \]
\[ \text{ymax} = a*\text{xmax}*\text{xmax} + b*\text{xmax} + c; \]
\[ \text{insq} = b*b - 4*a*(c - \text{ymax}/2); \]
\[
\text{printf(}"\text{insq is } \%f\text{\n}\", \text{insq});
\]
\[ y1 = (-b + \sqrt{b*b - 4*a*(c - \text{ymax}/2)})/2.0/a; \]
\[ y2 = (-b - \sqrt{b*b - 4*a*(c - \text{ymax}/2)})/2.0/a; \]
\[ y_{fwhm} = y1 - y2; \]
\[
\text{printf(}"\text{X FWHM is } %.2f, \text{ Y FWHM is } %.2f\text{\n}, x_{fwhm}, y_{fwhm};\n\)
\]

\text{get_toggles(response)}

\text{char response[ ] ;}
[
\text{/* Routine allows for the input of various toggles */}

\text{if (strlen(response) != 2) return;}

\text{if (response[1] == }'l'\text{) }
[
\text{printf("Turning list array toggle ");}
\text{if (list_array_on)}
[
\text{printf("off\n\n");}
\text{fflush(stdout);}
\text{list_array_on = 0;}
\]
\text{else}
[
\text{printf("on\n\n");}
\text{fflush(stdout);}
\text{list_array_on = 1;}
\]
]

\text{if (response[1] == }'s'\text{) }
[
\text{printf("Turning show array toggle ");}
\text{if (show_array_on)}
[
\text{printf("off\n\n");}
\text{fflush(stdout);}
\text{show_array_on = 0;}
\]
\text{else}
[
\text{printf("on\n\n");}
\text{fflush(stdout);}
\text{show_array_on = 1;}
\]
]

\text{if (response[1] == }'f'\text{) }
[
}
printf("Turning FWHM toggle ");
if (FWHM_on)
{
    printf("off\n\n");
    fflush(stdout);
    FWHM_on = 0;
}
else
{
    printf("on\n\n");
    fflush(stdout);
    FWHM_on = 1;
}
}
}

disp()
{
    /* Routine gets full image from a Trigger and displays it on the AED */

    TRsnd_entire_image(cameranumber,0);
    image_in_buffer = 1;
    aed_image();
    if(!image_in_buffer) return;
    surround_array();
    I Mcheck_toggles();
}

Disp()
{
    /* Routine differs from disp() only in that it does not use frame_buf */

    int aed,nbytes,n_read,i,hssl_port;
    unsigned char buf[NCOLUMNS];
    char fourteen = 14;
    char zero = 0;

    hssl_port = open("/dev/hssl",0);
    aed = open("/dev/aed",1);

    write(Trig_port[cameranumber],&fourteen,1);
    write(Trig_port[cameranumber],&zero,1);

    for(i=0;i<NROWS;i++)
    {
        n_read = 0;
        while (n_read < NCOLUMNS)
        {
            nbytes = read(hssl_port,buf[n_read],NCOLUMNS-n_read);
            n_read += nbytes;
        }
        I seek(aed,512*i,0);
        write(aed,buf,NCOLUMNS);
    }

    close(hssl_port);
}
close(aed);

image_in_buffer = 1;

}

surround_array()
{
    /* Routine draws a circle around the array location */
    int i,x,y,aed;
    char color;

    aed = open("/dev/aed",1);
    color = 3;
    for(i=1;i<22;i++)
    {
        x = x_array + 8.0*cos(0.314159*i);
        y = y_array + 8.0*sin(0.314159*i);
        lseek(aed,y*512+x,0);
        write(aed,&color,1);
    }
    close(aed);
}

long_exposures()
{
    /*
     * Routine allows the acquisition and storage on disk of long time-exposur
     * from both cameras. Image data is stored in CCPHOT format, two bytes
     * per pixel, in de-mapped format.
     */

    int n_read,i,j,k,HSSL,file,count,nbytes,error,long_time;
    unsigned char buf[NCOLUMNS];
    double long_days,end,start;
    struct ccheader hd;
    struct tm *time_str;
    long seconds;
    short outbuf[NCOLUMNS];
    char response[14],filename[30];

    printf("Setting header variables\n");
    hd.nph = NCOLUMNS;
    hd.npv = NROWS;
    hd.p0 = hd.00 = 0;
    hd.l0act = hd.p0act = 0;
    hd.l1act = NROWS - 1;
    hd.p1act = NCOLUMNS - 1;
    hd.nbpx = 1;

    printf("Enter exposure time in seconds: ");
    fflush(stdout);
    scanf("%d",&long_time);
    fflush(stdout);
seconds = time(0);
ttime_str = gmtime(&seconds);

hd.idt[0] = (char)(ttime_str->tm_mon)+1;
hd.idt[1] = (char)(ttime_str->tm_mday);
hd.idt[2] = (char)(ttime_str->tm_year);

hd.itm[0] = (char)(ttime_str->tm_hour);
hd.itm[1] = (char)(ttime_str->tm_min);
hd.itm[2] = (char)(ttime_str->tm_sec);

seconds += long_time;
ttime_str = gmtime(&seconds);

hd.itm[0] = (char)(ttime_str->tm_hour);
hd.itm[1] = (char)(ttime_str->tm_min);
hd.itm[2] = (char)(ttime_str->tm_sec);

hd.texp = long_time;
hd.tmsec = 0;

long_days = (double)long_time/86400.0;
UTreadout_one_frame();

printf("Exposure starting\n");
TRdsc_current_frame();
TRend_queries();

start = TIMjd_time();
end = start + long_days;

while(TImjd_time() < end);
printf("Exposure ended\n");
fflush(stdout);
COreadout_CCDs();
error = IMwatch_triggers();
if (error) printf("There was an error #d in the readout\n",error);

for (i=0;i<NUM_of_CAMERAS;i++)
{
    UTsky_brightness(i);
    do
    {
        printf("Enter name of file for camera %c's image: ",letters[i]);
        fflush(stdout);
        scanf("%s",response);
        sprintf(filename,"data/%s",response);
        file = open(filename,1);
        if (file != -1) printf("File %s already exists!\n",filename);
    }
    while(file != -1);

    creat(filename,0777);
    file = open(filename,1);
C_data[i].right_ascension = UTra_of_ha(UThour_angle(0));
sprintf(hd.Icom,"ETC Camera %c, centered on (%.2fh,%.2fd)",letters[i],
C_data[i].right_ascension/261800.0,C_data[i].declination/17453.3
TRsnd_entire_image(i,0);
aed_image();

HSSL = open("/dev/frame_buf",0);
write(file,&hd,140);
for (j=0;j<NROWS;j++)
{
    read(HSSL,buf,NCOLUMNS);
    /*for (k=0;k<NCOLUMNS;k++) outbuf[k] = (short)rom_mapping[(int)buf[k]];
     */write(file,outbuf,2*NCOLUMNS);
}
close(file);
close(HSSL);

IMwatch_triggers()
[
    /*
    Routine watches all Triggers for sift-termination reports expected
    after the readout of long time-exposures.
    */
    int i,k,nbytes,done_processing,report[NUM_of_CAMERAS],count,flag,response
    unsigned char error[rep],r1,r2;
    short row;

    response = done_processing = 0;
    for (i=0;i<NUM_of_CAMERAS;i++)
    {
        if (Trig_status[i] == -1)
        {
            done_processing++;
            report[i] = 1;
        }
        else
        report[i] = 0;
    }

    flag = 0;
    while((done_processing<NUM_of_CAMERAS) && (flag == 0))
    {
        for(i=0;i<NUM_of_CAMERAS;i++)
        {
            if(report[i]==0)
            {
ioctl(Trig_port[i], FIONREAD, &count);
if (count!=0)
{
    for(k=0;k<MAX_TRIES;k++)
    {
        ioctl(Trig_port[i], FIONREAD, &count);
        if (count) break;
    }
    if (count == 0) error = 7;
    else nbytes = read(Trig_port[i], &rep, 1);
}
for(k=0;k<MAX_TRIES;k++)
{
    ioctl(Trig_port[i], FIONREAD, &count);
    if (count) break;
}
if (count == 0) error = 7;
else nbytes = read(Trig_port[i], &error, 1);
for(k=0;k<MAX_TRIES;k++)
{
    ioctl(Trig_port[i], FIONREAD, &count);
    if (count) break;
}
if (count == 0) error = 7;
else nbytes = read(Trig_port[i], &r1, 1);
for(k=0;k<MAX_TRIES;k++)
{
    ioctl(Trig_port[i], FIONREAD, &count);
    if (count) break;
}
if (count == 0) error = 7;
else nbytes = read(Trig_port[i], &r2, 1);
row = 256*(short)r1 + (short)r2;
if (rep != 64)
{
    printf("rep: %d  error: %d  row: %d\n", rep, error, row);
    consume_bytes(Trig_port[i]);
}
else
{
    printf("%c complete", letters[i]);
    if (error != 4) printf("", error = %d", error);
    printf("\n");
    fflush(stdout);
}
if ((error == 2) || (error == 6) || (rep != 64)) response = 1;
done_processing++;
report[i] = 1;
IMase.c Page

```c
int camerasoffsets
while(1) {
    printf("Which camera? (-1 to quit) ");
    scanf("%d","&camera);
    if (camera == -1) return;
    printf("Present offset is %d
",COget_offset(camera,0));
    printf("Enter new offset: ");
    scanf("%d","&offset);
    if (offset != -1) {
        COset_offset(camera,offset,0);
    }
}

noise() {
    int average, i, sum;
    float faverage, sigma, fsum;
    UTreadout_one_frame();
    TRrandom_array(camera_number,175,150,size,first_array);
    UTreadout_one_frame();
    TRrandom_array(camera_number,175,150,size,second_array);
    disp();
    sum = 0;
    i = size*size;
    while (--i != -1) sum += (int)first_array[i];
    average = (int)((float)sum/(float)(size*size))+0.5;
    i = size*size;
    while (--i != -1) diff_array[i] = (int)first_array[i] - (int)second_array[i];
    sum = 0;
    i = size*size;
    while (--i != -1) sum += diff_array[i];
    faverage = (float)sum/(float)(size*size);
    fsum = 0.0;
```
i = size*size;
while (--i != -1)
    fsum += ((double)diff_array[i]-faverage)*((double)diff_array[i]-faverage);
sigma = sqrt(fsum/(double)(2*size*size-1));
printf("Mean diff is %.2f CDU, average is %d, sigma is %.2f CDU\n",
    faverage, average, sigma);
}
/*
This module contains all the subroutines involved in actual observations with (by) the ETC. The flow of the observation software is described in the documentation.
*/

#include "constants.h"
#include "extern.h"
#include <stdio.h>
#include <setty.h>
#include <math.h>
#define centi(p) (int)(p/100.0 + 0.5)
#define micro(p) (double)(p/1000000.0)
#define LEFT 800
#define RIGHT 100*NCOLUMNS - 800
#define BOTTOM 800
#define TOP 100*NROWS - 800
#define DEFECT_BOX 500
#define PPARALLAX 876 /* two 50 mm pixels */
#define RA_LOW 814238 /* Define for one-degree box around */
#define RA_HIGH 843824 /* ostensible Aries Flasher location */
#define DEC_LOW 541925
#define DEC_HIGH 576831 /* Not implemented 11/85 */

/********************************************
DECLARE EXTERNAL ROUTINES */
/********************************************/
extern ENtrigger_check();
extern DOadjust_dome();
extern TRend_queries(), TRsnd_standard_subarrays(), TRassign_standard();
extern TRset_sift_time();
extern TIwatch_frame_timer();
extern double TIMjd_time();
extern COreadout_CCDs(), COtemperature();
extern ASget_coordinates();
extern PHotometry();
extern SAfill_SAO_structure();
extern UTinitialize_variables(), UTsetup_mounts(), UTget_median_frames();
extern UTdo_astrometry(), UTreadout_one_frame();
extern UTperipheral_check(), UTclear_active_entry();
extern JPmove();
extern WEclouds();
/***  DECLARE INTERNAL VARIABLES  /**/

int dead_triggers, report[NUM_of_CAMERAS], done_processing;
int raparallax[NUM_of_CAMERAS], exposure_status;
float too_long;
OBservations()

/*
Main observing routine for ETC. Contains the continuous loop of reading out CCDs, waiting for reports of flash candidates from triggers, storing data from flash candidates, if necessary, and requesting and storing data from flashes in progress.

Check_time() will end routine if time is 1) past sunrise, 2) past the time to slew the mounts to their starting positions or 3) at the time for a refresh of the astrometry parameters.

Check_weather() will end routine if weather threatens to be bad.

Check_peripherals will end routine if a peripheral is not functioning properly.

A final check of the time since last readout will determine whether the next exposure is a valid one.
*/

/* start by reading in the first frame of the run */

int itime_flag;
unsigned char temp[NUM_of_CAMERAS];

printf("Beginning observing sequence\n");
fflush(stdout);

ENtrigger_check();
dead_triggers = 0;
for(i=0;i<NUM_of_CAMERAS;i++)
{
    if (Trig_status[i]==-1)
    {
        report[i] = -1;
        dead_triggers++;
    }
    else report[i] = 1;
    Archives.good_exposure_counter[i] = 0;
}

COtemperature(0;temp);
for(i=0;i<NUM_of_CAMERAS;i++) /* only if #cameras <= 4 */
    C_data[i].temperature = temp[i];

UTinitialize_variables();
for (i=0;i<NUM_of_CAMERAS;i++)
    raparallax[i] = PPARALLAX/cos(micro(C_data[i].declination));
Total_number_of_exposures = 0;

if (UTreadout_one_frame() == -1)
    {
        Overseer_status = 7;
        return;
    }

Archives.start_time_of_run = TIMjd_time();
Dome_adjust_time = TIMjd_time() + S_data.dome_interval;

store_standards(0);  /* store pre-run standard arrays */
OBSERVED = 1;
too_long = 1.1*(float)S_data.exposure_time/100.0;
printf("Too long is %.2f\n", too_long);

/* start continuous observations */

while(Overseer_status == 1)
    {
        Trend_queries();
        TIMwatch_frame_timer();
        exposure_status = record_exposure_time();
        COreadout_CCDs();
        Total_number_of_exposures += 1;
        if (watch_for_trigger_reports() == 1) restart();
        else
            {
                if (Overseer_status != 1) /* then a trigger has died */
                    break;

            get_data_from_active_flashes();

        /* All check routines check to see whether the circumstances require an
         interruption of the current run -- if so, Overseer_status is set to
         the appropriate value and a 1 is returned. */

        check_time();  /* alters Overseer_status appropriately if some */
        /* time limit has been exceeded */

        /* weather_check();  

        UTperipheral_check();
        /* Sets Overseer_status to 4 when a peripheral dies */

        ioctl(Timer_port, ZERO_DETECT, &time_flag); /* in case the readout time
        if (time_flag == 1) restart();  /* was exceeded in get_data */
    }

/* store post-run standard arrays */
if (Overseer_status!=9) store_standards(1);

}  /* OBservations() */
watch_for_trigger_reports()
{
    /*
    Polls all active triggers for candidate and sift termination reports:
    candidates reported are stored in Active flash file
    */

    register short i;
    int count, time_flag, full_flag = 0;

    /* account for those triggers or cameras which are down */
    done_processing = dead_triggers;
    if (dead_triggers) printf("Number of dead triggers = %d\n", dead_triggers)
        for (i=0; i<NUM_of_CAMERAS; i++) if (report[i] != -1) report[i] = 0;

    /* start watching the triggers for reports */
    time_flag = 0;
    if (Num_stored_flashes == MAX_FLASHES_per_NIGHT) full_flag = 1;
    while ((done_processing != NUM_of_CAMERAS) && (time_flag == 0))
    {
        for (i=0; i<NUM_of_CAMERAS; i++)
        {
            ioctl(Trig_port[i], FIONREAD, &count);
            if (count != 0) read_and_file_trigger_report((int)i);
        }

        /* check to see whether the exposure time has run out, in which case
        one or more triggers are dead or slow */
        ioctl(Timer_port, ZERO_DETECT, &time_flag);
    } /* while */

    if (!full_flag)
    {
        for (i=0; i<NUM_of_CAMERAS; i++)
        {
            if ((report[i] == 0) && (report[Archives.counter_camera[i]] == 0))
                Archives.good_exposure_counter[i]++;
        }
        return(time_flag);
    } /* watch_for_trigger_reports */

read_and_file_trigger_report(trigger)
int trigger;
{
    /*
    Reads data from a trigger which has bytes on the I/O line.
    Returns a 1 if the trigger reports either a flash candidate or a sift termination; else, returns a 0
    */
int port, nbytes;
unsigned char token;

port = Trig_port[trigger];
nbytes = read(port, &token, 1);

if (token == 64) sift_termination_report(trigger);
else if (token == 128) candidate_report(trigger);
else printf("%c: %d\n", letters[trigger], token);
/* read_and_file_trigger_report */

sift_termination_report(trigger)
int trigger;
{
    /* Absorbs the data from a sift termination */

    int nbytes, port, count;
    char error_flag;
    unsigned char al, ah;
    short abort_row;

    port = Trig_port[trigger];
    nbytes = read(port, &error_flag, 1);

    ioctl(port, FIONREAD, &count);
    if (count == 0) report[trigger] = 7;
    else nbytes = read(port, &ah, 1);

    ioctl(port, FIONREAD, &count);
    if (count == 0) report[trigger] = 7;
    else nbytes = read(port, &al, 1);

    abort_row = (short)(ah << 8) + (short)al;

    if (report[trigger] != 7) report[trigger] = (int)error_flag;
    printf("s_t%c(%d;%d;%d)", letters[trigger], Num_of_active_entries,
    report[trigger], Num_stored_flashes);

    fflush(stdout);
    done_processing++;
} /* sift_termination_report */

candidate_report(trigger)
int trigger;
{
    /* Absorbs and stores the data from a flash candidate report */

    register int i;
    int k, nbytes, port, count;
unsigned char trigger_candidate_number, quality[3], xh, yh, xl, yl;
short x_candidate, y_candidate;

port = Trig_port[trigger];
nbytes = read(port,&trigger_candidate_number,1);

nbytes = read(port,&x_candidate,2);
if (nbytes == 1)
{
    nbytes = read(port,(char *)&x_candidate+1,1);
}

nbytes = read(port,&y_candidate,2);
if (nbytes == 1)
{
    nbytes = read(port,(char *)&y_candidate+1,1);
}

for(i=0;i<3;i++)
{
    ioctl(port,FIONREAD,&count);
    if (count != 0) nbytes = read(port,&quality[i],1);
}

write(port,&trigger_candidate_number,1); /* handshaking */
printf("nc%hc",letters[trigger]);

if (Num_stored_flashes == MAX_FLASHES_per_NIGHT) return;
if (exposure_status) return;
store_candidate_data(trigger,trigger_candidate_number,x_candidate,
y_candidate,quality);
} /* candidate_report */

store_candidate_data(trigger,number,x,y,quality)
int trigger;
unsigned char number; /**** DO SOMETHING WITH THIS ****/ 
short x,y;
unsigned char quality[3];
{

    Store data requested from a recently reported flash candidate
    in the active flash file for later scrutiny
*/

register short i;
int flag, xint, yint, ra, dec, delra, deldec, cam0, cam1;

convert_to_overseer_positions(x,y,&xint,&yint);

if (defect(xint,yint,trigger)) return;
if (((xint<LEFT) || (xint>RIGHT)) || (yint<BOTTOM) || (yint>TOP)) return;
ASget_coordinates(trigger,xint,yint,&ra,&dec);
if ((ra < RA_LOW) || (ra > RA_HIGH) || (dec < DEC_LOW) || (dec > DEC_HIGH))
{
    printf("R");
    return;
}

printf("(%d,%d;%d),%d,%d,quality[0]"),
flush(stdout);

flag = 0;
if (COINCIDENCE)
{
    for(i=0;i<Num_of_active_entries;i++)
    {
        register struct Per_active_entry_str *lff = &Live_flash_file[i];
        if (flag == 1) break;
        delra = abs(lff->Live_camera_entry[0].Flash_coordinates.x-ra);
        deldec = abs(lff->Live_camera_entry[0].Flash_coordinates.y-dec);
        cam0 = lff->Live_camera_entry[0].Camera_number;
        cam1 = lff->Live_camera_entry[1].Camera_number;
        if (cam1 == -1) /* otherwise it would at best be a brightened flash */
            if ((deldec < PPARALLAX) && (delra < raparallax[cam0]))
                /* Then we have a brightening detected */
                if (cam0 != trigger) /* Then we have a real coincidence detection */
                    flag = 1;
                    printf("COINCIDENCE! (%d,%d)
", ra, dec);
                    fflush(stdout);
                    prepare_active_entry(1, trigger, (int)i, ra, dec, xint, yint);
    }
}

if (flag==0) /* if no coincidence was found, or if not in coincidence mo */
{
    if ((Num_of_active_entries < MAX_NUM_of_ACTIVE_FLASHES) && (Num_stored_flashes < MAX_FLASHES_per_NIGHT))
    {
        prepare_active_entry(0, trigger, Num_of_active_entries, ra, dec, xint, yint,
        Num_of_active_entries++;
    }
} /* store_candidate_data */
prepare_active_entry(one_or_zero, trigger, entry_number, ra, dec, x, y)
int one_or_zero, trigger, entry_number, ra, dec, x, y;
{
    /*
    Actual storage routine for data from a flash candidate. Above data
    is stored, with others, in Live_flash_file
    */
    int flag, counter;
    register short i;
    register struct Active_per_camera_str *lfflce =
        &Live_flash_file[entry_number].Live_camera_entry[one_or_zero];

    /* first, find a trigger number */
    flag = 0;
    for (i = NUM_of_STANDARDS_per_FOV; i < 64; i++)
    {
        if ((Used_trigger_number[trigger*64+i] == 0) && (flag == 0))
        {
            lfflce->Trigger_flash_number = (unsigned char)i;
            Used_trigger_number[trigger*64+i] = 1;
            flag = 1;
        }
    }

    /* now, store passed data */
    {
        register struct Per_active_entry_str *lff = &Live_flash_file[entry_number]
            lfflce->Camera_number = trigger;
            lff->Update_time = Image_exposure_time.a;
            lff->Number_of_exposures = 0;
            lfflce->Flash_position.x = x;
            lfflce->Flash_position.y = y;
            lfflce->Flash_coordinates.x = ra;
            lfflce->Flash_coordinates.y = dec;
    }
    if (one_or_zero)
    {
        Live_flash_file[entry_number].Storage_number = Num_stored_flashes;
        Num_stored_flashes++;
    }
} /* prepare_active_entry */
get_data_from_active_flashes()
{
    /*
    Major routine in second phase of observations. All triggers have
    reported in or reported sick, so it is time to request data for
    all active flashes in active flash file and store it in the data
    storage structure

    Routine first sorts out solos in Active flash file (if COINCIDENCE is 1
    and then asks for and stores appropriate data in Data_storage
    */

    register short j,k,i;
    int cam,cam0,cam1,counter,storage_number,l;

    if (COINCIDENCE)
        {
            i = 0;
            while (i<Num_of_active_entries)
                {
                    if (Live_flash_file[i].Live_camera_entry[1].Camera_number == -1)
                        remove_active_entry((int)i);
                    else i++;
                }
            i = 0;
            while (i<Num_of_active_entries)
                {
                    register struct Per_active_entry_str *lff = &Live_flash_file[i];
                    if (lff->Number_of_exposures == 0)
                        {
                            cam0 = lff->Live_camera_entry[0].Camera_number;
                            cam1 = lff->Live_camera_entry[1].Camera_number;
                            if ((report[cam0]==2) || (report[cam0]==3) || (report[cam0]==6)
                                {
                                    storage_number = lff->Storage_number;
                                    for(l=0;l<Num_of_active_entries;l++)
                                        {
                                            if (Live_flash_file[l].Storage_number > storage_number)
                                                Live_flash_file[l].Storage_number--;
                                        }
                                    remove_active_entry((int)i);
                                    Num_stored_flashes--;
                                }
                            else
                                {
                                    for(k=0;k<=COINCIDENCE;k++)
                                        {
                                            cam = lff->Live_camera_entry[k].Camera_number;
                                            counter = 0;
                                            for(j=0;j<NUM_OF_STANDARDS_PER_FOV;j++)
                                                {
                                                    if (counter == NUM STANDARDS PER_FLASH) break;
                                                }
                                        }
                    }
                }
    }
if (SAO_file[cam].SAO_coordinates[j].x != 0) {
    lff->Stand_coordinates[counter] =
    SAO_file[cam].SAO_coordinates[j];
    lff->Live_camera_entry[k].Stand_position[counter] =
    SAO_file[cam].SAO_position[j];
    lff->Live_camera_entry[k].Stand_trigger_number[counter] =
    (unsigned char) j;
    counter++;
} /* if */
} /* for j */
} /* for k */
i++; /* else */
} /* if */
else i++;
} /* while */

for (i=0;i<Num_of_active_entries;i++)
{
    if (Num_stored_flashes <= MAX_FLASHES_per_NIGHT) {
        if (Live_flash_file[i].Number_of_exposures == 0) store_nth_data((int)i);
    }
    else if (Live_flash_file[i].Number_of_exposures > 0) store_nth_data((int)i);
}
store_first_data(number)
int number;
{
    /*
    Routine stores data from the flash immediately after discovery.
    This data includes data from the recent and old frames, as well
    as "discovery" data, such as where and when.
    Routine presently takes ~2 seconds per frame.
    */

    register short j,k;
    int i,storage_number,camera;
    int x,y,xi,yi;

    storage_number = Live_flash_file[number].Storage_number;

    register struct Per_flash_data_str *dssfd =
        &Data_storage[storage_number].Single_flash_data;

    for (j=0;j<=COINCIDENCE;j++)
    {
        register struct Active_camera_str *lfflce =
            &Live_flash_file[number].Live_camera_entry[j];

        camera = dssfd->Camera_number[j] = lfflce->Camera_number;
        dssfd->Flash_number[j] = lfflce->Trigger_flash_number;

        Data_storage[storage_number].Number_of_exposures = 1;
        dssfd->Flash_coordinates[j] = lfflce->Flash_coordinates;
        dssfd->Flash_position[j] = lfflce->Flash_position;
        x = dssfd->Flash_position[j].x;
        y = dssfd->Flash_position[j].y;

        xi = (int)(x/100.0+0.5);
        yi = (int)(y/100.0+0.5);
        TRassign_standard(camera,lfflce->Trigger_flash_number,xi,yi);

        JPDmove(&Row_threshold[camera*NCOLUMNS],&dssfd->Row_threshold[j*NCOLUMNS]);
        JPDmove(&Column_threshold[camera*NROWS],&dssfd->Column_threshold[j*NROWS]);
        TRsnd_standard_subarrays(camera,dssfd->Flash_number[j]);

        copy(1,dssfd->R_array[j].array);
        copy(2,dssfd->O_array[j].array);

        /*
        dssfd->Rec_photometry[j] = PHotometry(HSSL_array[1].array,x,y,camera);
        */
        /*
        dssfd->Old_photometry[j] = PHotometry(HSSL_array[2].array,x,y,camera);
        */

        for (k=0;k<NUM_STANDARDS_per_FLASH;k++)
        {
            
        }
dssfd->Stand_coordinates[k] = 
    Live_flash_file[number].Stand_coordinates[k];
dssfd->Stand_position[j*NUM_STANDARDS_per_FLASH+k] = 
    iflce->Stand_position[k];
dssfd->Stand_number[j*NUM_STANDARDS_per_FLASH+k] = 
    iflce->Stand_trigger_number[k];

TRsnd_standard_subarrays(camera,
    dssfd->Stand_number[j*NUM_STANDARDS_per_FLASH+k]);

} /* k */
} /* j */

Data_storage[storage_number].Number_of_exposures = 0;
} /* store_first_data */

store_nth_data(number)
int number;
{
    /* Routine stores image data subsequent to the initial detection. */

    register short j, k, number_of_exposure, storage_number, camera;
    int x, y;
    short photometry[COINCIDENCE+1];

    number_of_exposure = Live_flash_file[number].Number_of_exposures;
    storage_number = Live_flash_file[number].Storage_number;

    register struct Per_exposure_str *dssfdsed = &Data_storage[storage_number]
        Single_flash_data.Single_exposure_data[number_of_exposure];
    register struct Per_flash_data_str *dssfd = &Data_storage[storage_number].Single_flash_data;

    for (j=0; j<=COINCIDENCE; j++)
    {
        camera = Live_flash_file[number].Live_camera_entry[j].Camera_number;
            return;

        TRsnd_standard_subarrays((int)camera, dssfd->Flash_number[j]);
        copy(0, dssfdsed->Data_from_exposure[j].Flash_array.array);
        dssfdsed->Exposure_time = Image_exposure_time;

        /* Now, do photometry */
        x = dssfd->Flash_position[j].x;
        y = dssfd->Flash_position[j].y;
/** photometry[j] = dssfd->Data_from_exposure[j].Flash_photometry = Photometry(HSSL_array[0].array,x,y,(int)camera);*/

for(k=0;k<NUM_STANDARDS_per_FLASH;k++)
{
    TRsnd_standard_subarrays((int)camera, 
    dssfd->Stand_number[j*NUM_STANDARDS_per_FLASH+k]);
    copy(0,dssfd->Data_from_exposure[j].Stand_array[k].array);
}

Data_storage[storage_number].Number_of_exposures++; 
Live_flash_file[number].Number_of_exposures++; 

/!* now check whether flash lives! */  
if (Data_storage[storage_number].Number_of_exposures >= 
    MAX_EXPOSURES_per_FLASH)
{
    if ((Num_of_active_entries == 1) && 
        (Num_stored_flashes == MAX_FLASHES_per_NIGHT)) Overseer_status = 8; 
    /* break to store data */
    remove_active_entry(number);
}

/!* Supressing this section to speed up store_nth_data from 0.6sec/flash */
/!*else*
{
    for(j=0;j<COINCIDENCE;j++)
    {
        if ((photometry[j] - dssfd->Rec_photometry[j]) > C_data[camera].quiescent_difference)
        {
            Live_flash_file[number].Update_time = Image_exposure_time.a;
        }
    }
    if ((Live_flash_file[number].Update_time != Image_exposure_time.a) && 
        (Image_exposure_time.a - Live_flash_file[number].Update_time) > S_data.time_after_subsidal))
    {
        if ((Num_of_active_entries == 1) && 
            (Num_stored_flashes == MAX_FLASHES_per_NIGHT)) 
            Overseer_status = 8; /*/ /* break to store data *//*
        remove_active_entry(number);
    }
}/* store_nth_data */
copy(which_one,array)
int which_one;
unsigned char array[SUBARRAY_SIZE*SUBARRAY_SIZE];
{
    /* Routine simply copies subarrays */
    JPDmove(HSSL_array[which_one].array,array,S_SQUARED);
}

remove_active_entry(which_one)
int which_one;
{
    /* Routine removes all traces of an active entry from the Live_flash_file
     after the flash has been declared either dead or unconfirmed. */
    register short isk;
    int camera,fn;
    Num_of_active_entries--;
    for(k=0;k<COINCIDENCE;k++)
    {
        camera = Live_flash_file[which_one].Live_camera_entry[k].Camera_number;
        fn = Live_flash_file[which_one].Live_camera_entry[k].Trigger_flash_numb
        Used_trigger_number[64*camera+fn] = 0;
    }
    for(i=which_one;i<Num_of_active_entries;i++)
    {
        Live_flash_file[i] = Live_flash_file[i+1];
    }
    UTclear_active_entry(Num_of_active_entries);
    /* end */
}

check_time()
{
    /* Routine checks time to determine whether it is time to slew,
     close, or refresh astrometry. */
    double t;
    t = TImjd_time();
    if (t > AM_twilight) Overseer_status = 3;
else if (t > Slew_time) {
    if (Num_of_active_entries != 0) {
        if (t > Slew_time + PATIENCE) Overseer_status = 0;
        else Overseer_status = 0;
    }
}
else if (t > Astrometry_time) {
    Overseer_status = 6;
    if (((Num_of_active_entries != 0) && (t < Astrometry_time + PATIENCE))
        Overseer_status = 1;
    }
else if ((t > Dome_adjust_time) && (Num_stored_flashes < MAX_FLASHES_per_NIGHT)
    DOadjust_dome();
    Dome_adjust_time += S_data.dome_interval;
}

record_exposure_time() {
    /*
    Routine increments Image_exposure_time to reflect the start and end
    times of the last exposure made
    */
    float e_time;

    Image_exposure_time.a = Image_exposure_time.b;
    Image_exposure_time.b = Tlmjd_time();
    e_time = 86400.0*(Image_exposure_time.b-Image_exposure_time.a);
    printf("\nXt %.2f\n",e_time);
    if (e_time > too_long) return(1);
    else return(0);
} /* record_exposure_time */

convert_to_overseer_positions(x,y,xf,yf) short x,y;
int *xf,*yf; {
    /* Routine converts positions from format used by trigger to centipixels
    */
    *xf = x*1.5625;
    *yf = y*1.5625;
}
defect(x,y,camera) int x,y,camera; /* x,y in centipixels */ {
    /* Routine checks to see whether x,y are near a pixel or column defect */
    register short i;
    if (Col_defect_table[camera].defective_column[x/100]) return(1);
for (i=0; i<NUM_PIXEL_DEFECTS; i++)
{
    if ((abs(x-Pixel_defect_table[camera].location[i].x) < DEFECT_BOX)
        && (abs(y-Pixel_defect_table[camera].location[i].y) < DEFECT_BOX))
        return(1);
}

return(0);
}

restart()
{
/* Routine restarts observations cleanly after an exceeded readout time */

int i, j, count;
char buf[50];

printf("\nReadout time exceeded!\n");
fflush(stdout);
sleep(1);
count = 0;
for (i=0; i<NUM_of_CAMERAS; i++)
{
    for (j=0; j<MAXTRIES; j++)
    {
        ioctl(Trig_port[i], FIONREAD, &count);
        if (count != 0) break;
    }

    while(count != 0)
    {
        read(Trig_port[i], buf, 50);
        ioctl(Trig_port[i], FIONREAD, &count);
    }
}

if (COINCIDENCE)
{
    i = 0;
    while (i<Num_of_active_entries)
    {
        if (Live_flash_file[i].Live_camera_entry[1].Camera_number == -1)
            remove_active_entry(i);
        else i++;
    } /* while */
}

/* now restart the run */

TRset_sift_time();
UTset_frame_timer();
COflush_CCDs();
T1start_timer();
record_exposure_time();
TRdsc_current_frame();
}
store_standards(which)
int which;
{
    /* Routine stores standards at beginning or end of an observation cycle */
    int i,j;

    printf("\nStoring archival standards at observation's ");
    if (which) printf("end\n");
    else printf("start\n");
    for(j=0;j<NUM_of_CAMERAS;j++)
    {
        for(i=0;i<NUM_of_STANDARDS_per_FOV;i++)
        {
            TRsnd_standard_subarrays(j,i);
            if (which) copy(OArchives.standard_archives[J].sao_end_array[i].array);
            else copy(OArchives.standard_archives[J].sao_start_array[i].array);
            Archives.standard_archives[J].sao_coords[i] = SAO_file[J].SAO_coordinates[i];
            Archives.standard_archives[J].sao_pos[i] = SAO_file[J].SAO_position
            Archives.standard_archives[J].sao_mag[i] = SAO_file[J].SAO_magnitude
            Archives.standard_archives[J].sao_number[i] = SAO_file[J].SAO_star_number[i];
            Archives.standard_archives[J].sao_brightness[i] = SAO_file[J].SAO_brightness[i];
        }
    }
}

weather_check()
{
    /* Routine checks sky for clouds. Presently unimplemented */
    if (WEclouds()) printf("WEclouds claims the sky is cloudy\n");
}
```
#include "constants.h"
#include "extern.h"

/***
 DECLARE EXTERNAL ROUTINES ***/
extern DAYtime();
extern ENtrigger_check();
extern OBServations();
extern ASrough_astrometry();
extern double TIMjd_time();
extern TISet_frame_timer();

extern TRset_sift_time(), TRset_cosmic_sensitivity(), TRset_brightened_delta();
extern TRset_flash_subarray_size(), TRset__standard_subarray_size();

extern U Tinialize_variables(); U T idle_mounts();
extern U Tstore_flash_data(), U Tdo_ astrometry(), U Tweather_ check();
extern U Tcalculate_noise(), U Ttape_ storage();
```
operations()
{
    /*
     * Routine controls the flow of the overseer software by calling routines based on the value of Overseer_status. (see overseer_flow for more information). Overseer_status is set to 3 in entry routine.
     */
    printf("In operations() \n");
    UTinitialize_variables();
    while(1)
    {
        switch(Overseer_status)
        {
        case 0:
            /*
             * Make preparations for a tracking cycle. Assumes sky is clear and all systems are go, including that the dome is open
             */
            /*OBprepare_to_track();*/
            Overseer_status = 1;
            break;
        case 1:
            /*
             * Start main observing cycle
             */
            UTcalculate_noise();
            OBservations();
            UTstore_flash_data();
            UTinitialize_variables();
            break;
        case 2:
            /*
             * Invoked when weather or some peripheral indicates the weather is worsening. The tracking run in progress is considered over. Data is stored and the mounts are idled until the overseer feels it is safe to open again
             */
            UTidle_mounts();
            OPwait_until_clear();
            break;
        case 3:
            /*
             * Daytime and entry routine. Closes dome, idles mounts, store any data around. Waits until dusk breaks, whereupon the sky is checked and operations begin
             */
UTtape_storage();
DAytime();
break;
case 4:
    /*
     * Invoked in any panic mode. Shop is closed up and panic
     * procedures are implemented
     */
    UTidle_mounts();
    OPpanic();
    break;
case 5:
    /*
     * Exit case. Closes shop and exits to the entry routine
     */
    UTidle_mounts();
    exit(1);
case 6:
    /*
     * Pauses and recalculates the astrometry parameters
     */
    UTdo_astrometry();          /* NEED A NOISE CALCULATION ROUTINE
    Overseer_status = 1;
    break;
case 7:
    /*
     * Invoked from OBServations, if a trigger suddenly refuses
     * to report in. System is kept tracking, in the assumption
     * that the trigger needs to be reset
     */
    printf("Not all triggers reported in\n\n");
    ENtrigger_check();         /* now set up the triggers again */
    Overseer_status = 6;
    break;
case 8:
    Overseer_status = 1;
    break;
default:
    Overseer_status = 3;
    break;
} /* switch */
} /* while */
} /* operations */
OPsetup_operations()
{
    printf("This routine is presently out of order -- try the 'u' option\n")
}

OPwait_until_clear()
/*
Routine simply watches weather station reports for clear skies, checking continuously if day has broken in the meantime.
*/

int flag;

printf("In OPwait_until_clear()\n");
flag = 0;
while (flag == 0)
{
    if (UTweather_check() == 1)
    {
        if (TImjd_time() > Sunrise)
        {
            Overseer_status = 3;
            flag = 1;
        }
    }
    else
    {
        Overseer_status = 0;
        flag = 1;
    }
}
OPset_triggers()
{
    printf("In setup_triggers()\n");
    TRset_sift_time();
    TRset_cosmic_sensitivity();
    TRset_brightened_delta();
    TRset_flash_subarray_size();
    TRset_standard_subarray_size();
}

OPpanic()
{ }

OPpanic()
This module contains subroutines used in the semi-automatic, user-aided operation software. The implementation of this code is discussed elsewhere.

```c
#define micro(p) (double)p/1000000.0
#define MAX_DIFFERENCE 0.0
#define MAX_RA_DIFF 100000
#define AF_RA 829031
#define AF_DEC 559378

/*** DECLARE EXTERNAL ROUTINES ***/

extern ENtrigger_check();
extern UTinitialize_variables(),UTreadout_one_frame(),UTdetermine_threshold;
extern UTcalculate_noise(),do_astrology(),UTsky_brightness();UTdo_astrology;
extern UTslew_to(),UTha_from_ra(),UThour_angle();UTra_of_ha();
extern UTadjust_offsets(),UTfind_biasses();UTget_median_frames();
extern UTstore_flash_data();UTsave_frames();UTdirty_frame();
extern TRsnd_entire_image();TRset_cosmic_sensitivity();TRget_median_data();
extern TRsnd_standard_subarrays();
extern TRset_brightened_delta();
extern COSlew_clutch();
extern ASrough_astrology(),ASget_position();
extern SAget_stars(),SAassign_standards(),SAauto_lineup();
extern OBServations();
extern DOadjust_dome(),DOinitialize_dome();
extern Tlget_rise_and_set_times();
extern double TImjd_time();
extern DECLarations();
extern DAtetime();
extern WEclouds();
int report[NUM_of_CAMERAS],automatic_astrology_flag;
```
USer()
{
    /* Routine is node for user software */

    int i,bias,time,offset,flag;
    char response[];

    Declarations();
    announce_time();
    Daytime();

    while (USAdjust_mount() == 0)
    {
        user_queries();

        announce_time();

        if (!SQRT_ROM)
        {
            printf("\n\nAdjusting the offsets of the cameras so that the\n");
            printf("sky level is in the linear range\n\n");
            UTadjust_offsets();
        }

        announce_time();
        UStime_check();
        if (Overseer_status == 3)
        {
            CoSlew_clutch(0);
            return;
        }

        flag = 1;
        while (flag != 0)
        {
            announce_time();
            printf("Save frames? ");
            fflush(stdout);
            scanf("%s",response);
            if (response[0] == 'y')
            {
                UTreadout_one_frame();
                for(i=0;i<NUM_of_CAMERAS;i++)
                {
                    printf("Displaying field of view of camera %c\n",letters[i]);
                    fflush(stdout);
                    TRsnd_entire_image(i,0);
                    display_from_frame_buf();
                    sleep(2);
                }
                flag = 2;
                UTsave_frames();
            }
            else flag = 0;
        }
    }
}
announce_time();
TRsnd_entire_image(0,0);
display_from_frame_buf();
printf(“Enter a character and hit return when the dome slit is\n”);
printf(“properly aligned with the cameras\n”);
fflush(stdout);
scanf(“%s”,response);
DOinitialize_dome();

announce_time();
if (!SQRT_ROM)
{
    UTfind_biasses();

    printf(“Calculating total noise in each camera\n”);
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        UTcalculate_noise(i);
        Sky_brightness[i] = Global_brightness;
    }
}

UTdetermine_thresholds();

announce_time();
sao();

announce_time();
UTget_median_frames();

announce_time();
printf(“\nSetting the necessary trigger parameters\n”);
TRset_brightened_delta();
TRset_cosmic_sensitivity();

announce_time();
observe();
if (Overseer_status == 3) DAYtime();
} 
COSlew_clutch(0);
} /* USer */
user_queries()
{
    /* Routine fields responses from user to questions regarding operations */
    int i, time, offset;
    char response[];

    printf("The present exposure time is %d\n", S_data.exposure_time);
    printf("Enter new exposure time (0 for no change): ");
    fflush(stdout);
    scanf("%d",&time);
    if (time != 0) S_data.exposure_time = (short)time;
    printf("\nThe present exposure time is %d\n", S_data.exposure_time);

    printf("The present sift time is %d\n", T_data.sift_time);
    printf("Enter new sift time (0 for no change): ");
    fflush(stdout);
    scanf("%d",&time);
    if (time != 0) T_data.sift_time = (short)time;
    printf("\nThe present sift time is %d\n", T_data.sift_time);

    automatic_astrometry_flag = 1;
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        printf("Present number of sigma used for threshold offset %c is %d\n", \\
               letters[i], C_data[i].threshold_offset);
        printf("Enter new number of sigma (0 for no change): ");
        fflush(stdout);
        scanf("%d", &offset);
        if (offset != 0) C_data[i].threshold_offset = (unsigned char) offset;
    }

    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        printf("Present number of sigma used for brightened delta %c is %d\n", \\
               letters[i], C_data[i].brightened_delta);
        printf("Enter new number of sigma (0 for no change): ");
        fflush(stdout);
        scanf("%d", &offset);
        if (offset != 0) C_data[i].brightened_delta = (unsigned char) offset;
    }
}

USAdjust_moun(t())
{
    /*
    Routine allows for iterative adjustment of tracking mount to any
    hour angle
    */

    int ha, ra, flag;
    char response[];
flag = 1;
while(flag == 1)
{
    printf("You are now at a right ascension of %d\n", 
            UTra_of_ha(UTHour_angle(0)));
    printf("Acceptable? ");
    fflush(stdout);
    scanf("%s",response);
    if (response[0] == 'y') flag = 0;
    else if (response[0] == 'x') return(1);
    else if (response[0] == 'a')
    {
        UTslew_to(UTha_from_ra(AF_RA),0);
        flag = 1;
    }
    else if (response[0] == 's')
    {
        printf("Enter right ascension to slew to: ");
        fflush(stdout);
        scanf("%d",&ra);
        UTslew_to(UTha_from_ra(ra),0);
        flag = 1;
    }
    else
    {
        printf("y to continue\na to slew to Aries Flasher\n");
        printf("s to slew to any RA\nn to report the RA of the mount\n");
        printf("x to abort\n");
        flag = 1;
    }
}
return(0);

UTime_check()
{
    /* Routine checks time to see whether we should be observing at all */
    int hours,minutes,ha,diff;
    char response[];
    double time;

    time = TImjd_time();

    if (TIget_rise_and_set_times() == 3)
    {
        printf("It is still daytime. Should we continue? ");
        fflush(stdout);
        scanf("%s",response);
        if (response[0] != 'y')
        {
            Overseer_status = 3;
            return;
        }
    }
ha = UThour_angle(0);

if ((MAX_HOURANGLE - ha) > (Sunrise - time)*TWO_PIe6)
    diff = (Sunrise - time)*TWO_PIe6;
else diff = MAX_HOURANGLE - ha;

minutes = diff/4363.3;

hours = minutes/60;
minutes -= 60*hours;

printf("The observations will end in %d hours, %d minutes\n",hours,minutes);
Slew_time = TImjd_time() + (double)diff/6283185.3;

sao()
[
    /* Routine allows for manual and automatic alignment of the CCD fields */
    int i,ra,aed;
    ra = UTra_of_ha(UThour_angle(0));
    printf("Preparing to align cameras with the sky\n\n");
    C_data[0].declination = 2.0165*COsynchro_angle(0) + 516100;
    C_data[1].declination = -1.8818*COsynchro_angle(0) + 560784;
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        C_data[i].right_ascension = ra + C_data[i].ra_difference;
        ASrough_astrometry(i);
        SAgamers(i);
        TRsnd_entire_image(i,0);
        display_from_frame_buf();
        plot_standards(i);
        aed = open("/dev/aed",1);
        circle_standards(i,aed);
        close(aed);
    }

    if (SAauto_lineup()) manual_astrometry(0);
    else
    {
        UTdirty_frame();
        for(i=0;i<NUM_of_CAMERAS;i++)
        {
            TRsnd_entire_image(i,0);
            display_from_frame_buf();
            plot_standards(i);
            aed = open("/dev/aed",1);
            circle_standards(i,aed);
            close(aed);
        }
    }
]
observe()
{
    /*
     * Routine begins a semi-automatic observation sequence.
     * Routine assumes all relevant parameters are preset.
     */

    int i, minutes, readout_time, sky, flag, xaf, yaf, aed;
    printf("\n\nBeginning observation cycle\n\n");
    Overseer_status = 1;
    WEclouds();
    flag = 0;
    while ((Overseer_status != 0) && (Overseer_status != 3))
    {
        DOadjust_dome();
        if (Overseer_status != 8)
        {
            announce_time();
            printf("\nChecking for dome occultation...\n");
            UTdirty_frame();
            for (i = 0; i < NUM_of_CAMERAS; i++)
            {
                TRsnd_entire_image(i, 0);
                display_from_frame_buf();
                if (i == 0)
                {
                    printf("\nYOU HAVE 5 SECONDS TO MAKE A JUDGEMENT\n");
                    sleep(5);
                }
            }
        }
    }
    aed = open("/dev/aed", 1);
    ASget_position(1, AF_RA, AF_DEC, &xaf, &yaf);
    draw_circle(xaf, yaf, 60, aed);
    close(aed);
    Dome_adjust_time = Timjd_time() + S_data.dome_interval;
    Overseer_status = 1;
    if (flag == 0) Astrometry_time = Timjd_time() + S_data.astrometry_interval;
    announce_time();
    OBServations();
    flag = 0;
    announce_time();
UTstore_flash_data();
if (Overseer_status == 8) flag = 1;

if((Overseer_status == 3) || (Overseer_status == 0)) break;

UTinitialize_variables();
if(Overseer_status != 8)
{
    DOadjust_dome();
    if (!SQRT_ROM)
    {
        if (UTadjust_offsets() == 1)
        {
            UTfind_biasses();
            UTreadout_one_frame();
        }
    }

    announce_time();
    if (!SQRT_ROM)
    {
        for(i=0;i<NUM_of_CAMERAS;i++) UTcalculate_noise(i);
        UTdetermine_thresholds();
    }

    announce_time();
    UTget_median_frames();

    if (!SQRT_ROM)
    {
        for(i=0;i<NUM_of_CAMERAS;i++)
        {
            UTsky_brightness(i);
            Sky_brightness[i] = Global_brightness;
        }
    }
}

if (Overseer_status == 6)
{
    printf("\nRefreshing the astrometry of all the cameras\n\n");
    manual_astrometry(automatic_astrometry_flag);
    DOadjust_dome();
    Overseer_status = 1;
}

if (Overseer_status == 7)
{
    ENtrigger_check();
    Overseer_status = 1;
}
} /* while */
C0slew_clutch(0);
display_from_frame_buf()
{
    /* Routine displays image in frame_buf to AED */

    int chan,aed,j,i,n_read,nbytes;
    unsigned char buf[NCOLUMNS];

    aed = open("/dev/aed",1);
    chan = open("/dev/frame_buf",0);

    for (i=0;i<NROWS;i++)
    {
        n_read = 0;
        while (n_read<NCOLUMNS)
        {
            nbytes = read(chan,&buf[n_read],NCOLUMNS-n_read);
            n_read += nbytes;
        }

        lseek(aed,512*(i+5)+50,0);
        write(aed,buf,NCOLUMNS);
    }

    close(chan);
    close(aed);
    } /* display_from_frame_buf */

plot_subarray(number)
int number;
{
    /* Routine displays image subarrays to AED */

    int i,j,k,aed,diff;
    unsigned char foo[SUBARRAY_SIZE*SUBARRAY_SIZE];

    aed = open("/dev/aed",1);

    for(j=0;j<number;j++)
    {
        for (i=0;i<SUBARRAY_SIZE;i++)
        {
            lseek(aed,512*(400+i)+50+20*j,0);
            write(aed,&HSSL_array[j].array[SUBARRAY_SIZE*i],SUBARRAY_SIZE);
        }
    }

    close(aed);
}

plot_standards(camera)
int camera;
{
    /* Routine displays standard subarrays to AED */

    int i,j,aed;
unsigned char number, buf[SUBARRAY_SIZE];

eaed = open("/dev/aed", 1);

for (i = 0; i < SUBARRAY_SIZE; i++) buf[i] = 0;

for (i = 0; i < NUM_of_STANDARDS_per_FOV; i++)
{
    if (Used_trigger_number[camera*64+i] == 1)
    {
        number = (unsigned char) i;
        TRsnd_standard_subarrays(camera, number);

        for (j = 0; j < SUBARRAY_SIZE; j++)
        {
            Iseek(aed, 512*(380+j)+50+10*i, 0);
            write(aed, &HSSL_array[0].array[SUBARRAY_SIZE*j], SUBARRAY_SIZE);
        }
    }
    else
    {
        for (j = 0; j < SUBARRAY_SIZE; j++)
        {
            Iseek(aed, 512*(380+j)+50+10*i, 0);
            write(aed, buf, SUBARRAY_SIZE);
        }
    }
}

close(aed);

circle_standards(camera, aed)
int camera;
{
    /* Routine circles SAO standards stars on AED */

    int i, x, y, radius;

    for (i = 0; i < NUM_of_STANDARDS_per_FOV; i++)
    {
        if (SAO_file[camera].SAO_star_number[i] != 0)
        {
            ASget_position(camera, SAO_file[camera].SAO_coordinates[i].x,
                           SAO_file[camera].SAO_coordinates[i].y, &x, &y);
            radius = 10 - SAO_file[camera].SAO_magnitude[i]/10;
            draw_circle(x, y, radius, aed);
        }
    }
}

draw_circle(x, y, size, aed)
int x, y, size, aed;
{
    /* Generic circle-drawing routine */
int i, xf, yf;
char fifteen;
double phase;

if ((x>51200) || (y>51200) || (x<0) || (y<0)) return;
fifteen = 3;
for(i=1; i<22; i++)
{
    phase = 0.314159*i;
    xf = 50 + x/100 + (int)(size*cos(phase));
    yf = 50 + y/100 + (int)(size*sin(phase));
    lseek(aed, (long)(yf*512+xf), 0);
    write(aed, &fifteen, 1);
}

if (size<20) return;

for(i=1; i<220; i++)
{
    phase = 0.0314159*i;
    xf = 50 + x/100 + (int)(size*cos(phase));
    yf = 50 + y/100 + (int)(size*sin(phase));
    lseek(aed, (long)(yf*512+xf), 0);
    write(aed, &fifteen, 1);
}

clear_aed()
{
/* Routine clears AED screen */

    int i, aed;
    char buf[512];

    for (i=0; i<512; i++) buf[i] = 0;
    aed = open("/dev/aed", O);
    for (i=0; i<512; i++) write(aed, buf, 512);
    close(aed);
}

manual_astrometry(auto_flag)
int auto_flag;
{
    /*
       Routine does astrometry on all fields with present astrometric
       parameters.
       Improvement of (ra,dec) queries suppressed when auto_flag = 1
       */

    int i, j, aed, x, y, x_offset, y_offset, ra_c, dec_c, error, ra_offset, dec_offset;
    double cos_dec, sin_angle, cos_angle;

    UTdirty_frame();
    for (i=0; i<NUM_of_CAMERAS; i++)
cos_dec = \cos((\text{micro}(C\_data[i].\text{declination})))
\cos_angle = \cos((\text{micro}(C\_data[i].\text{ccd}\_angle)))
\sin_angle = \sin((\text{micro}(C\_data[i].\text{ccd}\_angle)))
clear\_aed()

printf("\nDisplaying field-of-view of camera \%c\n\n", letters[i]);
TRsnd\_entire\_image(i,0);
display\_from\_frame\_buf();

error = 1;
while (error == 1)
{
    error = 0;
    x\_offset = y\_offset = 0;
    while((x\_offset != 0) && (y\_offset != 0))
    {
        aed = open("/dev/aed","1");
        printf("\nCircling expected locations of SAO stars on AED\n\n");
        ASget\_position(i,AF\_RA,AF\_DEC,&x,&y);
draw\_circle(x,y,60,aed);
SAassign\_standards(i); close(aed);
plot\_standards(i);
aed = open("/dev/aed","1");
circle\_standards(i,aed); close(aed);
if (auto\_flag) x\_offset = y\_offset = 0;
else
{
    printf("\nEnter amount to move circles to find stars (in pixels) \n";
    printf("Enter x first, then y, separated by a space\n");
    printf("Enter '0 0' if close enough to do astrometry: ");
    fflush(stdout);
    scanf("%d %d","&x\_offset,"&y\_offset);

display\_from\_frame\_buf();
aed = open("/dev/aed","1");
ra\_offset = (x\_offset*cos\_angle + y\_offset*sin\_angle)/cos\_dec;
dec\_offset = y\_offset*cos\_angle - x\_offset*sin\_angle;
C\_data[i].right\_ascension -= C\_data[i].radians\_per\_pixel*ra\_offse
C\_data[i].declination -= C\_data[i].radians\_per\_pixel*dec\_offset;
}
close(aed);
}

printf("\n\nAttempting astrometry on field displayed\n\n");
error = do_astrometry(i);
if (error) auto_flag = 0;
if (error) printf("\nPleas attempt to line up fields again
\n"");
}
}

announce_time()
{
/* Routine announces present time to user */

int min,hour;
double time;

time = Ti_mjd_time();

time = time - (double)(int)time;
min = time * 1440.0;
hour = min/60;
min -= hour*60;
if (min < 10) printf("\nThe time is %d:0%d UT\n",hour,min);
else printf("\nThe time is %d:%d UT\n",hour,min);
}
This module contains all subroutines which are used in the calculation of the parameters of the astrometric fit in the ETC CCDs. The subroutines were developed by Charles Lawrence for plate-fitting routines. They were adapted for use with the ETC.

#include "constants.h"
#include "extern.h"
#include <math.h>

#define micro(p) ((double)p/1000000.0)
#define mega(p) ((double)p*1000000.0)
#define DEGREES_PER_RADIAN 180/3.1415926536
#define MINIMUM_NUMBER_of_STARS 6

/*** DECLARE EXTERNAL ROUTINE /***/
extern SAassign_standards();

/*** DECLARE INTERNAL GLOBALS (LOCAL GLOBALS?) /***/
double dx[50],dy[50],dxi[50],deta[50];
/* 'AStrometry.c' ROUTINES */

ASrough_astrometry(camera)
int camera;
{
    /* Routine sets up rough astrometry parameters based on a linear fit and the solid angle subtended by each pixel, which depends on the focal length of the lens, the size of the pixel and which is aided by the use of a fudge factor which will take out the uncertainty (< 5%) in the focal length of the lens */

    int i, r_p_p;
    double a1, a2, b1, c1, b2, c2, beta1, beta2, gamma1, gamma2, cos_angle, sin_angle;
    float fudge;
    register struct C_option_str *cdc = &C_data[camera];

    fudge = cdc->fudge_factor;
    cos_angle = cos(micro((double)cdc->ccd_angle));
    sin_angle = sin(micro((double)cdc->ccd_angle));
    r_p_p = cdc->radians_per_pixel;

    for(i=0; i<6; i++)
    {
        cdc->p_to_c_parameters[i] = 0.0;
        cdc->c_to_p_parameters[i] = 0.0;
    }

    b1 = cdc->p_to_c_parameters[1] = 0.01 * r_p_p * cos_angle;
    c1 = cdc->p_to_c_parameters[2] = 0.01 * r_p_p * sin_angle;
    b2 = cdc->p_to_c_parameters[4] = -0.01 * r_p_p * sin_angle;
    c2 = cdc->p_to_c_parameters[5] = 0.01 * r_p_p * cos_angle;
    a1 = cdc->p_to_c_parameters[0] = -b1*NCOLUMNS*50 - c1*NROWS*50;
    a2 = cdc->p_to_c_parameters[3] = -b2*NCOLUMNS*50 - c2*NROWS*50;

    beta1 = cdc->c_to_p_parameters[1] = fudge*c2/(c2*b1-b2*c1);
    gamma1 = cdc->c_to_p_parameters[2] = -fudge*c1/(c2*b1-b2*c1);
    beta2 = cdc->c_to_p_parameters[4] = fudge*b2/(c1*b2-b1*c2);
    gamma2 = cdc->c_to_p_parameters[5] = -fudge*b1/(c1*b2-b1*c2);
    cdc->c_to_p_parameters[0] = -beta1*a1-gamma1*a2;
    cdc->c_to_p_parameters[3] = -beta2*a1-gamma2*a2;
} /* ASrough_astrometry */
ASget_coordinates(camerax, y, ra, dec)
int camerax, y;
int *ra, *dec;
{
    /*
     Routine returns (ra, dec) corresponding to an (x, y) in a given
     camera, based on the parameters found in C_data[camera]
     */

    register struct C_option_str *cd = &C_data[camera];

    prdsq(x, y, cd->right_ascension, cd->declination, cd->p_to_c_parameters, ra, dec);
    if (*ra > TWO_PI6) (*ra) -= TWO_PI6;
    if (*ra < 0) (*ra) += TWO_PI6;
}

ASget_position(camerara, dec, x, y)
int camerara, dec, *x, *y;
{
    /*
     Routine returns (x, y) corresponding to an (ra, dec) in a given
     camera, based on the parameters found in C_data[camera]
     */

    register struct C_option_str *cd = &C_data[camera];
    pxysq(cd->right_ascension, cd->declination, ra, dec, cd->c_to_p_parameters, x, y);
Routine calculates the precise astrometric parameters for a given camera from a linear least-squares fit of the \((x, y)\) location of SAO stellar images on a CCD to their reduced coordinates. The fit process is iterative, throwing out those stars with residuals to the fit exceeding a programmed value.

Routine returns 0 if the rms error of the fit is higher than a programmed value AND the number of stars used in the fit exceeds a programmed value.

Routine returns 1 if this does not apply.

```
register short icounter;
int RA,DEC,flag,old,max,j,diffradifffdec,maxnum;
int x[NUM_of_STANDARDS_per_FOV],y[NUM_of_STANDARDS_per_FOV];
int ra[NUM_of_STANDARDS_per_FOV],dec[NUM_of_STANDARDS_per_FOV];
double xi[NUM_of_STANDARDS_per_FOV],eta[NUM_of_STANDARDS_per_FOV];
long diff,maxdiff,maxsquared;
register struct C_option_str *cdc = &C_data[camera];
counter = 0;
for(i=0;i<NUM_of_STANDARDS_per_FOV;i++)
{
    register struct Astr_data_str *adi = &Ast_data[i];
    if((adi->ra!=0) && (adi->dec!=0))
    {
        ra[counter] = adi->ra;
        dec[counter] = adi->dec;
        x[counter] = adi->x;
        y[counter] = adi->y;
        radxin(cdc->right_ascension,cdc->declination, ra[counter],dec[counter],&xi[counter],&eta[counter]);
        counter++;
    }
}
/* Begin iterative fitting process */

flag = 0;
max = cdc->radians_per_pixel;
maxsquared = (long)max*max/4.0;
while (counter >= MINIMUM_NUMBER_of_STARS)
{
    fit_xy((int)counter,x,y,xi,eta,cdc->p_to_c_parameters);
    fit_xieta((int)counter,xi,eta,x,y,cdc->c_to_p_parameters);
    old = counter;
    maxdiff = 0;
    maxnum = -1;
    for(i=0;i<counter;i++)
    {
```
ASget_coordinates(camera, x[i], y[i], &RA, &DEC);

if (diff > maxsquared)
{
  if (diff > maxdiff)
  {
    maxdiff = diff;
    maxnum = i;
  }
}

if (maxdiff != 0)
{
  printf("Throwing #%d out of linear fit\n", maxnum);
  counter--;
  for (j = maxnum; j < counter; j++)
  {
    ra[j] = ra[j+1];
    dec[j] = dec[j+1];
    xi[j] = xi[j+1];
    eta[j] = eta[j+1];
    x[j] = x[j+1];
    y[j] = y[j+1];
  }
  if (old == counter) break;
}

if (counter < MINIMUM_NUMBER_of_STARS) flag = 1;
return(flag);
ASrecenter(camera)
int camera;
{
    /*
     * Routine uses the calculated fit parameters to calculate the actual
     * (ra, dec) of the center of a CCD, so that IF ASrough_astrometry
     * need be invoked, the rough astrometry will be sufficient to
     * find all the SAO stars in the field.
     */

double cos_dec, cos_angle, sin_angle;
float x, y, delra, deldec, x, y;
register struct C_option_str *cdc = &C_data[camera];

cos_dec = cos(micro(cdc->declination));
cos_angle = cos(micro(cdc->ccd_angle));
sin_angle = sin(micro(cdc->ccd_angle));

x = NCOLUMNS*0.5*cdc->fudge_factor;
y = NROWS*0.5*cdc->fudge_factor;

x = cdc->c_to_p_parameters[0]/100.0;
y = cdc->c_to_p_parameters[3]/100.0;

delra = -(x-xc)*cos_angle - (y-yc)*sin_angle;
deldec = -(y-yc)*cos_angle + (x-xc)*sin_angle;

cdc->right_ascension += delra*cdc->radians_per_pixel/cos_dec;
cdc->declination += deldec*cdc->radians_per_pixel/cos_dec;

cdc->c_to_p_parameters[0] = 100.0*xc;
cdc->c_to_p_parameters[3] = 100.0*yc;

SAassign_standards(camera);
} /* ASrecenter */
/*************************************/
/* AStrometry.c SUBROUTINES */
/*********************/

fit_xy(number, x, y, xi, eta, fitpar)
int number;
int x[], y[];
double xi[], eta[];
double fitpar[6];
{
    /*
    Routine calculates parameters of linear fit of (xi,eta) to (x,y)
    (i.e. (xi,eta) = f(x,y))
    */
    register short i = number;

    while (--i != -1)
    {
        dx[i] = (double)x[i];
        dy[i] = (double)y[i];
        dxi[i] = xi[i];
        deta[i] = eta[i];
    }

    linear_fit(number, dx, dy, dxi, deta, fitpar);
} /* fit_xy */

fit_xieta(number, xi, eta, x, y, fitpar)
int number;
double xi[], eta[];
int x[], y[];
double fitpar[6];
{
    /*
    Routine calculates parameters of linear fit of (x,y) to (xi,eta)
    (i.e. (x,y) = f(xi,eta))
    */
    register short i = number;

    while (--i != -1)
    {
        dx[i] = (double)x[i];
        dy[i] = (double)y[i];
        dxi[i] = xi[i];
        deta[i] = eta[i];
    }

    linear_fit(number, dxi, deta, dx, dy, fitpar);
} /* fit_xieta */

linear_fit(number, x, y, xi, eta, fitpar)
int number;
double x[],y[],xi[],eta[],itpar[6];

/*
Routine calculates parameters of linear fit of (xi,eta) to (x,y)
(i.e. (xi,eta) = f(x,y), where x,y,xi and eta are just dummy variables)
*/
double matrix[3][3],sum[2][3];
int i,j;

/* initialize sums and counters */
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        matrix[i][j] = 0.0;
        if (i<2)
        {
            itpar[3*i+j] = 0.0;
            sum[i][j] = 0.0;
        }
    }
}

/* calculate sums */
for(i=0;i<number;i++)
{
    matrix[0][1] += (double)x[i];
    matrix[0][2] += (double)y[i];
    matrix[1][1] += (double)x[i]*(double)x[i];
    matrix[1][2] += (double)x[i]*(double)y[i];
    matrix[2][1] += (double)y[i]*(double)x[i];
    matrix[2][2] += (double)y[i]*(double)y[i];

    sum[0][0] += (double)xi[i];
    sum[1][0] += (double)eta[i];
    sum[0][1] += (double)xi[i]*(double)xi[i];
    sum[1][1] += (double)eta[i]*(double)eta[i];
    sum[0][2] += (double)xi[i]*(double)eta[i];
    sum[1][2] += (double)eta[i]*(double)eta[i];
}

matrix[0][0] = (double)number;
matrix[1][0] = matrix[0][1];
matrix[2][1] = matrix[1][2];
matrix[2][0] = matrix[0][2];

matinv3(matrix);

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        itpar[i] += (matrix[i][j])*(sum[0][j]);
    }
}
fitpar[3+i] += (matrix[i][j])*(sum[1][j]);
}
} /* linear_fit */

matinv3(array)
double array[3][3];
{
    /*
    Routine is a generic matrix inversion routine, taken from Bevington
    by Charles Lawrence and adapted for use with the ETC. Routine
    inverts a 3x3 matrix.
    */
    double save,amax,det;
    int ik[10],jk[10],i,j,k,l;
    det = 1.0;
    for(k=0;k<3;k++)
    {
        /* find largest element in rest of matrix */
        amax = 0.0;
        i = -1;
        j = -1;
        while(j<k) /* this inequality should be checked with the original program */
        {
            while(i<k)
            {
                i = k;
                j = k;
                for(i=k;i<3;i++)
                {
                    for(j=k;j<3;j++)
                    {
                        if (fabs(amax)<=fabs(array[i][j]))
                        {
                            amax = array[i][j];
                            ik[k] = i;
                            jk[k] = j;
                        } /* if */
                    } /* for j */
                } /* for i */
                if (amax==0.0)
                {
                    det = 0.0;
                    return;
                }
                i = ik[k];
            } /* while i */
            /* interchange rows and columns to put amax in array[k][k] */
            if (i>k)
            {
                for(j=0;j<3;j++)
                {
                    
                } /* if */
            } /* if */
    } /* for k */
} /* matinv3 */
save = array[k][j];
array[k][j] = array[i][j];
array[i][j] = -save;
} /* for j */
} /* if */
j = jk[k];
} /* while j */

if (j>k)
{
    for (i=0;i<3;i++)
    {
        save = array[i][k];
        array[i][k] = array[i][j];
        array[i][j] = -save;
    } /* for i */
} /* if */

/* accumulate elements of inverse matrix */
for (i=0;i<3;i++)
{
    if (i!=k) array[i][k] = -array[i][k]/amax;
}
for (i=0;i<3;i++)
{
    for (j=0;j<3;j++)
    {
        if ((i!=k) && (j!=k))
        {
            array[i][j] = array[i][j] + (array[i][k])*(array[k][j]);
        } /* for j */
    } /* for i */
}
for (j=0;j<3;j++)
{
    if (j!=k) array[k][j] = array[k][j]/amax;
} /* for j */
array[k][k] = 1.0/amax;
det = det*amax;

/* restore ordering of matrix */
for (l=0;l<3;l++)
{
    k = 2-l;
    j = ik[k];
    if (j>k)
    {
        for (i=0;i<3;i++)
        {
            save = array[i][k];
            array[i][k] = -array[i][j];
            array[i][j] = save;
        }
    }
}
Nov 19 01:05 1985  AStrometry.c Page 11

```c
for (j=0; j<3; j++)
    { 
    save = array[k][j];
    array[k][j] = -array[i][j];
    array[i][j] = save;
    }
}
}

prdsq(x,y,ra0,dec0,fitpar,ra,dec)
int x,y,ra0,dec0;
double fitpar[6];
int *ra,*dec;    /* ra and dec are pointers!! */
{
    /*
    prdsq calculates the ra and dec of a given x,y given the
    parameters from a linear fit.
    */
    int i;
    double *p,*q,xi,eta;

    p = fitpar;
    q = &fitpar[3];

    xi = p[0]+p[1]*(double)x+p[2]*(double)y;
    eta = q[0]+q[1]*(double)x+q[2]*(double)y;
    xinrad(ra0,dec0,xi,eta,ra,dec);    /* ra and dec are pointers!! */
}    /* end */

xinrad(ra0,dec0,xi,eta,ra,dec)
int ra0,dec0;
double xi,eta;
int *ra,*dec;    /* ra and dec are pointers!! */
{
    /*
    xinrad converts reduced coordinates xi and eta to RA and dec
    using the tangent-plane approximation.
    */
    double arg,ddec0,ddra,den,num,sindec0,cosdec0,etad;

ddec0 = micro(((double)dec0));

cosdec0 = cos(ddec0);
sindec0 = sin(ddec0);
etad = eta;
den = cosdec0 - micro(etad)*sindec0;
    if (den == 0.0)
```
{
    if (xi != 0.0) *ra = ra0 + PIe6/2*xi/abs((int)xi);
    else *ra = ra0;
}
else
{
    arg = micro(xi/den);
    *ra = ra0+mega(atan(arg));
}

ddra = micro(((double)((*ra) - ra0)));
num = ((micro(eta))*cosdec0+sindec0)*sin(ddra);

if (xi == 0.0)
{
    if (num != 0.0) *dec = PIe6/2*num/abs((int)num);
    else *dec = 0.0;
}
else
{
    arg = num/(micro(xi));
    *dec = mega(atan(arg));
}

if ((dec0>500000) && ((*dec)<0)) (*dec) += PIe6;
}

pxysq(ra0,dec0,ra,dec,fitpar,x,y)
int ra0,dec0,ra,dec;
double fitpar[6];
int *x,*y;      /* x and y are pointers!! */
{
    pxysq calculates the (x,y) in centipixels of a given (ra,dec)
in microradians given the parameters of a linear fit.
*/
int i;
double *p,*q;
double xi,eta;

    p = fitpar;
    q = &fitpar[3];

    radxin(ra0,dec0,ra,dec,&xi,&eta);

    *x = p[0]+p[1]*xi+p[2]*eta;
    *y = q[0]+q[1]*xi+q[2]*eta;
}

radxin(ra0,dec0,ra,dec,xi,eta)
int ra0,dec0,ra,dec;
double *xi,*eta;    /* xi and eta are pointers!! */
{
    Routine converts RA and dec to reduced coordinates xi and eta
using the tangent-plane approximation.

*/

double den, ddec, ddec0, ddra, sddra, cddra, cddec, cddec0, sddec, sddec0;

ddec = micro(((double)dec));
ddec0 = micro(((double)dec0));
ddra = micro(((double)(ra-ra0)));

sddra = sin(ddra);
cddra = cos(ddra);
cddec = cos(ddec);
sddec = sin(ddec);
cddec0 = cos(ddec0);
sddec0 = sin(ddec0);

den = sddec * sddec0 + cddec * cddec0 * cddra;
if (den == 0.0) *xi = *eta = 0;
else
{
  *xi = mega((cddec * sddra / den));
  *eta = mega((sddec * cddec0 - cddec * sddec0 * cddra) / den);
}

float rms(camera)
int camera;

/*
Routine calculates rms of fit and returns it in microradians
*/

int i;
double delra, deldec, sum;
int sigma, r, d;

sum = 0.0;
for (i=0; i<NUM_of_STANDARDSPer_FOV; i++)
{
  prdsq(Ast_data[i].x, Ast_data[i].y, C_data[camera].right_ascension,
        C_data[camera].declination,
        C_data[camera].p_to_c_parameters, &r, &d);
  delra = (double)(Ast_data[i].ra) - (double)mega(r);
  deldec = (double)(Ast_data[i].dec) - (double)mega(d);
  sum += delra * delra + deldec * deldec;
}

sigma = (int)(sqrt(sum / ((double)(2*NUM_of_STANDARDSPer_FOV-1))));
return(sigma);"
ASreturn_centroid(x,y)
int *x,*y;
{
    /*
     * Routine ostensibly returns the centroided value of the image in
     * HSSL_array[0]. This is done by 1) finding the highest pixel and
     * then performing a dotyfit centroiding operation on that pixel
     */

    int high,xh,yh;

    high = get_high_pixel();             /* returns value in unit pixels */
    yh = high/SUBARRAY_SIZE;
    xh = high - yh*SUBARRAY_SIZE;

    dotyfit(&xh,&yh);                    /* returns values in centipixels */

    if ((xh == -1) && (yh == -1)) *x = *y = -1;
    else
    {
        *x = (*x) + xh - 100*(SUBARRAY_SIZE/2);
        *y = (*y) + yh - 100*(SUBARRAY_SIZE/2);
    }
} /* ASreturn_centroid */

get_high_pixel()
{
    /*
     * Routine returns high pixel from the array HSSL_array[0]
     */

    register struct Subarray_str *hssl = &HSSL_array[0];
    unsigned char high;
    int j,high_j;

    high = 0;
    for(j=0;j<SUBARRAY_SIZE*SUBARRAY_SIZE;j++)
    {
        if (hssl->array[j]>high)
        {
            high = hssl->array[j];
            high_j = j;
        }
    }
    return(high_j);
} /* get_high_pixel */

dotyfit(x,y)
int *x,*y;
{
    /*
     * Routine finds the centroid of a stellar image via a parabolic
     * image profile, adapted by John Doty.
     */
Routine returns centroided coordinates in centipixels.

```c
int r, c, l, t, b, xv, yv;

xv = *x;
yv = *y;

if ((xv <= 0) || (xv >= SUBARRAY_SIZE - 1) ||
    (yv <= 0) || (yv >= SUBARRAY_SIZE - 1))
    {
    *x = 100 * (*x);
    *y = 100 * (*y);
    }
else
    {
    r = value(xv+1, yv);
    l = value(xv-1, yv);
    c = value(xv, yv);
    t = value(xv, yv+1);
    b = value(xv, yv-1);

    if (((r+l==2*c) || (t+b==2*c)) *x = *y = -1;
    else
        {
        *x = (int)(100.0 * (float)(*x) + 50.0 * ((float)(l-r)/(float)(r+l-2*c
            *y = (int)(100.0 * (float)(*y) + 50.0 * ((float)(b-t)/(float)(t+b-2*c
        }
    }
    */ dotyfit */
```

value(x, y)
```
int x, y;
{
    /*
    Routine returns value of pixel (x, y) in the array HSSL_array[0]
    in its de-mapped form.
    */
    int val;

    val = (int)rom_mapping[(int)HSSL_array[0].array[SUBARRAY_SIZE*y + x]];
    return(val);
} /* value */
/*
This module contains all subroutines involved in the execution of daytime tasks by the ETC, when automated.
*/

#include <stdio.h>
#include <sgtty.h>
#include "constants.h"
#include "extern.h"

/*** DECLARE EXTERNAL ROUTINES ***/

extern double TImjd_time();
extern Tlget_rise_and_set_times();
extern UTidle_mounts(); UTtape_storage();

DAYtime()
{
    /*
    Routine is used as entry point of overseer software, as well as being the routine to run at dawn. It simply idles the mounts at the meridian and then waits until dusk, when it starts again.

    As the entry routine, it checks whether this is an entry (by checking Sunset) sets sunrise and sunset times, determines the time of day, doing dusk routines if night and re-running this routine (with Sunset != 0.0) if day.
    */

    DAYTIME = 0;

    if (Sunset==0.0)  /* then just starting system up */
    {
        if ((Overseer_status = Tlget_rise_and_set_times()) == 2) return;
        /* sets Sunrise and Sunset, returning 2 if night, 3 if day */
    }

    UTidle_mounts();
    DAYTIME = 1;

    Tlget_rise_and_set_times();
    DAwait_until_dark();
    Overseer_status = 2;
}

DAwait_until_dark()
{
    printf("\n\nWaiting for night to fall\n\n\n\n");
    fflush(stdout);
    while(TImjd_time()<Sunset) sleep(60);
    DAYTIME = 0;
    printf("Proceeding to observation software...\n\n\n");
} /* DAwait_until_dark */
/*
This module contains all subroutines which control the dome
motion and encoding.
*/

#include <math.h>
#include <stdio.h>
#include <sgtty.h>
#include "constants.h"
#include "extern.h"

#define micro(p) (double)p/1000000.0

#define HUNDRED_MS 10000 /* values calibrated using half-second */
#define HALF_SECOND 45900 /* pauses between pulses 12/14/84 */
#define ONE_SECOND 82100
#define FIVE_SECONDS 358300

#define THIRTY_DEGREES 523599
#define FIFTEEN_DEGREES .2618
#define TWO_DEGREES 34907

extern UThour_angle(void);
extern double T1local_sidereal_time(void);

dOread_encoder()
{
    /*
    Routine reads dome encoder value. A returned value of 2313 (0x0909)
    means the dome encoder circuitry is not working
    */
    unsigned char hibyte, lobyte;
    int count;

    write(Cosmac_port[0],"azimuth ",10);
    read(Cosmac_port[0],&hibyte,1);
    sleep(1);
    ioctl(Cosmac_port[0],FIONREAD,&count);
    if (count) read(Cosmac_port[0],&lobyte,1);
    else
    {
        write(Cosmac_port[0],"azimuth ",10);
        read(Cosmac_port[0],&hibyte,1);
        ioctl(Cosmac_port[0],FIONREAD,&count);
        if (count) read(Cosmac_port[0],&lobyte,1);
        else
        {
            write(Cosmac_port[0],"azimuth ",10);
            read(Cosmac_port[0],&hibyte,1);
            ioctl(Cosmac_port[0],FIONREAD,&count);
            if (count) read(Cosmac_port[0],&lobyte,1);
        }
    }
    return((int)(hibyte<<8)+(int)lobyte);
}
DOazimuth()
{
    /*
    Routine calculates the dome azimuth from its encoder value. The
dome encoder is presently incremental, so the returned
value need not have anything to do with reality.
*/

    int azimuth, encoder;

    encoder = DOread_encoder();
    if (encoder == 2313) return(-1); /* encoder fault symptom */
    if (encoder == 2312) return(-1); /* encoder fault symptom */

    azimuth = 6041.5*encoder;

    while (azimuth > TWO_PI6) azimuth -= TWO_PI6;
    while (azimuth < 0) azimuth += TWO_PI6;

    return(azimuth); }

DOcalculate_azimuth()
{
    /* Routine calculates the azimuth of the ETC cameras */

    int dec, lat;
    double dha, ddec, dlat, A, den, num;

    dha = micro(TIlocal_sidereal_time() - C_data[0].right_ascension);

    dec = 523600;
    ddec = micro(dec);
    lat = LATITUDE;
    dlat = micro(lat);

    den = cos(ddec)*cos(dha)*sin(dlat) - sin(ddec)*cos(dlat);
    num = cos(ddec)*sin(dha);

    if (den == 0.0)
    {
        if (num == 0.0) A = 0.0;
        else if (num < 0.0) A = 4.7123390;
        else A = 1.5707963;
    }
    else A = atan(num/den);
    printf("A is %f\n", A);

    if (A < 0) A += 2.0*PI; /* compensates for negative azimuth */

    if (dha < 0.0)
    {
        if (A > PI) A -= PI;
    }
    else
    {
if (A < PI) A += PI;
}

printf("A is %f\n", A);
return((int)(A*1000000.0));
}

D0move_dome(direction, microradians)
int direction, microradians;
{
    /*
     * Routine moves the dome in a specified direction a specified angle. The motion is made in small time steps
     */
    int i, five, one, hundred, half;
    char byte;
    if (microradians < 0) microradians = -microradians;
    printf("Will be moving the dome ");
    if (direction) printf("left ");
    else printf("right ");
    printf("%d microradians\n\n", microradians);
    if (direction == 1) write(Cosmac_port[0], "left ", 7);
    else write(Cosmac_port[0], "right ", 8);
    sleep(1);
    five = microradians/FIVE_SECONDS;
    for (i=0; i<five; i++)
    {
        write(Cosmac_port[0], "long_pulse ", 13);
        read(Cosmac_port[0], &byte, 1);
    }
    one = (microradians - five*FIVE_SECONDS)/ONE_SECOND;
    for (i=0; i<one; i++)
    {
        write(Cosmac_port[0], "medium_pulse ", 15);
        read(Cosmac_port[0], &byte, 1);
    }
    half = (microradians - five*FIVE_SECONDS - one*ONE_SECOND)/HALF_SECOND;
    for (i=0; i<half; i++)
    {
        write(Cosmac_port[0], "half_pulse ", 13);
        read(Cosmac_port[0], &byte, 1);
    }
    hundred = (microradians - five*FIVE_SECONDS - one*ONE_SECOND - half*HALF_SECOND)/HUNDRED_MS;
    for (i=0; i<hundred; i++)
    {
        write(Cosmac_port[0], "short_pulse ", 14);
    }
read(Cosmac_port[0],&byte,1);
}
write(Cosmac_port[0],"right",8);
}

DOadjust_dome()
{
    /*
     * Routine attempts to move the dome an amount corresponding to
     * the motion of the cameras since the last dome motion.
     * The routine presently sucks, since the dome encoding hardware
     * does not work consistently.
     */
    int new_azimuth,delta_azimuth,azimuth,i;

    printf("Old dome azimuth was \%d\nOld camera azimuth was \%d\n",Dome_azimuth,Cam_azimuth);
    printf("The new camera azimuth is \%d\n",DOcalculate_azimuth());
    delta_azimuth = DOcalculate_azimuth() - Cam_azimuth;
    printf("Delta is \%d\n",delta_azimuth);
    while (delta_azimuth > PIe6) delta_azimuth -= TWO_PIe6;
    while (delta_azimuth < -PIe6) delta_azimuth += TWO_PIe6;
    azimuth = DOazimuth();
    /* present dome azimuth */

    if (ENCODER_WORKED)
    {
        if (azimuth == -1)
        {
            ENC_WORKING = 0;
            printf("The dome encoder is not working!!\n\n");
            if (delta_azimuth > 0) DOMove_dome(0,delta_azimuth);
            else DOMove_dome(1,delta_azimuth);
            Dome_azimuth += delta_azimuth;
        }
        else
        {
            ENC_WORKING = 1;
            for(i=0;i<4;i++)
            {
                if (delta_azimuth > 0) DOMove_dome(0,delta_azimuth);
                else DOMove_dome(1,delta_azimuth);
                new_azimuth = DOazimuth();
                if (new_azimuth == -1)
                {
                    ENC_WORKING = 0;
                    printf("The dome encoder is not working!!\n\n");
                    break;
                }
                else delta_azimuth -= new_azimuth - Dome_azimuth;
                Dome_azimuth = new_azimuth;
            }
            if (!ENC_WORKING) Dome_azimuth += delta_azimuth;
        }
    } /* if (ENCODER_WORKED) */
else
{
    if (azimuth == -1)
    {
        ENC_WORKING = 0;
        printf("The dome encoder is not working!!\n\n");
        if (delta_azimuth > 0) move_dome(0,delta_azimuth);
        else move_dome(1,delta_azimuth);
        Dome_azimuth += delta_azimuth;
    }
    else
    {
        ENC_WORKING = 1;
        if (delta_azimuth > 0) move_dome(0,delta_azimuth);
        else move_dome(1,delta_azimuth);
        Dome_azimuth += delta_azimuth;
    }
} /* else (ENCODER_WORKED) */

Cam_azimuth = DOcalculate_azimuth();
printf("New dome azimuth is \%d\nNew camera azimuth is \%d\n",Dome_azimuth, Cam_azimuth);
}

DOinitialize_dome()
{
    int azimuth;

    azimuth = DOazimuth();
    if (azimuth == -1)
    {
        ENC_WORKING = ENCODER_WORKED = 0;
        Dome_azimuth = DOcalculate_azimuth();
        Cam_azimuth = DOcalculate_azimuth();
        printf("Dome encoder is not working\n\n");
        printf("BE AWARE THAT THE DOME MAY NOT MOVE!!\n");
    }
    else
    {
        ENC_WORKING = ENCODER_WORKED = 1;
        Dome_azimuth = azimuth;
        Cam_azimuth = DOcalculate_azimuth();
        printf("The dome encoder worked at initialization!!\n");
    }
}

DOmecheck()
{
    int i;
    char response[];

    C_data[0].right_ascension = UTra_of_ha(UThour_angle(0));
    C_data[1].right_ascension = C_data[0].right_ascension;
    printf("RA is \%d\n",C_data[0].right_ascension);
    printf("Initialize dome! ");
scanf("%s", response);
D0initialize_dome();

for (i = 0; i < 3; i++)
{
    printf("Adjust dome! ");
    scanf("%s", response);
    C_data[i].right_ascension = UTra_of_ha(UTHour_angle(0));
    D0adjust_dome();
}
/*
   This module is included to have a place to put all future routines
   for the communication of information to peripheral computer systems,
   specifically the RMT.
*/

#include "constants.h"
#include "extern.h"

EXsend_to_RMT(ra,dec)
double ra,dec;
{
    int nbytes;

    printf("In EXsend_to_RMT()\n");
    nbytes = write(RMT_port,&ra,8);
    nbytes = write(RMT_port,&dec,8);

    /* wait for response?? */
}

EXphone_home()
{
}
This module contains only subroutines contributed by John Doty to improve the speed of execution of the Overseer software. Comments below are jpd's

```
#define micro(p) ((double)p/1000000.0)
#define mega(p) ((double)p*1000000.0)

JPDmove( source, dest, count )
char *source, *dest;
unsigned count;
{ /* 2/25/85 jpd */
  /* Move bytes from source to destination;
  After the JPDmove, the destination area always contains an
  exact copy of what the source area contained before the move.
  Simple cases are done very fast. */

  register char *s = source;
  register char *d = dest;
  register unsigned c = count;
  register int key = (int) d - (int) s;

  if( !c ) return; /* Degenerate case */
  if(( key & 1 ) == 0
      && key <= -4 || key > c
      && c > 4 ) { /* Fixup odd addr */
      if((int) s & 1 ) {
        *d++ = *s++;
        --c;
      }
  }

  { /* Move four bytes at a time */
    register long *qs = (long *) s;
    register long *qd = (long *) d;
    register unsigned qc = c >> 2;
    do {
      *qd++ = *qs++;
    } while( --qc );
    if(( c = c & 3 ) == 0 ) return; /* No residue */
    s = (char *) qs;
    d = (char *) qd;
  }

  if( s > d ) { /* Forward is safe */
    do {
      /*
      */
    } while( --qc );
  }
  if( !c ) return; /* No residue */
}
```
*d++ = *s++;
} while( --c );
}

} else {
  s += c;
  d += c;
  do {
    *--d = *--s;
  } while( --c );
}
JPDsnth( data, numdata, n )
short data[]; /* Data array (will be scrambled on return) */
int numdata; /* length of data array */
int n; /* index if item to find:
1 <= n <= numdata */
{
  /* Find the nth from the minimum value in an array */
  /* Monte Carlo method intended for finding medians */
  /* 2/13/85 jpd */
  /* For random data, this routine takes about */
  /* 2.6*numdata + O( log( numdata ) ) comparisons */
  /* If the data is tightly clustered about the mean, */
  /* there is a speedup; it may take as few as */
  /* 0.5*numdata comparisons. */
  /* There is a slight penalty if the array is completely */
  /* or partially sorted; it is at most 25%. */

  register short boundary, thisdata;
  register short *lowp, *highp;
  short v1, v2;
  int nlowbin;

  lowp = data; /* Init data pointer */
  v1 = data[ ixrand( numdata )];
  {
    register short v1r = v1;
    int nc = 1 + numdata - n; /* "Complement" of n */
    if( nc > n )
      highp = lowp + nc;
    else
      highp = lowp + n; /* Limit to test for done */

    /* Scan for the first point which doesn't match the boundary point */
    if( v2 < v1 )
      for( ; lowp < highp; thisdata = *lowp )
        if( thisdata <= boundary )
          *lowp = *--highp; /* Exchange */

    /* Back up to set point */
    v2 = *--lowp;
  }

  /* Beware overflows */
  boundary = ( v1 >> 1 ) + ( v2 >> 1 );
  highp = data + numdata;
  thisdata = *lowp; /* Now process the whole thing */
  /* Prime the pump */

  if( v2 < v1 )
    /* Bin 2 is low bin */
    for( ; lowp < highp; thisdata = *lowp )
      if( thisdata <= boundary )
        *lowp = *--highp; /* Exchange */
*highp = thisdata;
}
else ++lowp; /* Data point in right place */
}

nlowbin = numdata - ( lowp - data );
if( nlowbin >= n ) return( JPDSnth( highp, nlowbin, n ) );
else return( JPDSnth( data, lowp - data, n - nlowbin ) );
}

else { /* Primary bin is low bin */

for( ; lowp < highp; thisdata = *lowp ) {
    if( thisdata > boundary ) { /* Bin 2 */
        *lowp = *--highp; /* Exchange */
        *highp = thisdata;
    }
else ++lowp; /* Don't move point */
}

nlowbin = ( lowp - data );
if( nlowbin >= n ) return( JPDSnth( data, nlowbin, n ) );
else return( JPDSnth( highp, numdata - nlowbin, n - nlowbin ) );
}
/* 3/16/85 jpd */
/*
Select a well spaced set of stars from a larger set.
To use this package, first call JPDstarlim() to define
the field of view. Next, call JPDputstar() for each
star in the larger set. Then call JPDgetstar() to obtain
selected stars, one star per call. Finally, call
JPDfreestars() to release the memory occupied by the
larger set. Comments below describe details of each
function.
*/
char *malloc();    /* keep lint happy */

struct stars {
    int nearest, x, y;
    long id;
    struct stars *link
};
#define NULL_STAR ((struct stars *) 0)
static struct stars *first_star = NULL_STAR;
static struct stars **last_link = &first_star;
static int xmin, xmax, ymin, ymax;
#define NORM(x,y) (ABS(x)+ABS(y))    /* Minkowskian norm */
#define ABS(x) (((x)<0)?(-(x)):x))
#define MIN(x,y) (((x)<(y))?(x):(y))

/*
JPDgetstar() returns the id of the star which is "farthest"
from all other selected stars and from the borders.
*/
long JPDgetstar()
{
    register int norm, x, y, dx, dy;
    register struct stars *this_star;
    register struct stars *best_star = NULL_STAR;

    /* Scan to find star farthest from anything */
    this_star = first_star;
    norm = 0;    /* best distance so far */
    while( this_star ) {
        if( (this_star->nearest > norm) {
            norm = this_star->nearest;
            best_star = this_star;
        }
        this_star = this_star->link;    /* Next */
    }

    if( NULL_STAR == best_star ) return(-1);    /* fail */
/* readjust distances as needed */

x = best_star->x;
y = best_star->y;
this_star = first_star;
while( this_star ) {
    dx = x - this_star->x;
    dy = y - this_star->y;
    norm = NORM( dx, dy );
    if( norm < this_star->nearest )
        this_star->nearest = norm;
    this_star = this_star->link;
}

return( best_star->id );

JPDputstar( id, x, y )
long id;
int x, y;
[...]
new_star = (struct stars *) malloc( sizeof(struct stars) );
new_star->id = id;
new_star->x = x;
new_star->y = y;
dx1 = x - xmin;
dx2 = xmax - x;
dy1 = y - ymin;
dy2 = ymax - y;
dx = MIN( dx1, dx2 );
dy = MIN( dy1, dy2 );
new_star->nearest = MIN( dx, dy )*2;
new_star->link = NULL_STAR;
*last_link = new_star;
last_link = &(new_star->link);
]

/*
JPDfreestars() removes all stars from the list and
releases the memory occupied by the list for other
use.

JPDfreestars()
{
    register struct stars *star = first_star;

    while( star ) {
        first_star = star->link;
        free((char *) star);
        star = first_star;
    }
    last_link = &first_star;
}

JPDstarlim() sets the boundaries of the field of view.

JPDstarlim( xlow, xhigh, ylow, yhigh )
int xlow, xhigh, ylow, yhigh;
{
    xmin = xlow;
    xmax = xhigh;
    ymin = ylow;
    ymax = yhigh;
}
JPDxytoradec( x, y, f, ra, dec )

int x, y; /* ccd coordinates */
struct jpdxyfit *f; /* parameters */
int *ra, *dec; /* output pointers */
[

double v1, v2, v3; /* Vector coordinates */

v1 = f->f1 + f->fx1 * x + f->fy1 * y;
v2 = f->f2 + f->fx2 * x + f->fy2 * y;
v3 = f->f3 + f->fx3 * x + f->fy3 * y;

*ra = mega( atan2( v2, v1 ));
if( *ra < 0 ) *ra += 2*PIe6;
*dec = mega( atan2( v3, hypot( v1, v2 )));
]

JPDmkxyfit( ra0, dec0, fitpar, f )

int ra0, dec0;
double fitpar[ 6 ];
struct jpdxyfit *f;
[

double p0 = micro( fitpar[0] ),
p1 = micro( fitpar[1] ),
p2 = micro( fitpar[2] ),
q0 = micro( fitpar[3] ),
q1 = micro( fitpar[4] ),
q2 = micro( fitpar[5] );
double dra = micro( ra0 ),
ddec = micro( dec0 );
double sr = sin( dra ),
 cr = cos( dra ),
sd = sin( ddec ),
 cd = cos( ddec );
double crsd = cr * sd,
srsd = sr * sd;

f->f1 = cr + sr * p0 - crsd * q0;
f->f2 = sr + cr * p0 - srsd * q0;
f->f3 = sd + cd * q0;

f->fx1 = sr * p1 - crsd * q1;
f->fx2 = cr * p1 - srsd * q1;
f->fx3 = cd * q1;

f->fy1 = sr * p2 - crsd * q2;
f->fy2 = cr * p2 - srsd * q2;
f->fy3 = cd * q2;
]
JPDradectoxy(ra, dec, f, x, y)
int ra, dec;
struct jpdfitxy *f;
int **x, **y;
{
    double dra = micro(ra);
    ddec = micro(dec);
    double vx = cos(dra);
    vy = sin(dra);
    vz = sin(ddec);
    double tpcos = vx * f->x0 + vy * f->y0 + vz * f->z0;
    double vtx = vx / tpcos;
    vty = vy / tpcos;
    vtz = vz / tpcos;
    **x = mega(f->fx0 +
               f->fx1 * vtx +
               f->fx2 * vty +
               f->fx3 * vtz);
    **y = mega(f->fy0 +
               f->fy1 * vtx +
               f->fy2 * vty +
               f->fy3 * vtz);
}

JPDMkfitxy(ra0, dec0, fitpar, f)
int ra0, dec0;
double fitpar[6];
register struct jpdfitxy *f;
{
    double p0 = fitpar[0],
            p1 = fitpar[1],
            p2 = fitpar[2],
            q0 = fitpar[3],
            q1 = fitpar[4],
            q2 = fitpar[5];
    double dra = micro(ra0),
            ddec = micro(dec0);
    double sr = sin(dra),
            cr = cos(dra),
            sd = sin(ddec),
            cd = cos(ddec);
    double crsd = cr * sd,
            srsd = sr * sd;
    f->x0 = cr;
    f->y0 = sr;
    f->z0 = sd;
    f->fx1 = p1 * sr - p2 * crsd;
    f->fx2 = p1 * cr - p2 * srsd;
    f->fx3 = p2 * cd;
    f->fx0 = p0 - cr * f->fx1 - sr * f->fx2 - sd * f->fx3;
f->ty1 = q1 * sr - q2 * crsd;
f->ty2 = q1 * cr - q2 * srsd;
f->ty3 = q2 * cdi;
f->ty0 = q0 - cr * f->ty1 - sr * f->ty2 - sd * f->ty3;
This module includes all tools for calculating the brightness of flash and stellar images. Presently unimplemented.

```c
#include "constants.h"
#include "extern.h"

PHotometry(arrayxycamera) unsigned char array[SUBARRAY_SIZE*SUBARRAY_SIZE]; int x,y,camera;
{ /*
   Routine calculates the amount of adu by which the sum of the values of the pixels in array[] exceeds the background, based on the row and column vectors previously calculated.
*/
    register char *column;
    register unsigned char *row;
    int i,j,h,sum;
    x = 0.01*x + 0.5;
    y = 0.01*y + 0.5;
    h = SUBARRAY_SIZE/2;
    sum = S_SQUARED*(int)C_data[camera].adu_offset;
    row = &Row_threshold[NCOLUMNS*camera+x-h];
    column = &Column_threshold[NROWS*camera+y-h];

    for(i=0;i<SUBARRAY_SIZE;i++)
    { for(j=0;j<SUBARRAY_SIZE;j++)
        sum += rom_mapping[(int)(*array++)] - *row++ - *column;
        row -= SUBARRAY_SIZE;
        column++;
    }

    return(sum);
}
This module contains all subroutines involved in the selection, assignment and display of SAO standard stars used in the ETC.

#include "constants.h"
#include "extern.h"

#include <stdio.h>
#include <sgtty.h>
#include <math.h>

#define micro(p) \n(p/1000000.0)
#define MAX(x,y) (((x)>(y))?(x):(y))
#define MIN(x,y) (((x)<(y))?(x):(y))

#define BORDER 5
#define SEARCH_SIZE 131 /* Subarray size (odd number). */
#define S_SIZE_SQ SEARCH_SIZE*SEARCH_SIZE
#define SPS 3 /* Number of locations checked in algorithm
#define SSAS 3 /* Sub-subarray size (odd num.) */
#define PIX_DIST 1

/*** DEDECLARE EXTERNAL ROUTINES ***/
extern TRassign_standard();
extern ASget_position();
extern long JPDgetstars();
extern JPDputstars(),JPDfreestars(),JPDstarlim();

/*** TWO INTERNAL VARIABLES ***/
int SAO_channel,max_ra,max_dec;
unsigned char lineup_array[S_SIZE_SQ],freq_array[S_SIZE_SQ];
struct pix
{
    unsigned char value;
    int i;
    int j;
} high[SPS];

/*** SAo.c ROUTINES ***/

SAfill_SA0_structure()
{
    /* Routine runs SAget_stars for all cameras */
    int i;
    for(i=0;i<NUM_of_CAMERAS;i++) SAget_stars(i);
}

SAget_stars(camera)
int camera;
Routine finds NUM_of_STANDARDS_per_FOV stars in the FOV of camera #came given that it is centered on (ra0,dec0) and based on the angle of the C on the sky. To promote smoothness in the distribution, the field is divided into fourths, the quarter to the right of center being filled first.

The structure SAO_file is filled with the pertinent data.

```c
int i,j,k,straddle,ra_ur,ra_ll,dec_ur,dec_ll,hash[NUM_of_STANDARDS_per_FOV];
int x,y,ra_high,ra_low,dec_high,dec_low,nbytes,offset;counter=0;
struct star_str
{
  int number,ra,dec;
  short vmag;
} star[100];
register struct SAO_entry_str *saaf = &SAO_file[.camera];
SAO_channel = open("/usr/ETC/overseer/SAOfile",0);

printf("Fillin SAO structure for camera %c\n",letters[.camera]);
lseek(SAO_channel,0,0);

ASget_coordinates(camera,NCOLUMNS*100,NROWS*100,&ra,dec)
ASget_coordinates(camera,0,0,&ra,dec)
ra_high = MAX(ra,ra_1l);
ra_low = MIN(ra,ra_1l);
dec_high = MAX(dec,dec_1l);
dec_low = MIN(dec,dec_1l);

JPDstarlim(BORDER*100,(NCOLUMNS-BORDER)*100,BORDER*100,(NROWS-BORDER)*100)
straddle = 0;
if (ra_high - ra_low > PIe6) straddle = 1;

for(i=0;i<NUM_of_STANDARDS_per_FOV;i++) saaf->SAO_star_number[i] = 0;
while ((nbytes=read(SAO_channel,star,1400)) > 0)
{
  for (j=0;j<nbytes/14;j++)
  {
    register struct star_str *sj = &star[j];
    if (((sj->dec > dec_low) & (sj->dec < dec_high))
      {
      if (((sj->ra < ra_high) & (sj->ra > ra_low) & (!straddle))
        {
          ASget_position(camera,sj->ra,sj->dec,&x,&y);
          JPDputstar(tell(SAO_channel)-nbytes+14*j,x,y);
          counter++;
        }
      }
    }
  }
if (straddle)
{
  if (((sj->ra > ra_high) & (sj->ra < TWO_PI_e6))
```
Nov 19 01:06 1985 SAo.c Page 3

```
((sj->ra < ra_low) && (sj->ra > 0))

ASget_position(camera,sj->ra,sj->dec,&x,&y);
JPDputstar(teil(SAO_channel)-nbytes+14,x,y);
counter++;
}
}

printf("There are \n%d SAO stars in this field",counter);
for(i=0;i<NUM_of_STANDARDS_per_FOV;i++)
{
    register struct star_str *sj;
    offset = JPDgetstar();

    if(offset != -1)
    {
        lseek(SAO_channel,offset,0);
        read(SAO_channel, sj, 14);
        saof->SAO_coordinates[i].x = sj->ra;
        saof->SAO_coordinates[i].y = sj->dec;
        ASget_position(camera,sj->ra,sj->dec,&x,&y);
        saof->SAO_position[i].x = x;
        saof->SAO_position[i].y = y;
        saof->SAO_star_number[i] = sj->number;
        saof->SAO_magnitude[i] = sj->vmag;
        /*
        saof->SAO_brightness[i] = (short)calc_brightness(sj->vmag,camera,x,y)
        */
    }
}
close(SAO_channel);
SAassign_standards(camera);
JPDfreestars();
}

calc_brightness(mag,camera,x,y)
short mag;
int camera,x,y;
{
    /*
    Routine calculates the expected brightness in adu of a mag-th
    magnitude star, based on the throughput of the lens, the gain
    of the chip, the vignetting of the lens and the area of the
    lens. Presently unimplemented.
    */
}
register struct C_option_str *cdc = &C_data[camera];
float area = cdc->area_of_lens/100000000.0;
float photons = cdc->fifteenth_mag * pow(10,15-mag);
float time = S_data.exposure_time/100.0;
float inv_gain = 1.0/cdc->gain;
float efficiency = cdc->peak_efficiency;
```
int radius;
float throughput;

printf("In calc_brightness\\n");
radius = sqrt((double)(x*x) + (double)(y*y));
throughput = 1+cdc->linear_term*radius+cdc->quadratic_term*(radius*radius)
return(photons*area*time/(throughput*inv_gain*efficiency));

SAassign_standards(camera)
int camera;
{
  /*
   Routine assigns each SAO standard selected by the ETC a reference
   number used by the Trigger in the transfer of image data.
   */

  register struct SAO_entry_str *saof = &SAO_file[camera];
  int i,ra,dec,x,y;

  for(i=0;i<NUM_OF_STANDARDS_PER_FOV;i++)
  {
    Used_trigger_number[64*camera+i] = 0;
    if (saof->SAO_star_number[i] != 0)
    {
      ra = saof->SAO_coordinates[i].x;
      dec = saof->SAO_coordinates[i].y;
      ASget_position(camera,ra,dec,&x,&y);

      if ((x>BORDER*100) && (x<(NCOLUMNS-BORDER)*100) &&
          (y>BORDER*100) && (y<NROWS*100))
      {
        TRassign_standard(camera,i,x/100,y/100);
        Used_trigger_number[64*camera+i] = 1;
      }
    }
  }
}

SAlineup(camera)
int camera;
{
  /* Shift (x, y) coords to line up with SAO stars. */
  /* 10/85 eaa (E. Ajhar) */

  /* Routine is an automatic field-finding subroutine, which determines
   the locations of the SAO standard stars in the field by finding
   the brightest pixels in a subarray around the predicted locations
   of the SAO stars (from ASrough_astrometry). The location
   most frequently containing the brightest pixel in the field
   is supposed to be the location of the SAO standards in the
   field. */
}
int hi_i, hi_j, delta_x, delta_y, check, j, k, s, OK_count, location, i, ki, kj;
double cos_dec, sin_angle, cos_angle;
unsigned char hi = 0;

printf("Attempting to autoalign camera %c \n", letters[camera]);
fflush(stdout);
cos_dec = cos((micro(C_data[camera].declination)));
cos_angle = cos((micro(C_data[camera].ccd_angle)));
sin_angle = sin((micro(C_data[camera].ccd_angle)));

i = S_SIZE*SQ;
while (--i != -1) freq_array[i] = 0;

for(s=0; s<NUM_of_STANDARDS_per_FOV; s++)
{
    register struct Position_str *saop = &SAO_file[camera].SAO_position[s];
    TRrandom_array(camera,(int)(saop->x/100.0+.5),(int)(saop->y/100.0+.5),
                   SEARCH_SIZE, lineup_array);
    for (k=0; k<SPS; ++k) high[k].value = high[k].i = high[k].j = 0;
    for (ki=0; ki<SEARCH_SIZE; ++ki)
    {
        for (kj=0; kj<SEARCH_SIZE; ++kj)
        {
            location = SEARCH_SIZE*ki+kj;
            if (lineup_array[location] > high[i].value)
            {
                OK_count = 0;
                for (k=0; k<SPS; ++k)
                {
                    if ((abs(high[k].i-ki) > PIX_DIST) &&
                       (abs(high[k].j-kj) > PIX_DIST)) OK_count++;
                }
                if (OK_count == SPS)
                {
                    for (j=SPS-1; j>i; --j) high[j] = high[j-1];
                    high[i].value = lineup_array[location];
                    high[i].i = ki;
                    high[i].j = kj;
                    /* if OK_count */
                } /* if lineup */
                /* if i */
                /* k */
                /* ki */
                increment_frequency_array(high);
            } /* if OK_count */
            /* if s */
        } /* ki */
    } /* kj */
}

for (i=0; i<SEARCH_SIZE; ++i)
{
    for (j=0; j<SEARCH_SIZE; ++j)
    {
        location = i*SEARCH_SIZE+j;
if (freq_array[location] > hi)
{
    hi = freq_array[location];
    hi_i = i;
    hi_j = j;
}
}

delta_x = hi_j - (SEARCH_SIZE-1)/2;
delta_y = hi_i - (SEARCH_SIZE-1)/2;

C_data[camera].right_ascension -= C_data[camera].radians_per_pixel*  
    (delta_x*cos_angle+delta_y*sin_angle)/cos_dec;
C_data[camera].declination -= C_data[camera].radians_per_pixel*  
    (delta_y*cos_angle-delta_x*sin_angle);
printf("Completed field alignment!\n");
fflush(stdout);
}
}
}
}
}
}
}
}
}
}

increment_frequency_array(high)
struct pix high[SPS];
{
    /*
     * Routine increments a counter of the frequency of selection of
     * a given location as containing one of the brightest pixels
     * in a subarray (see SA_lineup)
     */
    int i, j, k, location, x, y;
    for (k=0; k<SPS; k++)
    {
        i = high[k].i;
        j = high[k].j;
        for (x=i-1; x<i+2; x++)
        {
            for (y=j-1; y<j+2; y++)
            {
                if (x >= 0 && x <= SEARCH_SIZE-1 &&  
                    y >= 0 && y <= SEARCH_SIZE
                {
                    location = x*SEARCH_SIZE+y;
                    ++freq_array[location];
                }
            }
        }
    }
}

SAauto_lineup()
{
    /* Routine executes the automatic field-finding routine for all CCDs */
    int i, error = 0;

for(i=0;i<NUM_of_CAMERAS;i++)
{
    SAlineup(i);
    SAassign_standards(i);
    if ( do_astrometry(i) && (error == 0)) error = 1;
}
return(error);
#include <stdio.h>

extern WWv();
extern ENtry();

main()
{
    /*
    * This routine simply enters ENtry.c and lets it do all the work.
    * This routine is necessary because sift_test also wants to access ENtry.
    */
    /WWv();*/
    ENtry();
}
This module contains all subroutines involved in the timing of ETC operations, including the frame timer and the day-date timer.

```c
#include "constants.h"
#include "extern.h"
#include <sys/types.h>
#include <sys/timeb.h>
#include <math.h>
#include <stdio.h>

#define frac(p) (double)p - (double)(int)p
#define micro(p) (double)p/1000000.0

double sunrise(), sunset(), TImjd_time(), TIlocal_sidereal_time();
extern UThour_angle();

double TImjd_time()
{
  /* Routine returns present time in modified Julian days UT */
  struct timeb seconds;
  ftime(&seconds);
  return((seconds.time + seconds.millitm/1000.0)/86400.0 + 40587.0);
}

double TIlocal_sidereal_time()
{
  /* Routine returns present local sidereal time in real radians */
  double time, gmst, lmst;
  time = TImjd_time();
  gmst = 0.97321 + 0.01720279*((double)(int)time) + 6.300288*(frac(time));
  lmst = gmst + micro(EAST_LONGITUDE);
  while(lmst > TWO_PI) lmst -= TWO_PI;
  /*printf("LMST = %f RA = %d\n", lmst,C_data[0].right_ascension);*/
  UThour_angle();
  return(lmst);
}

TIget_rise_and_set_times()
{
  /* Routine calculates sunrise and sunset, returning 3 if it is daytime,
     a 2 if it is night */

  double time, day;
  int response;
```
time = TImjd_time();
day = (double)(int)time;
Sunrise = sunrise() + day;
Sunset = sunset() + day;

Tlam_twilight();
Tlpm_twilight();

if (PM_twilight > AM_twilight) /* PM_twilight before 5 PM KPNO */
{
    if ((time > AM_twilight) && (time < PM_twilight))
    {
        response = 3;
        Sunrise += 1.0;
    }
    else if (time < AM_twilight) response = 2; /* night */
    else if (time > PM_twilight)
    {
        response = 2;
        Sunrise += 1.0;
    }
}
else /* sunset after 5 PM AST */
{
    if ((time > PM_twilight) && (time < AM_twilight)) response = 2; /* night */
    if (time < PM_twilight) response = 3; /* day */
    if (time > AM_twilight)
    {
        response = 3;
        Sunrise += 1.0;
        Sunset += 1.0;
    }
}

Tlam_twilight();
Tlpm_twilight();

return(response);
} /* Tlget_rise_and_set_times */

solar_coordinates(ra,dec)
double *ra;
double *dec;
{
    /* Routine returns present solar coordinates in decimal (float) radians */

double g,l,lambda,time;

time = TImjd_time();
g = 5.5024 + 0.01720279*time;
l = 4.1151 + 0.01720279*time;
lambda = l + 0.03344*sin(g) + 0.000349*sin(2.0*g);

while(lambda > TWO_PI) lambda -= TWO_PI;
while (lambda < 0.0) lambda += TWO_PI;

*ra = atan(0.91747*tan(lambda));

while (lambda - (*ra) > TWO_PI/4.0) *ra += TWO_PI/2.0;
while (lambda - (*ra) < -TWO_PI/4.0) *ra -= TWO_PI/2.0;

*dec = asin(0.3978*sin(lambda));
}

double sunrise()
{
    /* Routine returns time of sunrise in fractional (<1.0) days UT */
    double time, ra, dec, lms, gmst0, rise_time;
    time = TImjd_time();
    solar_coordinates(&ra, &dec);
    lms = TIlocal_sideral_time();
    gmst0 = lms - micro(EAST_LONGITUDE) - 6.300288*(frac(time));
    rise_time = 0.1587205*(ra - micro(EAST_LONGITUDE) - acos(-tan(micro(LATITUDE))))
    if (rise_time<0.0) rise_time += 1.0;
    if (rise_time>1.0) rise_time -= 1.0;
    return(rise_time);
}

double sunset()
{
    /* Routine returns time of sunset in fractional days UT */
    double time, ra, dec, lms, gmst0, set_time;
    time = TImjd_time();
    solar_coordinates(&ra, &dec);
    lms = TIlocal_sideral_time();
    gmst0 = lms - micro(EAST_LONGITUDE) - 6.28319*(frac(time));
    set_time = 0.1587205*(ra - micro(EAST_LONGITUDE) + acos(-tan(micro(LATITUDE))))
    if (set_time<0.0) set_time += 1.0;
    if (set_time>1.0) set_time -= 1.0;
    return(set_time);
}

TIm_twilight()
{
    /* Routine sets AM twilight to be 1.1 hours before sunrise */
AM_twilight = Sunrise - 1.1/24.0;
}

T1pm_twilight()
{
    /* Routine sets PM twilight to be 1.1 hours after sunset */

    PM_twilight = Sunset + 1.1/24.0;
}
TIset_frame_timer(tmr_units)
int tmr_units;
{
    /*
       Sets frame timer to time indicated in argument list.
       Units are multiples of 4 microseconds.
    */
    ioctl(Timer_port,INIT_REQUEST,&tmr_units);
}

TIstart_timer()
{
    /* Routine starts frame timer */
    ioctl(Timer_port,START_TIMER,0);
}

TIwatch_frame_timer()
{
    /* Routine watches frame timer until it expires */
    int signal;

    signal = 0;
    while(signal == 0) ioctl(Timer_port,ZERO_DETECT,&signal);
}
This module contains many software subroutines which fit in no one defineable module. The subroutines found here bring together functional subroutines from other module and create small, concise subroutines which have a single, user-friendly function. The subroutines are roughly grouped by function:

*) Median frame acquisition
*) CCD image acquisition through Trigger processors
*) Sidereal drive control
*) Astrometry-related routines
*) Weather and peripheral I/O routines (presently undefined)
*) Data storage and management routines
*) Image-measurement and calculation routines
*) Timing control

All subroutines in this module were written by Roland Vanderspek.
All subroutines are in a state of constant update.
- RKV 21.10.85

*/

#include "constants.h"
#include "extern.h"
#include "ccphot.h"

#include <stdio.h>
#include <sgtty.h>
#include <math.h>
#include <time.h>

#define mega(p) (double)p*1000000.0
#define micro(p) (double)p/1000000.0

#define STEPS_per_SYNCH 4.166667
#define SKY_ARRAY_SIZE 35
#define MAX_SKY 55 /* was 75 */
#define MIN_SKY 20 /* was 40 */
#define IDEAL_SKY 30

/*****************************/
/* DECLARE EXTERNAL ROUTINES */
/*****************************/

extern TRTrendqueries(), TRcalculate_threshold_from_offset(), TRsnd_entire_image()
extern TRRsend_standard_subarrays(), TRget_median_data(), TRdscc_current_frame()
extern TRsift_against_old(), TRdscc_old_frame(), TRsnd_array(), TRassign_standard()
extern TRrandom_array(), TRset_sift_time(), TRlong_query();

extern TIlwatch_frame_timer(), TIlset_frame_timer();
extern double TIllocal_sidereal_time(), TIlmd_time();

extern COreadout_CCDS(), COflush_CCDS(), COTemperature(), COSlew_clutch();
extern COTrack_clutch(), COeastT(), COeastM(), COeastC(), COset_offset();
extern COwestT(), COwestM(), COwestC(), COSynchro_angle();
extern unsigned char COget_offset();

/* UTility.c: Introduced to ETC Overseer Computer software 12/84 */
extern ASdo_astrometry(), ASget_position(), ASreturn_centroid();
extern ASrough_astrometry(), ASrecenter();
extern SAassign_standards();
extern PHotometry();
extern JPDsntk();
struct tm * localtime(), * gmt ime();
UTget_median_frames()  /* asks all triggers to calculate a median frame based on the frame in 'current' memory and return the row and column vectors */
{
    /* start */
    int i;

    printf("Calculating comparison frame for all cameras\n");
    fflush(stdout);
    UTreadout_one_frame();
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        if (Trig_status[i] != -1)
        {
            printf("Requesting of %c with offset of %d\n",letters[i],
                    C_data[i].adu_offset);
            TRcalculate_threshold_from Offset(i,C_data[i].adu_offset);
        }
    }
    printf("\n");
    fflush(stdout);

    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        if (Trig_status[i] != -1)
        {
            TRlong_query(i);
            printf("Camera %c responded\n",letters[i]);
            TRget_median_data(i);
        }
    }
    printf("\n");
    fflush(stdout);
} /* UTget_median_frames() */
UTreadout_one_frame()
{
/*
UTreadout_one_frame does necessary manipulations to get one
    clean frame into trigger memory without it being sifted.
CCDs are read out twice to make VERY sure that the image exposure
time is precisely what it is supposed to be.
Returns -1 if any error condition persists
*/

int error, counter;

printf("\nReading a clean frame into trigger memory\n\n");
fflush(stdout);

eat_loose_bytes();
COflush_CCDs();         /* first, flush to get out the bad data */

TRset_sift_time();
UTset_frame_timer();
Tlstart_timer();        /* then read out one image */

error = 1;
counter = 0;
while ((error == 1) && (counter < MAX_TRIES))
{
    /* TRdsc_current_frame(); */
    TRdsc_old_frame();
    TRsift_against_old();
    TRend_queries();
    Tlwatch_frame_timer();
    COreadout_CCDs();
    error = watch_triggers();
    counter++;
}

if (counter == MAX_TRIES) return(-1);

error = 1;
counter = 0;
while ((error == 1) && (counter < MAX_TRIES))
{
    TRdsc_current_frame();
    TRend_queries();
    Tlwatch_frame_timer();
    Image_exposure_time.b = TImjd_time();
    COreadout_CCDs();
    error = watch_triggers();
}

if (counter == MAX_TRIES) return(-1);
return(0);
} /* UTreadout_one_frame */
UTdirty_frame()
[
    /*
        UTdirty_frame does necessary manipulations to get one
        clean frame into trigger memory without it being sifted
        Returns -1 if any error condition persists
    */
    int error, counter;
    printf("\nReading a dirty frame into trigger memory\n\n");
    fflush(stdout);

    eat_loose_bytes();
    COflush_CCDs(); /* first, flush to get out the bad data */
    TRset_sift_time();
    UTset_frame_timer();
    Tlstart_timer(); /* then read out one image */

    error = 1;
    counter = 0;
    while ((error == 1) && (counter < MAX_TRIES))
    {
        TRdesc_current_frame();
        TRend_queries();
        Tlwatch_frame_timer();
        COrreadout_CCDs();
        error = watch_triggers();
        counter++;
    }
    if (counter == MAX_TRIES) return(-1);
    return(0);
] /* UTdirty_frame */

watch_triggers()
[
    /*
        Routine monitors triggers for expected sift-termination reports
        issued by each Trigger after each image is read into Trigger memory.
        The copious ioctl calls are to prevent confusion due to the loss of byt
    */
    int i, k, nbytes, done_processing, report[NUM_of_CAMERAS], count, flag, response
    unsigned char error, rep, r1, r2;
    short row;

    response = done_processing = 0;
    for (i = 0; i < NUM_of_CAMERAS; i++)
    {
        if (Trig_status[i] == -1)
        {
            done_processing++;
            report[i] = 1;
            /* continue */
        }
        /* do something */
    }
] /* watch_triggers */
flag = 0;
while((done_processing(NUM_of_CAMERAS) && (flag == 0))
{
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        if(report[i]==0)
        {
            ioctl(Trig_port[i],FIONREAD,&count);
            if (count!=0)
            {
                for(k=0;k<MAXTRIES;k++)
                {
                    ioctl(Trig_port[i],FIONREAD,&count);
                    if (count) break;
                }
                if (count == 0) error = 7;
            }
            else nbytes = read(Trig_port[i],&rep,1);
            for(k=0;k<MAXTRIES;k++)
            {
                ioctl(Trig_port[i],FIONREAD,&count);
                if (count) break;
            }
            if (count == 0) error = 7;
            else nbytes = read(Trig_port[i],&error,1);
            for(k=0;k<MAXTRIES;k++)
            {
                ioctl(Trig_port[i],FIONREAD,&count);
                if (count) break;
            }
            if (count == 0) error = 7;
            else nbytes = read(Trig_port[i],&r1,1);
            for(k=0;k<MAXTRIES;k++)
            {
                ioctl(Trig_port[i],FIONREAD,&count);
                if (count) break;
            }
            if (count == 0) error = 7;
            else nbytes = read(Trig_port[i],&r2,1);
            row = 256*(short)r1 + (short)r2;
            if (rep != 64)
            {
                printf("rep: %d   error: %d   row: %d\n",rep,error,row);
                consume_byes(Trig_port[i]);
            }
            else
            {
                printf("%c complete",letters[i]);
            }
        }
    }
}
}
if (error != 4) printf("; error = %d\n",error);
printf("
");
flush(stdout);
}

if ((error == 2) || (error == 6) || (rep != 64)) response = 1;

done_processing++;
report[i] = 1;
}
}
ioctl(Timer_port, ZERO_DETECT, &flag);
}
if (flag == 1)
{
    response = 1;
    printf("Readout time exceeded\n");
    flush(stdout);
}
printf("\n");
return(response);
}/* watch_triggers */

consume_bytes(port)
int port;
{
    int count;
    unsigned char byte;

    ioctl(port, FIONREAD, &count);

    if (count!=0)
    {
        printf("Unrecognized bytes: \n");
        while(count!=0)
        {
            read(port, &byte, 1);
            printf(" %c %d\n",byte,byte);
            ioctl(port, FIONREAD, &count);
        }
    }
}
/*** Mount-related utility routines ***/

UTsetup_mounts()
{
    /*
      Routine slews the ETC sidereal mounts to their programmed
      (in DEclarations.c) starting positions.
    */

    int i, ra_limit;
    double time;

    printf("\nSetting up the tracking mounts\n");
    time = TImjd_time();

    for(i=0;i<NUM_COSMACS;i++)
    {
        if (UTslew_to(M_data[i].start_ha,i) == 1)
            printf("Slew error on mount number %d\n",i);
        else Slew_time =
            time + (M_data[i].end_ha - M_data[i].start_ha)/(double)(TWO_PIe6);
    }
}

/*UThour_angle*/

UThour_angle(mount)
int mount;
{
    /*
      Returns hour angle of mount in microradians.
      Synchro units decrease going west.
      Assumption: zero point on synchro is at or near nadir
    */

    int ha, outha, angle;

    angle = COSynchro_angle(0);
    printf("\n\t\t\t\t(Synchro reads %d)\n",angle);
    fflush(stdout);
    ha = -(angle - M_data[mount].synchro_angle_of_meridian);

    outha = (int)(mega(ha)/2607.5946);

    outha = -384.26*angle + 3603630;    /* from calibration of 10/21/85 */
    return(outha);
}

/*UTHa_from_ra*/

UTHa_from Ra(ra)
int ra;
{
    /*
      Returns hour angle of any right ascension passed.
    */
Units are microradians.
*/

int ha;
ha = (int)(mega(Tlocal_sidereal_time())) - ra;
if (ha>Pi6) ha -= TWO_Pi6;
return(ha);
}

UTra_of_ha(ha)
int ha;
{
    /*
    * Returns right ascension of any hour angle passed.
    * Units are microradians.
    */
    
    int ra,syn_ha;
double phase;

    ra = (int)(mega(Tlocal_sidereal_time())) - ha;
syn_ha = M_data[0].synchro_angle_of_meridian-Phase_of_synchro_error[0];
    /* [0] should be [mount] */
    phase = ((double)syn_ha)/1303.7973;

    /*
    ra -= Amplitude_of_synchro_error[0]*cos(phase)+Zero_point_of_synchro_error
    */
    /* [0] should be [mount] */
    /*ra += (int)(31482.0*sin(phase));*/

    while (ra < 0) ra += TWO_Pi6;
    while (ra > TWO_Pi6) ra -= TWO_Pi6;

    printf("Synchro angle converts to ra of %d\n",ra);
    fflush(stdout);
    return(ra);
}

UTslew_to(hourangle,mount)
int hourangle,mount;
{
    /*
    Generic slew routine. Slew a given sidereal drive to a passed
    hour angle, provided the hour angle does not exceed the maximum
    allowable hour angle defined in constants.h.
    */
    
    int ha,T,M,C,stepper,diff,time;
    char response[];

    printf("\nPREPARING TO SLEW...\n\n");
    fflush(stdout);
    if ((abs(hourangle))>(MAX_HOURANG))
C

cprintf("Source is presently not viewable\n");
return(1);
}

ha = UHour_angle(mount);
diff = hourangle - ha;
stepper = (int)(.0026075946*(double)diff*(double)STEPS_per_SYNCH);

time = abs(((int)(((float)stepper/133.25)));
cprintf("\nThe slewing motor will move %d steps (- is east)\n",stepper);
cprintf("The mount will be in motion for %d seconds\n",time);
cflush(stdout);
sleep(1);

cOslew_clutch(mount);
if (stepper>0) stepper = 100*(int)(stepper/100.0 + 0.5);
else stepper = -100*(int)(-stepper/100.0 + 0.5);

if (stepper<0)
{
    stepper = abs(stepper);
    T = stepper/10000.0;
    COeastT(T,mount);
    M = (stepper - 10000.0*T)/1000.0;
    COeastM(M,mount);
    C = (stepper - 10000.0*T - 1000.0*M)/100.0;
    COeastC(C,mount); 
}
else
{
    T = stepper/10000.0;
    COwestT(T,mount);
    M = (stepper - 10000.0*T)/1000.0;
    COwestM(M,mount);
    C = (stepper - 10000.0*T - 1000.0*M)/100.0;
    COwestC(C,mount); 
}

COtrack_clutch(mount);
return(0);
} /* UTslew_to */

UTidle_mounds()
{
    /*
     * Routine slews the sidereal drives to the meridian and engages
     * the slewing clutch.
     * Routine is implemented at daybreak and any other time the mounts
     * must be stowed safely.
     */
    int i;

cprintf("\n\nIdling tracking mounts at the meridian\n\n");
flush(stdout);
for(i=0;i<NUM_COSMACS;i++)
{
    UTslew_to(0,i);
    C0slew_clutch(i);
}
MOUNTS_LOCKED = 1;
} /* UTidle_mounts */
/**/ 
* UTdo_astrometry() 
* 
* Calculates accurate astrometric parameters by comparing locations of SAO stars in each FOV with their tabulated coordinates. 
*/

int i;
UTreadout_one_frame();
for(i=0;i<NUM_of_CAMERAS;i++) do_astrometry(i);
Astrometry_time = TImjd_time() + Sdata.astrometry_interval;

/* UTdo_astrometry */
do_astrometry(camera)
int camera;
{
/* 
Uses rough parameters calculated in ASrough_astrometry() to locate SAO catalog stars accurately and use these locations to calculate more precise astrometry parameters 
Routine starts by reading in a frame, then analyzes the frame for astrometric purposes */
int error;
UTfill_astrometry_structure(camera);
print_parameters(camera);
error = ASdo_astrometry(camera); /* returns 1 if rms too large */
print_parameters(camera);
if (error)
{
printf("\nError in astrometry\n");
fflush(stdout);
ASrough_astrometry(camera);
}
else
{
ASrecenter(camera);
ASrough_astrometry(camera);
ASdo_astrometry(camera);
print_parameters(camera);
SAassign_standards(camera);
}
return(error);
} /* do_astrometry */
print_parameters(camera)
int camera;
{
    int i;

    for(i=0;i<6;i++) printf("%.3f \n",C_data[camera].c_to_p_parameters[i]);
    printf("\n");
    for(i=0;i<6;i++) printf("%.3f \n",C_data[camera].p_to_c_parameters[i]);
    fflush(stdout);
    printf("\n");
}

UTfill_astrometry_structure(camera)
int camera;
{
    /*
     * Routine selects SAO standard stars to be used in the calculation
     * of the astrometry parameters. The structure Astr_data_str
     * is filled for later use.
     */
    int x,y,ra,dec,flag,i,counter;
    register struct SAO_entry_str *ses = &SAO_file[camera];

    for(i=0;i<NUM_of_STANDARDS_per_FOV;i++) Ast_data[i].ra = Ast_data[i].dec = 0;
    for(i=0;i<NUM_of_STANDARDS_per_FOV;i++)
        {
            register struct Astr_data_str *ad = &Ast_data[counter];
        if (ses->SAO_star_number[i] != 0)
        {
            flag = 0;
            ra = ad->ra = ses->SAO_coordinates[i].x;
            dec = ad->dec = ses->SAO_coordinates[i].y;
            ad->expected_brightness=ses->SAO_brightness[i];
            ASget_position(camera,ra,dec,&x,&y);
            if ((x>NCOLUMNS*100) || (y>NROWS*100) || (x<0) || (y<0))
            {
                ad->ra = ad->dec = 0;
                flag = 1;
            }
            TRassign_standard(camera,i,(int)(x/100.0),(int)(y/100.0));
            TRsend_standard_subarrays(camera,i);
            ASreturn_centroid(&x,&y);
            if (flag == 0)
TRassign_standard(camera, i, (int)(x/100.0), (int)(y/100.0));
TRsnd_standard_array(camera, i);
show_subarray(i);

if ((x == -1) && (y == -1)) /* if there is a problem in dotyfit */
{
    ad->ra = ad->dec = 0;
}

/*
ad->real_brightness = PHotometry(HSSL_array[0].array, x, y, camera);*/
ad->x = x;
ad->y = y;
}

/* UTfill_astrometry_structure */
UTweather_check()
{
  return(0);
}

UTperipheral_check()
{
  return(0);
}
/** Data storage management utility routines **/

UTinitialize_variables()
{
    int i;
    for (i=0; i<MAX_NUM_of_ACTIVE_FLASHES; i++)
        UTclear_active_entry(i);
    for (i=0; i<MAX_FLASHES_per_NIGHT; i++)
        Data_storage[i].Number_of_exposures = -1;

    Num_stored_flashes = 0;
    Num_of_active_entries = 0;

    for (i=0; i<NUM_of_CAMERAS; i++)
        Archives.good_exposure_counter[i] = 0;
    } /* MUCH more to come! */

UTclear_active_entry(entry)
int entry;
{
    register struct Per_active_entry_str *lff = &Live_flash_file[entry];
    lff->Storage_number = -1;
    lff->Number_of_exposures = -1;
    lff->Update_time = 0.0;
    lff->Live_camera_entry[0].Camera_number = -1;
    lff->Live_camera_entry[1].Camera_number = -1;
}

zero_data_structure()
{
    int i,j,k,l;
    for (i=0; i<MAX_FLASHES_per_NIGHT; i++)
    {
        for (j=0; j<MAX_EXPOSURES_per_FLASH; j++)
        {
            for (k=0; k<=COINCIDENCE; k++)
            {
                zero_array(Data_storage[i].Single_flash_data.Single_exposure_data[j].Data_from_exposure[k].Flash_array.array);
                for (l=0; l<NUM_STANDARDS_per_FLASH; l++)
                {
                    zero_array(Data_storage[i].Single_flash_data.Single_exposure_data.Data_from_exposure[k].Stand_array[l].array);
                }
            }
        }
    }
}

zero_array(array)
unsigned char array[SUBARRAY_SIZE*SUBARRAY_SIZE];
unsigned char *end = array + S_SQUARED;
for (;array<end;array++) *array = 0;
/** Data storage utility routines **/ 

UTstore_flash_data(
{
    /* 
    Routine stores all data taken in the last observation cycle in 
    a disk file coded by the time of storage. The format is 
    modified Julian day. 
    */

    int chan,i;
    char response[30];

    sprintf(response,"data/E%.4f",TImjd_time());
    printf("Storing flash data in file %s\n",response);
    fflush(stdout);
    chan = open(response,2);

    if (chan == -1)
    {
        creat(response,0777);
        chan = open(response,2);
    }

    lseek(chan,0,2);         /* EOF */

    Archives.end_time_of_run = Imageexposuretime.b;
    Archives.numstoredflashes = Numstoredflashes;

    write(chan,&A_str_length,4);
    write(chan,&Archives.A_str_length);
    write(chan,C_data,C_str_length*NUM_of_CAMERAS);
    write(chan,&M_data,M_str_length*NUM_COSMACS);
    write(chan,&T_data,T_str_length);
    write(chan,&S_data,S_str_length);

    for(i=0;i<Num_storedflashes;i++)
    { write(chan,&Data_storage[i],D_str_length); }

    OBSERVED = 0;

    close(chan);

    zero_data_structure();
} /* UTstore_flash_data */ 

UTtape_storage(file)
char file[];
{
    int chan,tape,nbytes,fd,count;
    char buf[10240];

    tape = open("/dev/nrmt8",1);
    if (tape == -1)
    {
printf("Cannot access Cipher\n");
fflush(stdout);
return;
}
count = 1;

chan = open(file,0);
nbytes = 1;
while(nbytes != 0)
{
   nbytes = read(chan,buf,10240);
   write(tape,buf,nbytes);
}
close(chan);
close(tape);
}

UTsave_frames()
{
/*
   Routine allows the optional storage of full-field frames on disk.
   The frame is stored in CCPHOT format, two bytes per pixel, with
   the effects of the ROM mapping mapped out again.
*/

int n_read,i,j,k,HSSL,file,count,nbytes;
unsigned char buf[NCOLUMNS];
struct ccheader hd;
struct tm *time_str;
long seconds;
short outbuf[NCOLUMNS];
char response[14],filename[30];

hd.nph = NCOLUMNS;
hd.npv = NROWS;
hd.p0 = hd.10 = 0;
hd.10act = hd.p0act = 0;
hd.11act = NROWS - 1;
hd.plact = NCOLUMNS - 1;
hd.nbpx = 2 - SQRT_ROM;

hd.texp = S_data.exposure_time/100;
hd.tmsec = 10*(S_data.exposure_time - 100*hd.texp);

seconds = time(0);
time_str = gmtime(&seconds);

hd.idt[0] = (char)(time_str->tm_mon)+1;
hd.idt[1] = (char)(time_str->tm_mday);
hd.idt[2] = (char)(time_str->tm_year);
hd.itm1[0] = (char)(time_str->tm_hour);
hd.itm1[1] = (char)(time_str->tm_min);
hd.itm1[2] = (char)(time_str->tm_sec);
seconds -= 15;
ttime_str = gmtime(&seconds);

hd.itm0[0] = (char)(ttime_str->tm_hour);
hd.itm0[1] = (char)(ttime_str->tm_min);
hd.itm0[2] = (char)(ttime_str->tm_sec);

for (i=0;i<NUM_of_CAMERAS;i++)
{
    printf("Enter name of file for camera \%c's image: ",letters[i]);
    fflush(stdout);
    scanf("%s",response);

    sprintf(filename,"data/%s",response);
    creat(filename,0777);
    file = open(filename,1);

    C_data[i].right_ascension = UTra_of_ah(UThour_angle(0));
    sprintf(hd.lcom,"ETC Camera \%c, centered on (\%.2fh,\%.2fd) ",letters[i],
            C_data[i].right_ascension/261800.0,C_data[i].declination/17453.1);  

    TRsnd_entire_image(i,0);

    HSSL = open("/dev/frame_buf",0);
    write(file,&hd,140);

    for (j=0;j<NROWS;j++)
    {
        read(HSSL,buf,NCOLUMNNS);
        if (SQRT_ROM) write(file,buf,NCOLUMNNS);
        else
        {
            for (k=0;k<NCOLUMNNS;k++) outbuf[k] = (short)rom_mapping[(int)buf[k]]
            write(file,outbuf,2*NCOLUMNNS);
        }
    }

    close(file);
    close(HSSL);
}
/***
Image-related utility routines ***/

UTcalculate_noise(camera)
int camera;
{
    /*
     Routine calculates total noise in a given camera from the sky
     level and readout noise per pixel.
    */

double sky_noise_squared;
if (SQRT_ROM) return;

UTsky_brightness(camera);

sky_noise_squared = (double)(Global_brightness -
    C_data[camera].bias_level)/C_data[camera].gain;

if (sky_noise_squared > 0.0)
    C_data[camera].total_noise = (float)sqrt(sky_noise_squared +
    pow((double)C_data[camera].readout_noise,2.0));
else C_data[camera].total_noise = C_data[camera].readout_noise;

printf("\nCalculated total noise to be %.1f electrons for %c\n",
    C_data[camera].total_noise,letters[camera]);
fflush(stdout);
}

UTaverage_one_frame()
{
    register long sum = 0;
    register int i;
    int chan,n_read,nbytes,j;
    unsigned char buf[NCOLUMNS];

    chan = open("/dev/frame_buf",0);
    for (i=0;i<NROWS;i++)
    {
        n_read = 0;
        while(n_read<NCOLUMNS)
        {
            nbytes = read(chan,&buf[n_read],NCOLUMNS-n_read);
            n_read += nbytes;
        }
        for(j=0;j<NCOLUMNS;j++) sum += buf[j];
    }
    return((int)(sum/NROWS/NCOLUMNS));
}

show_subarray(which_one)
int which_one;
{
    int i,aed;
aed = open("/dev/aed",1);
if (aed == -1) printf("Cannot access AED monitor\n");

for (i=0;i<SUBARRAY_SIZE;i++)
{
  Iseek(aed, 512*(440+i)+50+10*which_one, 0);
  write(aed,&HSSL_array[0].array[SUBARRAY_SIZE*i],SUBARRAY_SIZE);
}

close(aed);

UTadjustoffsets()
{
  /*
   Routine adjusts bias levels in all CCDs so that the average sky+bias level in the CCD lies between the maximum and minimum values defined at the beginning of the file.
   */

  int i, sky, delta, flag, sky_level[NUM_of_CAMERAS], rflag;
  if (SQRT_ROM) return(0);

  for(i=0;i<NUM_of_CAMERAS;i++) Offset[i] = (int)COset_offset(i,0);

  flag = 1;
  rflag = 0;
  while (flag == 1)
  {
    flag = 0;
    UTreadout_one_frame();
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
      UTsky_brightness(i);
      sky_level[i] = (int)(Global_brightness+0.5);
      if ((sky_level[i] > MAX_SKY) || ((sky_level[i] > 0) && (sky_level[i] < MIN_SKY)))
      {
        printf("Adjusting offset of camera %c\n", letters[i]);
        rflag = flag = 1;

        if (sky_level[i] > MAX_SKY) delta = (sky_level[i] - MIN_SKY)/35;
        else delta = (sky_level[i] - MAX_SKY)/35;

        Offset[i] += delta;
        if (Offset[i] < 0) Offset[i] = 0;
        COset_offset(i,Offset[i],0);
      }
      if (sky_level[i] == 0)
      {
        printf("Zeroing offset of camera %c\n", letters[i]);
        flag = 1;
        Offset[i] = 0;
        COset_offset(i,Offset[i],0);
      }
  
  UTsky_calibration();
}
Nov 19 01:07 1985 UTility.c Page 23

UTdetermine_thresholds()
{
/*
   Routine uses the values of total noise in each CCD and the
   threshold criteria defined in Declarations.c or interactively.
*/

int i;

printf("\nDetermining thresholds for all cameras\n");
for (i=0;i<NUM_of_CAMERAS;i++)
{
    if (!SQRT_ROM)
    {
        C_data[i].adu_offset = (unsigned char)(C_data[i].total_noise *
                C_data[i].gain * (float)C_data[i].threshold_offset);
        C_data[i].adu_delta = (unsigned char)(C_data[i].total_noise *
                C_data[i].gain * (float)C_data[i].brightened_delta);
    }
    else
    {
        C_data[i].adu_offset = (unsigned char)C_data[i].threshold_offset;
        C_data[i].adu_delta = (unsigned char)C_data[i].brightened_delta;
    }

    C_data[i].quiescent_difference = 0;
    printf("Threshold level of %c is %d\n",letters[i],C_data[i].adu_offset)
}
fflush(stdout);

UTfind_biasses()
{
/*
   Routine calculates the bias level in the CCD by comparing the
   sky+bias levels in a T and a T+5 second exposure.
*/

int i,skym[NUM_of_CAMERAS],skyn[NUM_of_CAMERAS],brightness,error;
short old_sift,old_exposure,m,n;

if (SORT_ROM) return;
error = 1;
while (error == 1)
{
    error = 0;
    printf("Attempting to determine bias level from two separate exposures\n    fflush(stdout);
old_sift = T_data.sift_time;
old_exposure = S_data.exposure_time;

S_data.exposure_time += 500;
m = S_data.exposure_time;

printf("\nTaking a %d-centisecond exposure\n",m);
fflush(stdout);
UTreadout_one_frame();

for (i=0;i<NUM_of_CAMERAS;i++)
{
    UTsky_brightness(i);
brightness = (int)(Global_brightness + 0.5);
skym[i] = (int)rom_mapping[brightness];
}

S_data.exposure_time -= 500;
n = S_data.exposure_time;

printf("\nTaking a %d centisecond exposure\n",n);
fflush(stdout);
UTreadout_one_frame();

for (i=0;i<NUM_of_CAMERAS;i++)
{
    UTsky_brightness(i);
brightness = (int)(Global_brightness + 0.5);
skyn[i] = (int)rom_mapping[brightness];
}

for (i=0;i<NUM_of_CAMERAS;i++)
{
    if (skym[i] < skyn[i])
    {
        error = 1;
        printf("Bias level error in camera %c\n",letters[i]);
        fflush(stdout);
    }
    C_data[i].bias_level = (n*skym[i] - m*skyn[i])/(float)(n-m);
    printf("Bias level of %c is %d\n",letters[i],C_data[i].bias_level);
    fflush(stdout);
}

T_data.sift_time = old_sift;
S_data.exposure_time = old_exposure;
}

UTsky_brightness(camera)
int camera;
{
    /*
        Routine calculates the average of the sky+bias level in a small
    */
}
square patch at the physical center of the CCD.
The value of Global brightness is set to the average in ADU.
The value of Sky sigma is set to the standard deviation in CDU.

```c
*/

int i,j,csum,asum,counter,size;
unsigned char array[SKY_ARRAY_SIZE*SKY_ARRAY_SIZE];
short sa[SKY_ARRAY_SIZE*SKY_ARRAY_SIZE],sarray[SKY_ARRAY_SIZE];
float fasum,fcsum,aaverage,caverage;

size = SKY_ARRAY_SIZE;
TRrandom_array(camera,NCOLUMNS/2-25,NROWS/2,size,array);

j = size*size;
csum = asum = 0;
while(--j != -1)
{
    sa[j] = rom_mapping((int)array[j]);
    asum += sa[j];
    csum += (int)array[j];
}

aaverage = (float)asum/(float)size/(float)size;
caverage = (float)csum/(float)size/(float)size;

j = size*size;
fcsun = fasum = 0.0;
while(--j != -1)
{
    fasum += (aaverage-sa[j])*(aaverage-sa[j]);
    fcsum += (caverage-(int)array[j])*(caverage-(int)array[j]);
}

Sky_sigma = sqrt(fcsum/((float)(size*size)-1.0));
printf("Sky sigma is %.2f CDU...Average is %.2f ADU",Sky_sigma,aaverage);
printf("", %.2f CDU
",caverage);

j = size*size;
counter = 0;
asum = csum = 0;
while(--j != -1)
{
    if (fabs(sa[j]-aaverage) < 4.0*Sky_sigma)
    {
        asum += sa[j];
        counter++;
    }
}

if (counter != 0)
{
    aaverage = (float)asum/(float)counter;
    Global_brightness = aaverage;
}
else
```
Nov 19 01:07 1985 UTility.c Page 26

{
    printf("Counter is zero!\n");
    Global_brightness = aaverage;
}
}

eat_loose_bytes()
{
    int i,j,count;
    char buf[];

    sleep(1);
    for (i=0;i<NUM_of_CAMERAS;i++)
    {
        for (j=0;j<MAX_TRIES;j++)
        {
            ioctl(Trig_port[i],FIONREAD,&count);
            if (count != 0) break;
        }

        while (count != 0)
        {
            read(Trig_port[i],buf,count);
            ioctl(Trig_port[i],FIONREAD,&count);
        }
    }
}

UTget_readout_time()
{
    /* Routine calculates and returns readout time in centiseconds */

    return(500);
}

UTset_frame_timer()
{
    T1set_frame_timer(2500*(int)S_data.exposure_time);
}
/*
   This module contains all weather-related subroutines.
   More subroutines will be added when the ADC is fully implemented.
*/

#include <stdio.h>
#include <sgtty.h>

#include "constants.h"
#include "extern.h"

#define CLOUD_CRITERION 1.5

extern UTreadout_one_frame(); UTsky_brightness();

WEclouds()
{
  /*
   Routine uses standard deviation of an image subarray to determine
   whether any stars are visible.
   */

  int i;
  float sigma_average = 0.0;

  UTreadout_one_frame();
  for (i = 0; i < NUM_of_CAMERAS; i++)
  {
    UTsky_brightness(i);
    sigma_average += Sky_sigma;
  }

  sigma_average = sigma_average / NUM_of_CAMERAS;
  printf("Average sky sigma = %.2f\n", sigma_average);
  if (sigma_average < CLOUD_CRITERION) return (1);
  else return (0);
}
/*
This module allows for the automatic setting of the day, date and
time via the Heathkit GC-1000 'Most Accurate Clock'.
Routine was originally written by George Mitsuoka, and was modified
by Ed Ajhar to take the delay in setting the date into account.
*/

#include <stdio.h>
#include <sys/ioctl.h>
#define WWV="/dev/wwv"
#define RW 2
#define ADJUSTMENT 3
/* Number of seconds to add to time. */

struct time_str
{
    char ten_hours, hours;
    char colon_1;
    char ten_minutes, minutes;
    char colon_2;
    char ten_seconds, seconds, period, tenths_seconds;
    char space_1;
    char A_or_P_or_space, M_or_space;
    char space_2, space_3;
    char ten_months, months;
    char slash_1;
    char ten_days, days;
    char slash_2;
    char ten_years, years;
    char terminator;
    char null;
} wwv_time;

char date_arg[20];
adjust_time ();

WWv() /* sync the computer time to wwv */
{
    int  wwv_receiver, status;

    if ((wwv_receiver = open(WWV,RW)) == -1)
    {
        fprintf(stderr,"stwwv: can’t open %s\n",WWV);
        exit(1);
    }
    if (ioctl(wwv_receiver, EXT_RESET))  /* tell receiver to send time */
    {
        fprintf(stderr,"stwwv: bad ioctl\n");
        exit(1);
    }
    sleep(2);  /* wait for wwv_receiver to send time */
    if (read(wwv_receiver,(char *) &wwv_time,sizeof(struct time_str)) == -1)
    {
        fprintf(stderr,"stwwv: read error\n");
        exit(1);
    }
}
adjust_time();

```c
printf (date_arg, "%c%c%c%c%c%c.%c%c", wtv_time.ten_months, wtv_time.months;
        wtv_time.ten_days, wtv_time.days, wtv_time.ten_hours, wtv_time.hours;
        wtv_time.ten_minutes, wtv_time.minutes, wtv_time.ten_seconds, wtv_time_tir

if (fork() == 0)
{
    printf("%s\n", (char *) &wtv_time);
    exec("/bin/date","/bin/date", date_arg, 0); /* set the correct time */
    fprintf(stderr,"wtv: exec of /bin/date failed\n");
    exit(1);
}
else
{
    wait(&status);
}
close(WWV);
```

adjust_time()
{ /* Routine adjusts time for the delay incurred during setting date */

    int month, day, hour, minute, second;
    char *m[2], *d[2], *h[2], *mi[2], *s[2];

    strftime(m, "%c", wtv_time.ten_months, wtv_time.months);
    strftime(d, "%c", wtv_time.ten_days, wtv_time.days);
    strftime(h, "%c", wtv_time.ten_hours, wtv_time.hours);
    strftime(mi, "%c", wtv_time.ten_minutes, wtv_time.minutes);
    strftime(s, "%c", wtv_time.ten_seconds, wtv_time.seconds);

    month = atoi(m);
    day = atoi(d);
    hour = atoi(h);
    minute = atoi(mi);
    second = atoi(s);

    second += ADJUSTMENT;

    if (second > 59)
    {
        second -= 60;
        ++minute;

        if (minute > 59)
        {
            minute -= 60;
            ++hour;

            if (hour > 23)
            {
```
hour -= 24;
++day;

if (day>28 && month==2)
{
    day -= 28;
    ++month;
    if (month > 12) month -= 12;
}
else if (day>30 &&
    (month == 4 || month == 6 || month == 9 || month ==
    {    
        day -= 30;
        ++month;
        if (month > 12) month -= 12;
    }
    else if (day > 31)
    {
        day -= 31;
        ++month;
        if (month > 12) month -= 12;
    }
}

sprintf (&wwv_time.tenmonths, "%d", (int) month/10);
sprintf (&wwv_time.months, "%d", (int) month%10);

sprintf (&wwv_time.tendays, "%d", (int) day/10);
sprintf (&wwv_time.days, "%d", (int) day%10);

sprintf (&wwv_time.tenhours, "%d", (int) hour/10);
sprintf (&wwv_time.hours, "%d", (int) hour%10);

sprintf (&wwv_time.tenminutes, "%d", (int) minute/10);
sprintf (&wwv_time.minutes, "%d", (int) minute%10);

sprintf (&wwv_time.tenseconds, "%d", (int) second/10);
sprintf (&wwv_time.seconds, "%d", (int) second%10);

return;
```c
#include <stdio.h>
#include <sgtty.h>
#include "constants.h"
#include "extern.h"

#define TRUE 1
#define FALSE 0
#define NOPAR 0
#define MARK 1
#define SPACE 2
#define ODD 3
#define EVEN 4
#define SIZ 100
#define ESC ((char)0x1B)

COdownload(cosmac,file_selector)
int cosmac,file_selector;
{
  /*
   Routine downloads a COSMAC routine defined by 'file_selector'
   Returns 0 if operating system is properly downloaded
   1 if not
   
   Routine is a modification (and extreme reduction) of Steve Rosenthal's
   'call_cos' used for Cosmac I/O
   */

  char in_char[1],byte;  /* input character */
  int c_read;        /* input character counter */
  int input_file,port,child_id,child_count,count,sign,nbytes,flag;
  char child_buf[SIZ];

  /* comment processing */
  int in_comment=FALSE;  /* currently trashing a comment?? */

  static char comm_start[] = "*";  /* comment start delimiter */
  static char comm_end[] = "\n";  /* comment end delimiter */

  char *comm_ptr;  /* Used in comment processing from
                   input file */
  char *finger;

  port = Cosmac_port[cosmac];

  if (file_selector != 0) COexit_hostcom(port);  /* take it out of hostcom */

  printf("Downloading ");
  fflush(stdout);
  if (file_selector == 0)


```c

[ input_file = open("Q_opsys",0);
 printf("operating system\n\n");
 ]
else if (file_selector==1)
 [ input_file = open("Sequel",0);
 printf("sequel routine\n\n");
 ]
else if (file_selector==2)
 [ input_file = open("Setup",0);
 printf("setup file\n\n");
 ]
fflush(stdout);
if (input_file == -1)
 [ printf("Error opening input file(%d)\n\n",file_selector);
 return(-1);
 ]

comm_ptr = comm_start;    /* set up for comment processing */
in_comment = FALSE;

/* spawn child process for reading download echoes */

child_id = fork();
if (child_id == 0)    /* in child */
 [ printf("In child!\n");
 while(1)
   [ child_count = read(port,child_buf,SLZ);
     write(1,child_buf,child_count);
   ]
 ]    /* end of child */
sleep(10);
while (TRUE)
 |
c_read = read(input_file, in_char, 1);    /* 1 char at a time */
if (c_read == 0)    /* EOF */
 |
   [ printf("Closing input file\n");
     close(input_file);
     sleep(3);
     kill(child_id,9);
   ]
 |
sign = 0;
swallow echoes(port);
write(port,"\n\r\n\r",4);
sleep(1);
ioctl(port,FIONREAD,&count);

```
flag = 0;
while(flag == 0)
{
    sleep(1);
    ioctl(port,FIONREAD,&count);
    while(count!=0)
    {
        read(port,&byte,1);
        if (byte == '$') flag = 1;
        ioctl(port,FIONREAD,&count);
    }
}

/* Try to put it into hostcom */

nbytes = write(port,"hostcom \n\r",10);

sleep(1);
ioclt(port,FIONREAD,&count);
while(count!=0)  /* eat up echoed hostcom (if there is one)*/
{
    nbytes = read(port,&byte,1);
    if (byte == '?')  /* then operating system is not loaded */
    {
        sign = 1;
        printf("Error in download\n\n");
    }
    ioctl(port,FIONREAD,&count);
}

swallow_echoes(port);
if ((file_selector == 2) && (sign == 0)) COSinit(cosmac);
return(sign);
else
{
    if (*inchar) /* looking for comment start */
    {
        if (*in_char == *comm_ptr)
        {  /* matches end string so far. */
            comm_ptr++;
            if (*comm_ptr == '\0')     /* found end of comment */
            {
                in_comment = FALSE;
                comm_ptr = comm_start;
            }
        } else /* no match. Try again from start of comm_end */
        {
            comm_ptr = comm_end;
        }
    }

} else /* looking for comment start */
Nov 19 01:05 1985 COsmac_10.c Page 4

```
[  
  comm_ptr++;  
  if (*comm_ptr == '\0') /* found a comment! */  
  {  
    in_comment = TRUE;  
    comm_ptr = comm_end;  
  }  
}  
else /* write out chars we've been looking at */  
{  
  finger = comm_start;  
  while (finger != comm_ptr)  
  {  
    write(port,finger,1);  
    finger++;  
  }  
  write(port,in_char,1);  
  comm_ptr = comm_start;  
}  
] /* end */

COreadout_CCDS()  
{  
/* Routine reads out all CCDs */  
  int i;  
  for (i=0; i<NUM_COSMACS; i++)  
  {  
    if (Cos_status[i]!=-1)  
    {  
      write(Cosmac_port[i],"seqgo ; ",8);  
    }  
  }  
}

COhousekeeping(cosmac)  
int cosmac;  
{  
/* Routine collects all housekeeping data from the COSMACs */  
  if (Cos_status[cosmac]!=1)  
  {  
    COtemperature(cosmac,Housekeeping[cosmac].temp);  
    current(cosmac,Housekeeping[cosmac].current);  
    t_setting(cosmac,Housekeeping[cosmac].t_setting);  
    c_setting(cosmac,&Housekeeping[cosmac].c_setting);  
  }  
}  
```
CO_flush_CCDs()
{
    /* Routine flushes all CCDs. Routine returns only after flush is complete */
    int i, nbytes;
    char byte;

    for(i=0; i<NUM_COSMACS; i++)
        write(Cosmac_port[i], "go go go go go go 1 \nemit ; ", 28);
    for(i=0; i<NUM_COSMACS; i++)
        nbytes = read(Cosmac_port[i], &byte, 1);
    return(1);
}

CO_temperature(cosmac, temp)
int cosmac;
unsigned char temp[4];
{
    /* Routine collects CCD temperatures from all cameras */
    int i;
    write(Cosmac_port[cosmac], "stemps ; ", 9);

    for(i=0; i<4; i++)
        read(Cosmac_port[cosmac], &temp[i], 1);
    printf("%d %d %d %d\n", temp[0], temp[1], temp[2], temp[3]);
}

current(cosmac, current)
int cosmac;
unsigned char current[4];
{
    /* Routine collects the heater currents from all cameras */
    int port, i;

    printf("In current(%d)\n", cosmac);
    port = Cosmac_port[cosmac];
    write(port, "scurrents ; ", 12);

    for(i=0; i<4; i++)
        read(port, &current[i], 1);
}

t_settins(cosmac, t_setting)
int cosmac;
unsigned char t_setting[4];
{
    /* Routine collects the temperature (thermostat) settings from all cameras */
    int port, i;

    printf("In t_settins(%d)\n", cosmac);
    port = Cosmac_port[cosmac];
    write(port, "st_settings ; ", 14);
for(i=0;i<4;i++) read(port,&t_setting[i],1);
}
c_setting(cosmac,c_setting)
int cosmac;
unsigned char *c_setting;
{
 /*************************************************************************/
 int port,nbytes;
 printf("In c_setting(%d)\n",cosmac);
 port = Cosmac_port[cosmac];
 write(port,"sc_setting ",13);
 nbytes = read(port,c_setting,1);
}
C0r0use_c0smac(cosmac)
int cosmac;
{
 /*************************************************************************
 Routine attempts to determine the state of a COSMAC from its
 responses to various queries.
 Routine returns 0 if the COSMAC is "dead" (no response)
 " " 1 if the COSMAC operating system is loaded
 " " 2 if something
 " " 3 if the COSMAC is alive but unloaded.
 /*************************************************************************/
 int port,nbytes,count,response;
 char byte;
 port = Cosmac_port[cosmac];
 printf("In C0r0use_c0smac\n");
 swallow_echoes(port);
 nbytes = write(port,"
\n\nquit ; \n\r",13);
 sleep(3);
 ioctl(port,FIONREAD,&count);
 printf("ioctl count is %d\n",count);
 if (count == 0) /* then the system is either dead or VERY confused */
 response = 0;
 else
 response = 2;
 while(count!=0)
 {
 nbytes = read(port,&byte,1);
 printf("%c",byte);
if ((byte == '$') && (response != 3)) response = 1;
if (byte == '?') response = 3;
ioctl(port,FIONREAD,&count);
if ((count == 0) && (response == 1))
{
    sleep(3);
    ioctl(port,FIONREAD,&count);
}
} /* while */
} /* else if count */

printf("After all that, response is %d\n",response);

if (response == 2) swallow_echoes(port);
return(response);
}

COcheck_full_load(cosmac)
int cosmac;
{
    /*
    Routine is similar to COroute_cosmac, but is attempting to
determine whether the Sequel routine and Setup file have been
downloaded.  
Routine returns 0 if the COSMAC is "dead" (no response)  
" " 1 if Sequel and Setup files are loaded 
" " 2 if something 
" " 3 if the Sequel and Setup files are unloaded.
*/

int port,nbytes,count,response;
char byte;

port = Cosmac_port[cosmac];
printf("In COcheck_full_load\n");

swallow_echoes(port);

nbytes = write(port,"\n\rccheckload\n\r",13);
sleep(3);
ioctl(port,FIONREAD,&count);
printf("ioctl count is %d\n",count);

if (count == 0) /* then the system is either dead or VERY confused */
    response = 0;
else
{
    response = 2;

    while(count!=0)
    {
        nbytes = read(port,&byte,1);
        printf("%c",byte);
        if ((byte == '$') && (response != 3)) response = 1;
        if (byte == '?') response = 3;
    }
ioctl(port,FIONREAD,&count);
if ((count == 0) && (response == 1)) {
    sleep(3);
    ioctl(port,FIONREAD,&count);
}
} /* while */
} /* else if count */

printf("After all that, response is %d\n",response);
return(response);

COenter_hostcom(port)
int port;
{ /* Routine puts the COSMAC into hostcom */
    write(port,\"\n\r\n\r\nquit ; \n\r\r",13);
    swallow echoes(port);
    write(port,"hostcom\n\r",9);
    swallow echoes(port);
}

COexit_hostcom(port)
int port;
{ /* Routine takes the COSMAC out of hostcom */
    write(port,\"\n\r\n\r\nquit ; \n\r\r",13);
    swallow echoes(port);
}

swallow echoes(port)
int port;
{ /* Routine is used to read in any responses from the COSMAC */
    int count,nbytes;
    char byte;
    sleep(2);
    ioctl(port,FIONREAD,&count);

    while(count!=0)
    {
        nbytes = read(port,&byte,1);
        ioctl(port,FIONREAD,&count);
    }
}
COSlew_clutch(mount)
int mount;
{
    /* Routine engages ETC slewing clutch */
    printf("\nEngaging slewing clutch\n\n");
    if(Cos_status[mount] != -1) write(Cosmac_port[mount],"slewon \n",9);
    TRACKING = 0;
}

COTrack_clutch(mount)
int mount;
{
    /* Routine engages ETC tracking clutch */
    printf("\nEngaging tracking clutch\n\n");
    if(Cos_status[mount] != -1) write(Cosmac_port[mount],"trackon \n",10);
    TRACKING = 1;
    MOUNTS_LOCKED = 0;
}

COSynchro_angle(mount)
int mount;
{
    /*
     * Returns synchro angle of mount, in synchro units
     */
    unsigned char b1,b2;
    short ha;
    int count;
    if(Cos_status[mount] != -1)
    {
        write(Cosmac_port[mount],"habyte \n",9);
        read(Cosmac_port[mount],&b1,2);
        sleep(1);
        ioctl(Cosmac_port[mount],FIONREAD,&count);
        if (count) read(Cosmac_port[mount],&b2,2);
        else
        {
            write(Cosmac_port[mount],"habyte \n",9);
            read(Cosmac_port[mount],&b1,2);
            read(Cosmac_port[mount],&b2,2);
        }
        ha = (int)(b1<<8) + (int)b2;
    }
    else ha = -1;

    if (ha == 25441)
    {
        printf("Synchro is wedged!\n");
        return(-1);
    }
return((int)ha);
} /* COsynchro_angle */

COeastT(number,mount)
int number,mount;
{
    /* Routine slews tracking mount 10000 steps east */

    int i,nbytes;
    char sign;

    if (Cos_status[mount] != -1)
    {
        for (i=0;i<number;i++)
        {
            write(Cosmac_port[mount],"eastT ; ",8);
            nbytes = read(Cosmac_port[mount],&sign,1);
        }
    }
}

COeastM(number,mount)
int number,mount;
{
    /* Routine slews tracking mount 1000 steps east */

    int i,nbytes;
    char sign;

    if (Cos_status[mount] != -1)
    {
        for (i=0;i<number;i++)
        {
            write(Cosmac_port[mount],"eastM ; ",8);
            nbytes = read(Cosmac_port[mount],&sign,1);
        }
    }
}

COeastC(number,mount)
int number,mount;
{
    /* Routine slews tracking mount 100 steps east */

    int i,nbytes;
    char sign;

    if (Cos_status[mount] != -1)
    {
        for (i=0;i<number;i++)
        {
            write(Cosmac_port[mount],"eastC ; ",8);
            nbytes = read(Cosmac_port[mount],&sign,1);
        }
    }
COwestT(number, mount)
int number, mount;
{
/* Routine slews tracking mount 10000 steps west */

int i, nbytes;
char sign;
if (Cos_status[mount] != -1)
{
  for (i=0; i<number; i++)
  {
    write(Cosmac_port[mount], "westT ; ", 8);
    nbytes = read(Cosmac_port[mount], &sign, 1);
  }
}

COwestM(number, mount)
int number, mount;
{
/* Routine slews tracking mount 1000 steps west */

int i, nbytes;
char sign;
if (Cos_status[mount] != -1)
{
  for (i=0; i<number; i++)
  {
    write(Cosmac_port[mount], "westM ; ", 8);
    nbytes = read(Cosmac_port[mount], &sign, 1);
  }
}

COwestC(number, mount)
int number, mount;
{
/* Routine slews tracking mount 100 steps west */

int i, nbytes;
char sign;
if (Cos_status[mount] != -1)
{
  for (i=0; i<number; i++)
  {
    write(Cosmac_port[mount], "westC ; ", 8);
    nbytes = read(Cosmac_port[mount], &sign, 1);
  }
}

COsinit(cosmac)
int cosmac;
{
    /* Routine initializes COSMAC state via COSMAC command 'cosinitall' */
    int nbytes;
    nbytes = write(Cosmac_port[cosmac], "cosinitall \", 13);
}

COset_offset(camera, offset, cosmac)
int camera, offset, cosmac;
{
    /* Routine allows the setting of any offset of any camera to any level */
    int port = Cosmac_port[cosmac];
    char string[5];
    sprintf(string, "%d \", offset);
    write(port, string, strlen(string));
    write(port, "offs", 4);
    sprintf(string, "%d", camera+2); /* KLUGE FOR CAMERAS B & C */
    write(port, string, 1);
    write(port, " \r", 4);
}

unsigned char COget_offset(camera, cosmac)
int camera, cosmac;
{
    /* Routine returns offset DAC value of any camera */
    unsigned char offset;
    int port = Cosmac_port[cosmac];
    char string[5];
    write(port, "poff", 4);
    sprintf(string, "%d", camera+2); /* KLUGE FOR CAMERAS B & C */
    write(port, string, strlen(string));
    write(port, " \r", 4);
    read(port, &offset, 1);
    return(offset);
}

COsend_break(cosmac)
int cosmac;
{
    /* Routine attempts to send the COSMAC a software reset, via two
        ioctl commands developed by John Doty */
    /*
int port, flag;
unsigned char byte;

port = Cosmac_port[cosmac];

printf("Before system\n");
system("cosbrk");
printf("After system\n");

/*flag = ioctl(port,TIOCSBRK,NULL);
printf("First flag was %d\n",flag);

flag = ioctl(port,TIOCCBRK,NULL);
printf("Second flag was %d\n",flag);*/

exit(1);

write(port,"\n\r",2);
swallow_echoes(port);
byte = 27;
write(port,&byte,1);
write(port,"\n\r",2);
swallow_echoes(port); 
}
This module contains all subroutines involved in communication with the Trigger processors, including downloading and parameter setting. Details of the Trigger software can be found in the Trigger documentation.

#include <sgtty.h>
#include <stdio.h>
#include <ctype.h>
#include "constants.h"
#include "extern.h"

#define MAX_ERR 10

struct sgttyb port_settings;

int HSSL_port;
char TRnull_query(trigger)
    int trigger;
    {
        /*
         * Routine queries Trigger for null response (to check if Trigger is alive)
         */

        int nbytes;
        char byte;

        write_trigger(trigger,1);
        sleep(1);
        ioctl(Trig_port[trigger],FIONREAD,&nbytes);

        if (nbytes != 0)
            {nbytes = read(Trig_port[trigger],&byte,1);
             return(byte);
            }
        else
            return(-1);
    }

TRlong_query(trigger)
int trigger;
{
    /* I'm not sure what this routine does! */

    char byte;
    write_trigger(trigger,1);
    read(Trig_port[trigger],&byte,1);
    return;
}

char TRversion(trigger)
int trigger;
{
    /* Routine queries Trigger for version number */

    int nbytes;
    char byte;
    write_trigger(trigger,2);
    nbytes = read(Trig_port[trigger],&byte,1);
    return(byte);
}

TR_ETRM(trigger)
int trigger;
{
    /* Routine returns Trigger to ETRM (not used) */

write_trigger(trigger,3);

TR_end_queries()
{
    /* Routine sends all Triggers an 'end-queries' command, preparing them to receive data from the CCDs. */
    int i;
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        write_trigger(i,4);
    }
}

TR_test_mode_on(trigger)
{
    /* Routine sends Triggers into test mode (not used) */
    write_trigger(trigger,9);
}

TR_test_mode_off(trigger)
{
    /* Routine takes Triggers out of test mode (not used) */
    write_trigger(trigger,10);
}

TR_sift_against_old()
{
    /* Routine commands Triggers to sift against old frame */
    int i;
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
        write_trigger(i,11);
    }
}

TR_disc_current_frame()
{
    /* Routine commands Triggers to discard current frame */
    int i;
    for(i=0;i<NUM_of_CAMERAS;i++)
    {
    }
void TRdsc_old_frame()
{
    /* Routine commands Triggers to discard old frame */

    int i;
    for (i = 0; i < NUM_of_CAMERAS; i++)
    {
        write_trigger(i, 12);
    }

    write_trigger(i, 13);
}
TRset_sift_time()
{
    /* Routine sets sift timer in Triggers to be T_data.sift_time */

    int i;
    for (i=0; i<NUM_of_CAMERAS; i++)
    {
        write_trigger(i, 5);
        write(Trig_port[i], &T_data.sift_time, 2);
    }
}

TRset_cosmic_sensitivity()
{
    /* Routine sets Trigger cosmic-ray sensitivity to T_data.cosmic_sensitivity */

    int i;
    for (i=0; i<NUM_of_CAMERAS; i++)
    {
        write_trigger(i, 6);
        write_trigger(i, T_data.cosmic_sensitivity);
    }
}

TRset_brightened_delta()
{
    /* Routine sends value of brightened_delta to Triggers */

    int i;
    for (i=0; i<NUM_of_CAMERAS; i++)
    {
        write_trigger(i, 8);
        write_trigger(i, C_data[i].adu_delta);
    }
}

TRset_flash_subarray_size()
{
    /* Routine sends value of subarray size to Triggers */

    int i;
    for (i=0; i<NUM_of_CAMERAS; i++)
    {
        write_trigger(i, 18);
        write_trigger(i, (unsigned char)SUBARRAY_SIZE);
        write_trigger(i, (unsigned char)SUBARRAY_SIZE);
    }
}
TRset__standard_subarray_size()
{
  /* Routine sends value of subarray size around standard stars to Triggers */
  int i;
  for(i=0;i<NUM_of_CAMERAS;i++)
  {
    write_trigger(i,19);
    write_trigger(i,(unsigned char)SUBARRAY_SIZE);
    write_trigger(i,(unsigned char)SUBARRAY_SIZE);
  }
}

TRassign__standard(trigger,number,x,y)
int trigger,number,x,y;
{
  /* Routine assigns a reference number to a specific location on a CCD */
  write_trigger(trigger,20);
  write_trigger(trigger,(unsigned char)number);
  write_2_bytes(trigger,(unsigned short)x);
  write_2_bytes(trigger,(unsigned short)y);
}

TRcalculate__threshold_from_offset(trigger,offset)
int trigger;
unsigned char offset;
{
  /* Routine commands a Trigger to calculate a threshold frame from the current frame and the value of threshold_offset */
  unsigned char byte;
  write_trigger(trigger,7);
  write_trigger(trigger,offset);
}
TRsnd_entire_image(trigger, frame_number)
int trigger;
unsigned char frame_number;
{
    /* Routine requests Trigger to send an entire image to frame_buf */
    int chan;

    HSSL_port = open("/dev/hssl", O);

    printf("\nSending entire image to GC-HSSL\n");
    write_trigger(trigger, 14);
    write_trigger(trigger, frame_number);
    read_frame_from_HSSL();
}

read_frame_from_HSSL() /* puts data into frame_buf */
{
    /* Routine reads image data from the HSSL */
    register int i, nbytes, n_read;
    unsigned char buf[NCOLUMNS];

    for (i = 0; i < NROWS; i++)
    {
        n_read = 0;
        while (n_read != NCOLUMNS)
        {
            nbytes = read(HSSL_port, &buf[n_read], NCOLUMNS - n_read);
            n_read += nbytes;
        } /* while */
    } /* for i */
    close(HSSL_port);
} /* end */

TRsnd_row_vector(trigger)
int trigger;
{
    /* Routine requests Trigger to send the row vector used in the threshold filter */
    /* The definition of row vector is the Overseer definition; i.e., the result of the median filtering of columns. The Overseer row vector, being the first to be filtered, is made up of unsigned characters; the Overseer column vector is char. */

    HSSL_port = open("/dev/hssl", O);
    write_trigger(trigger, 15);
    read_row_from_HSSL(trigger);
}

TRsnd_column_vector(trigger)
int trigger;
{
    /*
    Routine requests Trigger to send the column vector used in the
    threshold frame. The definition of column vector is the Overseer
definition; i.e., the result of the median filtering of rows.
The Overseer column vector, being the first to be filtered, is
made up of unsigned characters; the Overseer row vector is char.
    */

    HSSL_port = open("/dev/hssl",0);
    write_trigger(trigger,16);
    read_column_from_HSSL(trigger);
}

TRget_median_data(triger)
int trigger;
{
    /* Routine requests Trigger to send row and column vectors */

    TRsnd_column_vector(trigger);
    TRsnd_row_vector(trigger);
}

read_column_from_HSSL(trigger)
int trigger;
{
    /* Routine reads transmitted column vector from HSSL */

    register int nbytes,n_read;

    n_read = 0;
    while(n_read<NROWS)
    {
        nbytes =
            read(HSSL_port,&Column_threshold[trigger*NROWS+n_read],NROWS-n_read);
        n_read += nbytes;
    }

close(HSSL_port);
}

read_row_from_HSSL(trigger)
int trigger;
{
    /* Routine reads transmitted row vector from HSSL */

    register int nbytes,n_read;

    n_read = 0;
    while(n_read<NCOLUMNS)
    {
        nbytes =
            read(HSSL_port,&Row_threshold[trigger*NCOLUMNS+n_read],NCOLUMNS-n_read);
        n_read += nbytes;
    }
close(HSSL_port);

TRsnd_flash_subarrays(trigger, flash_number)
int trigger;
unsigned char flash_number;
{
    /* Routine requests Trigger to send subarrays from flash #flash_number */

    unsigned char bit_pattern;
    HSSL_port = open("/dev/hssl", 0);
    bit_pattern = 128 + flash_number;
    write_trigger(trigger, bit_pattern);
    read_subarrays_from_HSSL(3);
}

TRsnd_standard_subarrays(trigger, standard_number)
int trigger;
unsigned char standard_number;
{
    /* Routine asks Trigger to send subarrays from standard #standard_number */

    unsigned char bit_pattern;
    HSSL_port = open("/dev/hssl", 0);
    bit_pattern = 192 + standard_number;
    write_trigger(trigger, bit_pattern);
    read_subarrays_from_HSSL(3);
}

TRsnd_array(trigger, where_x, where_y, width, height)
int trigger;
unsigned short where_x;
unsigned short where_y;
unsigned char width;
unsigned char height;
{
    /* Routine requests the transfer of an image array of random size */

    HSSL_port = open("/dev/hssl", 0);
    write_trigger(trigger, 17);
    write_2_bytes(trigger, where_x);
    write_2_bytes(trigger, where_y);
    write_trigger(trigger, width);
    write_trigger(trigger, height);
    read_subarrays_from_HSSL(3);
}

read_subarrays_from_HSSL(how_many)
int how_many;
{
    /* Generic subarray-reading routine */

    register int i, nbytes, n_read, j;
}
for(i=0;i<how_many;i++)
{
    n_read = 0;
    while (n_read !< SUBARRAY_SIZE*SUBARRAY_SIZE)
    {
        nbytes = read(HSSL_port,&HSSL_array[i].array[n_read],
                       SUBARRAY_SIZE*SUBARRAY_SIZE - n_read);
        n_read += nbytes;
    } while (n_read < SUBARRAY_SIZE*SUBARRAY_SIZE)
} /* for i */
close(HSSL_port);
} /* end */

TRrandom_array(trigger,where_x,where_y,size,array)
int trigger,where_x,where_y,size;
unsigned char array[];
{ /* Routine requests the transfer of a square image subarray of random size */
    register int n_read,nbytes;
    int count;

    HSSL_port = open("/dev/hssl",0);
    write_trigger(trigger,17);
    write_2_bytes(trigger,(unsigned short)where_x);
    write_2_bytes(trigger,(unsigned short)where_y);
    write_trigger(trigger,(unsigned char)size);
    write_trigger(trigger,(unsigned char)size);
    n_read = 0;
    while (n_read != size*size)
    {
        nbytes = read(HSSL_port,&array[n_read],size*size - n_read);
        n_read += nbytes;
    }
    close(HSSL_port);
} /* TRrandom_array */
write_trigger(trigger,byte)
int trigger;
unsigned char byte;
{
    /* Generic write-to-trigger routine */
    int nbytes;
    nbytes = write(Trig_port[trigger],&byte,1);
}

write_2_bytes(trigger,two_bytes)
int trigger;
unsigned short two_bytes;
{
    /* Generic write-to-trigger routine, at two bytes per write */
    int nbytes;
    nbytes = write(Trig_port[trigger],&two_bytes,2);
}
TRig_load_trigger(trigger)
int trigger;
{
    /* Routine downloads trigger software */
    int tchan, error;
    tchan = open("../trig/trigger.bin", 0);
    lseek(tchan, 0, 0);
    error = write_from_file(Trig_port[trigger], tchan, 0x800);
    if (error != 0) Trig_status[trigger] = -1;
    else Trig_status[trigger] = 1;
    close(tchan);
    return(error);
}

TRig_go_trigger(trigger)
int trigger;
{
    /* Routine executes trigger software */
    int error;
    error = 0;
    if (Trig_status[trigger] != -1)
        error = go(Trig_port[trigger], 0x800);
    if (error == 1) error = go(Trig_port[trigger], 0x800);
    if (error == 1) Trig_status[trigger] = -1;
}

gO(port, addr)
int port, addr;
{
    /*
    effect
    commands ETRM to begin execution at specified
    address.
    signals
    0: no error; execution beginning
    1: bad address
    2: address xfer error
    NOTE: This routine does do up to MAX_ERR retries.
    */
    char cmd_char, response;
    int err_cnt = 0;
    int count, flag;
clear(port);
cmd_char = 'g';

while (err_cnt < MAX_ERR)
{
    write(port,&cmd_char,1);
    send_addr(port_addr);
    sleep(1);
    /* send address and complement */
    /* to give response a chance to get back */
    /* get response */
    /* kluge ioctl response added by rkv 10/20/84 */
ioctl(port,FIONREAD,&count);
    flag = 0;
    if(count != 0)
    {
        read(port,&response,1);
        if (response == 'y') flag = 1;  /* using flag instead of break */
        if (response == 'n') flag = 1;  /* because of if brackets */
        err_cnt++;
        /* if response was anything else */
    }
    else
        err_cnt++;
    if (flag == 1) break;
}
if (err_cnt >= MAX_ERR) return(2);
if (response == 'n') return(1);
else return(0);

write_from_file(port,file_des,addr)
int  port,file_des,addr;
{
    /*
    effect          The contents of the supplied file are sent to
                   the Omnibyte at the specified address.
    signals          0: Xfer successful
                     1: Error (what kind of error is not signalled)
    */
    int err_cnt,err_val,num_read;
    int err_flag = 0;
    char buf[256];

    while((num_read = read(file_des, buf, 256)) != 0)
    {
        err_cnt = 0;
        while (err_cnt < MAX_ERR)
        {
            err_val = send_block(port_addr,num_read,buf);
            if (err_val == 0) break;        /* no error */
            if (err_val == 1) break;        /* parameters refused */
            if (err_val == 2) err_cnt++;
            /* parameter xmit error */
            if (err_val == 3) err_cnt++;
            /* data checksum error */
        }
    }
}
if (errcnt >= MAX_ERR)
{
    err_flag = 1;
    break;
}
if (err_val == 1)
{
    err_flag = 1;
    break;
}
addr += num_read; /* for next block */
if (err_flag) return(1);
else return(0);

send_addr(port, addr)
int port, addr;
/*
effect Pack the given addr into 3 bytes; send these bytes
and the 3 complement bytes to ETRM.
*/
{
    unsigned char buf[6];
    int ni;

    buf[0] = (unsigned char) (addr >> 16);
    buf[1] = (unsigned char) (addr >> 8);
    buf[2] = (unsigned char) addr;

    for (n = 0; n < 3; n++) buf[n+3] = ~(buf[n]);
    write(port, buf, 6);
}

clear(port)
int port;
{
    /*
effect flush serial line input
*/
    struct sgttyb port_settings;
    getty(port, &port_settings);
    ioctl(port, TIOCFLUSH, &port_settings);
}

send_block(port, addr, size, block)
int port, addr, size;
unsigned char *block;
/*
effect Sends block of bytes to ETRM using the "s" ETRM
command.
*/
MAX_ERR retries added by necessity 10/20/84 (rkv)
size is in the range 1-256 inclusive.

signals
0: Successful data xfer.
1: Parameter error (lands outside RAM)
2: Parameter xmit error
3: Data checksum error

* /
{
    int s_size, n, nbytes, count, flag, err_cnt;
    unsigned int our_checksum;
    char response, cmd_char;
    unsigned char s_checksum;
    clear(port);
    if (size == 256) s_size = 0;
    else s_size = size;
    flag = 0;
    err_cnt = 0;
    while((err_cnt < MAX_ERR) && (flag == 0))
    {
        cmd_char = 's';
        write(port, &cmd_char, 1);
        send_addr(port, addr);
        send_size(port, s_size);
        sleep(1);
        ioctl(port, FIONREAD, &count);
        if (count != 0)
        {
            nbytes = read(port, &response, 1);
            flag = 1;
        }
        err_cnt++;
    } /* while */
    if (err_cnt >= MAX_ERR) return(1);
    if (response == 'n') return(1);
    if (response == 'e') return(2);
    if (response != 'y') return(2);

    /* ETRM swallowed parameters. Send data block while figuring out checksum. */
    our_checksum = 0;
    for (n = 0; n < size; n++)
    {
        our_checksum += (unsigned) block[n];
        write(port, &block[n], 1);
    } /* write checksum */
    s_checksum = (unsigned char) our_checksum;
    write(port, &s_checksum, 1);

    /* get ETRM response to checksum */
    read(port, &response, 1);
if (response == 'y') return(0);
else return(3);
}

send_size(port, size)
int port, size;
{
unsigned char c;

c = (unsigned char) size;
write(port, &c, 1);

c = ~c;
write(port, &c, 1);
}

TRreset_trigger(trigger)
int trigger;
{
    /* Routine written by Howard Stearns to send a software reset to a Trigger */
    int *argp;

    Extr_port = open("/dev/extr", O);
    argp = 0;

    clear(Trig_port[trigger]);
    if (ioctl(Extr_port, trigger, argp) == -1)
    {
        printf("Error resetting trigger #%d\n", trigger);
    }
    else
    {
        printf("Reset trigger #%d\n", trigger);
    }

    close(Extr_port);
}/* TRreset_triggers */
DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

pages 1-8 of section Nov 19 01:03 1985 omni_io.c have been omitted by the author. This is the most complete copy available.
if (*ACIA1 & RORF)  /* char available at terminal input */
{
    c = *(ACIA1 + 2);  /* pick up char and queue it if queue is not full */
    if ( !full_q(&queues[0]) ) enqueue(c, &queues[0]);
}
if (*ACIA2 & RORF)  /* char available at modem input */
{
    c = *(ACIA2 + 2);
    if ( !full_q(&queues[2]) ) enqueue(c, &queues[2]);
}
if (*ACIA1 & TDRE)
{
    if ( !empty_q(&queues[1]) ) /* is there data to send? */
    {
        *(ACIA1 + 2) = dequeue(&queues[1]);  /* send data */
        /* Turn on xmit interrupt if not already on */
        if (!(word1 & TINTEN))
        {
            word1 |= TINTEN;
            *ACIA1 = word1;
        }
    }
    else if (word1 & TINTEN)  /* queue is empty. Shut off xmit interrupt if it is on */
    {
        word1 &= ~TINTEN;  /* no xmit interrupt */
        *ACIA1 = word1;
    }
}
if (*ACIA2 & TDRE)
{
    if ( !empty_q(&queues[3]) )
    {
        *(ACIA2 + 2) = dequeue(&queues[3]);
        if (!(word2 & TINTEN))
        {
            word2 |= TINTEN;
            *ACIA2 = word1;
        }
    }
    else if (word2 & TINTEN)
    {
        word2 &= ~TINTEN;
        *ACIA2 = word1;
    }
}
/* routines to manipulate character queues. RS-232 interrupts are shut off while diddling these queues for atomicity */

static init_q(q)
struct queue *q;
/*
effect  Initialize the head and tail pointers of the
requires

Unlike the other queue manipulation primitives, this one does not mask interrupts. This must be done by the caller.

*/
{
q->head = q->buf;
q->tail = q->buf;
}

static enqueue(c, q)
char c;
struct queue *q;
/* effect places the given character onto the given queue. If the queue is full, this routine waits for space to appear in the queue before enqueueing the character and returning.
*/
{
int lev;
while( full_q(q) ) [] /* wait for room in the queue */
/* queue has room. Go ahead and put the character on the queue */
lev = spl5(); /* disable acia interrupts */
*(q->head++) = c; /* put char on q and advance pointer */
if ((q->head - q->buf) >= Q_SIZE) q->head = q->buf; /* fix ptr */
intr.omni_io(); /* poke interrupt handler into life */
splx(lev); /* restore interrupt level */
}

static char dequeue(q)
struct queue *q;
/* effect Pulls a character off the queue. If the queue is empty, this routine waits for the queue to become non-empty before pulling a character off and returning.
*/
{
int lev;
char c;
while( empty_q(q) ) [] /* wait for something in q */
lev = spl5(); /* disable acia interrupt */
c = *(q->tail++); /* pull a character and advance pointer */
if ((q->tail - q->buf) >= Q_SIZE) q->tail = q->buf; /* fix ptr */
splx(lev); /* restore interrupt level */
return(c);
static int full_q(q)
    struct queue *qi;
    /*
     * effect   Determines whether queue is full.
     * returns   0: queue not full
                 1: queue full
     */
    int lev;
    int answer;

    lev = spl5();

    if ((q->tail - q->head) == 1) answer = 1;
    else if ((q->head - q->tail) == Q_SIZE-1) answer = 1;
    else answer = 0;
    splx(lev);
    return(answer);

static int empty_q(q)
    struct queue *qi;
    /*
     * effect   Determines whether queue is empty
     * returns   0: queue not empty
                 1: queue empty
     */
    int lev;
    int answer;

    lev = spl5();

    if (q->head == q->tail) answer = 1;
    else answer = 0;
    splx(lev);
    return(answer);
/*
Original Author: Steve Rosenthal
Please put your name next to any non-trivial changes/bug-fixes.
*/
#include "frame.h"

extern FRAME_PTR Fmv_left(), Fmv_right(), Fmv_up(), Fmv_down();
extern int cosmic_sensitivity, brightened_delta, global_threshold_offset;
extern long rec_offset;

/* predicates = Pcosmic_ray, Pcentered, Pgood_profile, Pbright_star_edge
These predicates evaluate a pixel in the third through fifth levels of frame sifting.
All routines return 1 if true, 0 if false.
*/
#define UINT unsigned int
#define delta(p) (int) *p

static int got_bright(); /* forward referenced function */

/* operations */
#if 0
/* calculated explicitly in sift.c to save call overhead */
int Pbrightened(pixel)
    FRAME_PTR pixel;
/*
effect
Compares specified pixel to its counterpart in the recent frame. If the values differ by more than the programmed threshold (brightened_delta), then we return 1 (true); otherwise false.
This simple scheme will work (we think as of 1/84) because the pixel values are compressed into 8-bits using a square-root scheme, thus yielding constant noise values across all 8-bit pixel values.
*/
{
    return( delta(pixel) > brightened_delta );
}
#endif
#define LOWER 0.0
#define UPPER 0.66667
#define SATURATED 255
int Pcosmicray(pixel)
    FRAME_PTR pixel;
/*
effect Calculates Erica index on delta values for the specified pixel and its row neighbors (to the immediate left and right). Result is compared to a programmed threshold.

We use delta values (the difference of the pixel over the recent value of the pixel) to eliminate fixed pixel-to-pixel variations in the CCD, and to make sure that we are looking only at contributions from brightening.

The idea is that an optical point source is recorded in several closely spaced pixels; the profile along a row is not very narrow. But a cosmic ray's profile is very narrow, and can thus be differentiated from an optical point source.

If the specified pixel is at the very left or right of the CCD (and thus is missing a neighbor) we punt and say that it is a cosmic ray (and not a real event).

UNRESOLVED What to do if the denominator in the e_index expression is zero. Currently we punt and say we've got a cosmic ray. The idea is that a 0 denominator in the e_index should never arise and if it does the frame data is probably weird and should not be taken seriously.
*/
{
    FRAME_PTR left_pix, right_pix;
    int left_delta, right_delta, center_delta;
    float e_index, threshold;

    if ( ((left_pix = Fmv_left(pixel)) == NULL_FP) &&
        ((right_pix = Fmv_right(pixel)) == NULL_FP) )
        return(1);    /* punt (by saying there is a cosmic ray)
        if we are at left or right edge */
    left_delta = delta(left_pix);
    right_delta = delta(right_pix);
    center_delta = delta(pixel);

    threshold = LOWER + ((UPPER - LOWER) * cosmic_sensitivity / 255.0);

    /* if zero denominator, punt by saying we've got a cosmic ray */
    if ((left_delta + right_delta + center_delta) == 0) return(1);

    e_index = (left_delta + right_delta) /
        (float) (left_delta + right_delta + center_delta);

    return(e_index < threshold);
}
#define EDGEMULTIPLIER 3

int Pbright_star_edge(pixel)
{ FRAME_PTR pixel;

/*
   author Roland Vanderspek 3/18/85.
   effect Checks whether a pixel passing the centering test is
   at the edge of a bright star. This is done by checking
   whether neighboring pixels are significantly brighter
   than the triggering pixel in the recent frame.
   The level of significance is a multiplicative factor
   times the threshold offset.
*/

{ FRAME_PTR left, right, up, down, upleft, upright, downleft, downright;
  register int amount, rec_center;

  left = Fmv_left(pixel);
  right = Fmv_right(pixel);
  up = Fmv_up(pixel);
  down = Fmv_down(pixel);
  upleft = Fmv_up(left);
  upright = Fmv_up(right);
  downleft = Fmv_down(left);
  downright = Fmv_down(right);

  amount = EDGEMULTIPLIER * global_threshold_offset;
  rec_center = (int) *REC(pixel);

  if (((int)*REC(left) - rec_center > amount) return(1);
  if (((int)*REC(right) - rec_center > amount) return(1);
  if (((int)*REC(up) - rec_center > amount) return(1);
  if (((int)*REC(down) - rec_center > amount) return(1);
  if (((int)*REC(upleft) - rec_center > amount) return(1);
  if (((int)*REC(upright) - rec_center > amount) return(1);
  if (((int)*REC(downleft) - rec_center > amount) return(1);
  if (((int)*REC(downright) - rec_center > amount) return(1);
  return(0);
}
int Psaturated(pixel)
FRAME_PTR pixel;
/
/*
author Roland Vanderspek 3/18/85.
effect Checks whether the neighbors of a pixel passing the centering test were saturated in the recent frame.
*/

FRAME_PTR left, right, up, down, upleft, upright, downleft, downright;
register int amount, rec_center;

left = Fmv_left(pixel);
right = Fmv_right(pixel);
up = Fmv_up(pixel);
down = Fmv_down(pixel);
upleft = Fmv_up(left);
upright = Fmv_up(right);
downleft = Fmv_down(left);
downright = Fmv_down(right);

if ((int)*REC(left) == SATURATED) return(1);
if ((int)*REC(right) == SATURATED) return(1);
if ((int)*REC(up) == SATURATED) return(1);
if ((int)*REC(down) == SATURATED) return(1);
if ((int)*REC(upleft) == SATURATED) return(1);
if ((int)*REC(upright) == SATURATED) return(1);
if ((int)*REC(downleft) == SATURATED) return(1);
if ((int)*REC(downright) == SATURATED) return(1);
return(0);
int Pcentered(pixel)
    FRAME_PTR pixel;
    /*
       effect
       Figure out if the specified pixel is at the center
       of a region of brightening.  Do this by looking at
       the delta values of the given pixel and its
       von Neumann neighborhood.
       We are centered if the given pixel brightened more
       than its right and lower neighbors, and if brightened
       more than or just as much as its left and upper
       neighbors.
       requires
       The specified pixel must be away from the edge of
       the CCD.  If one of the neighbors is at the edge
       of the CCD, we say that the specified pixel is
       NOT centered.
    */
    {
    FRAME_PTR left, right, up, down;
    register int center_del, left_del, right_del, up_del, down_del;

    if ( ((left = Fmv_left(pixel)) == NULL_FP) ||
        ((right = Fmv_right(pixel)) == NULL_FP) ||
        ((up = Fmv_up(pixel)) == NULL_FP) ||
        ((down = Fmv_down(pixel)) == NULL_FP) )
        return(0);  /* say not centered if at edge of CCD */

    center_del = delta(pixel);
    left_del = delta(left);
    right_del = delta(right);
    up_del = delta(up);
    down_del = delta(down);

    return( (center_del > right_del) &&
            (center_del > down_del) &&
            (center_del >= up_del) &&
            (center_del >= left_del) );
    }
int Pgood_profile(pixel)
    FRAME_PTR pixel;

    /*
    effect
    This predicate tries to distinguish between point-like images and "streaked" images (such as might be produced by a meteor or an airplane).

    This goal is accomplished by examining the perimeter of a box surrounding the specified pixel. If any point on the perimeter of this box has brightened (passes the brightened test) also, then we conclude that the image is not from a point source (it is streaked).

    If the specified pixel is sufficiently close to the edge of the frame so that the box cannot be constructed, then we punt and claim that the pixel is ill-formed.

    If the specified pixel is REALLY bright then we must take a couple of things into account: We may have to use a larger box, and there may be blooming along its column.
    */
#define BOXSIZE 7   /* must be odd; gives length of side of box */
{
    register int n;
    register int size;
    register FRAME_PTR fp1, fp2;

    size = BOXSIZE;

    /* find upper left and lower right corners */
    fp1 = pixel;       /* start in middle */
    fp2 = pixel;
    for (n = 0 ; n < size/2 ; n++)
    {
        fp1 = Fmv_left(Fmv_up(fp1));       /* seek upper left corner */
        fp2 = Fmv_right(Fmv_down(fp2));    /* seek lower right corner */
    }

    /* could we reach extreme corners of box? We can't if the pixel is too close to the edge of the CCD */
    if ( (fp1 == NULL_FP) || (fp2 == NULL_FP) )
    return(0);           /* no, say ill-formed */

    /* OK, look at perimeter pixels. */
    /* first, slide fp1 to the right and fp2 to the left */
    for (n = 0 ; n < size-1 ; n++)
    {
        /* if a perimeter point brightened, pixel is not a point source */
        if (got_bright(fp1) || got_bright(fp2)) return(0); /* exit */
        fp1 = Fmv_right(fp1);
        fp2 = Fmv_left(fp2);
    }

    /* now slide fp2 up and fp1 down */
    for (n = 0 ; n < size-1 ; n++)
    {
        /* if a perimeter point brightened, not a point source */
    }
if (got_bright(fp1) || got_bright(fp2)) return(0);
fp1 = Fmv_down(fp1);
fp2 = Fmv_up(fp2);
}

/* Perimeter quiet, so the pixel is point-like */
return(1);
}

/* internal procedures */

/* replaced by macro at top */
#if 0
static int delta(pixel)
    FRAME_PTR pixel;
    {
        return((int) (*pixel - *REC(pixel)));
    }
#endif

static int got_bright(pixel)
    FRAME_PTR pixel;
    /*
    effect Determines if the pixel brightened. Same as
    Pbrightened (no longer defined), which is
    what the sifter uses to determine if a pixel
    brightened.
    */
    {
        return( delta(pixel) > brightness_delta );
    }
/*
Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.
*/

/* perform query operations; for descriptions of individual
   queries see "query.doc" */

#include "config.h"
#include "frame.h"
#include "coord.h"
#include "omni_io.h"

#define BYTE unsigned char
#define WORD unsigned short

extern Hinit(), Hsnd_config();
extern int Hsnd_done(), Hsnd_busy();
extern Finit(), Fmake_sub_frame();
extern FRAME_PTR Fget_base();
extern Cmake_entry();
extern int Cget_x(), Cget_y();
extern Sinit();
extern int Sreadout();
extern THcalc_threshold();
extern BYTE *THget_col_vec();
#ifndef REFCOLVEC
extern char *THget_row_vec();
#endif

extern int r_chan, t_chan;
extern BYTE *sub_area;
extern long frame_area:

extern int current_frame, recent_frame;
extern long rec_offset;
#ifndef OLD_FRAME
extern int old_frame;
extern long old_offset;
#endif

extern int cur_discarded, rec_discarded;
#ifndef OLD_FRAME
extern int old_discarded;
#endif

extern unsigned int sift_time;
extern int f_width, f_height, s_width, s_height;
extern COORD flashes[], standards[];

extern int cosmic_sensitivity, brightened_delta;
extern int test_mode;

/* declare private functions before use */
static BYTE get_char();
static WORD get_short();

int do_queries()
/*
effect Perform query operations as requested by the Overseer.
returns 0: Queries terminated by the end_queries operation.
-1: Queries terminated by the go_to_ETRM operation.
*/
{
char cmd_char, answer;
int n, x, y, w, h;
COORD *which_table;
int num_left;
BYTE *vec;

while(1)
{
    read(TERMIN, &cmd_char, 1);

    switch( (unsigned int) cmd_char)
    {
    case 1: /* query */
        answer = 0;
        write(TERMOUT, &answer, 1); /* respond with null */
        break;
    case 2: /* version */
        answer = VERSION;
        write(TERMOUT, &answer, 1); /* respond with version number */
        break;
    case 3: /* go_to_ETRM */
        return(-1); /* indicate we should quit */
        break;
    case 4: /* end_queries */
        /* swap current_frame and recent_frame */
        n = current_frame;
        current_frame = recent_frame;
        recent_frame = n;
        /* swap discarded flags for current and recent */
        n = cur_discarded;
        cur_discarded = rec_discarded;
        rec_discarded = n;
        /* calculate new rec_offset and maybe old_offset */
        rec_offset = Fget_base(recent_frame) - Fget_base(current_frame);
        ifdef OLD_FRAME
        old_offset = Fget_base(old_frame) - Fget_base(current_frame);
        endif
        return(0); /* terminate query mode, prepare for sift */
        break;
    case 5: /* set_sift_time */
        sift_time = (unsigned int) get_short(); /* pick up arg */
        break;
    case 6: /* set_cosmic_sensitivity */
        cosmic_sensitivity = (int) get_char();
        break;
    }
case 7:    /* calc_threshold_from_current */
    /* calculate threshold vector(s) or frame using offset */
    THcalc_threshold(current_frame, (int) get_char());
    break;
    
    case 8:    /* set_brightened_delta */
    brightened_delta = (int) get_char();
    break;
    
    case 9:    /* test_mode */
    test_mode = 1;
    break;
    
    case 10:    /* operational */
    test_mode = 0;
    break;
    
    case 11:    /* sift_against_old */
    
    ifdef OLD_FRAME

    /* swap old frame and current frame */
    n = current_frame;
    current_frame = old_frame;
    old_frame = n;
    /* swap discarded flags */
    n = cur_discarded;
    cur_discarded = old_discarded;
    old_discarded = n;
    
    old_offset = Fget_base(old_frame) - Fget_base(current_frame);
    rec_offset = Fget_base(recent_frame) - Fget_base(current_frame);
    /* Last line added 11/7/85 by RKV */
    
    #endif
    break;
    
    case 12:    /* discard_current_frame */
    cur_discarded = 1;
    break;
    
    case 13:    /* discard_old_frame */
    ifdef OLD_FRAME
    old_discarded = 1;
    #endif
    break;
    
    case 14:    /* send_entire_image */
    n = (unsigned int) get_char(); /* frame_number */
    if (n > NFRAME-1) n = 0;
    n = current_frame; /* current frame fix by RKV */
    /* 10/19/85 -- let's hope it work */
    Hsnd_config((BYTE *) Fget_base(n), (unsigned long) FRAME_SIZE);
    break;
    
    case 15:    /* send_column_vector */
    /* send column vector out. The catch is that we must place it in Multibus memory so the HSSL can find it; we use the subarray area for this. But the subarray area may not be big enough for the vector, so break up the vector into pieces if needed. */
    num_left = COL; /* bytes to send */
    vec = THget_col_vec();
    while(num_left)
    {
        if (num_left <= MAX_SUB_SIZE)
        {

```
for (n = 0 ; n < num_left ; n++) sub_area[n] = *vec++;
Hsnd_config( sub_area, (unsigned long) num_left );
num_left = 0;
}
else
{
for (n = 0 ; n < MAX_SUB_SIZE ; n++) sub_area[n] = *vec++;
Hsnd_config( sub_area, (unsigned long) MAX_SUB_SIZE );
while(!Hsnd_done()) {}
num_left -= MAX_SUB_SIZE;
}
break;
case 16: /* send_row_vector */
 ifndef REF_COL_VEC
 num_left = ROW;
 vec = (BYTE *) THget_row_vec();
 while(num_left)
 {
 if ( num_left <= MAX_SUB_SIZE )
 {
 for (n = 0 ; n < num_left ; n++) sub_area[n] = *vec++;
 Hsnd_config( sub_area, (unsigned long) num_left );
 num_left = 0;
 }
 else
 {
 for (n = 0 ; n < MAX_SUB_SIZE ; n++) sub_area[n] = *vec++;
 Hsnd_config( sub_area, (unsigned long) MAX_SUB_SIZE );
 while(!Hsnd_done()) {} 
 num_left -= MAX_SUB_SIZE;
 }
 }
 #endif
 break;
case 17: /* send_subarray */
 x = (unsigned int) get_short(); /* where_x */
 y = (unsigned int) get_short(); /* where_y */
 w = (unsigned int) get_char(); /* width */
 h = (unsigned int) get_char(); /* height */
 if ( (w * h) <= MAX_SUB_SIZE )
 {
 Fmake_sub_frame(current_frame, x, y, w, h, sub_area);
 Hsnd_config(sub_area, (unsigned long) w * h);
 while(!Hsnd_done()) {}
 Fmake_sub_frame(recent_frame, x, y, w, h, sub_area);
 Hsnd_config(sub_area, (unsigned long) w * h);
 #ifdef OLD_FRAME
 while(!Hsnd_done()) {}
 Fmake_sub_frame(old_frame, x, y, w, h, sub_area);
 Hsnd_config(sub_area, (unsigned long) w * h);
 #endif
 break;
case 18: /* set_flash_subarray_size */
 w = (unsigned int) get_char();
h = (unsigned int) get_char();
if ( (w * h) <= MAX_SUB_SIZE )
{
    f_width = w;
    f_height = h;
}
break;
case 19:
    /* set_standard_subarray_size */
    w = (unsigned int) get_char();
    h = (unsigned int) get_char();
    if ( (w * h) <= MAX_SUB_SIZE )
    {
        s_width = w;
        s_height = h;
    }
    break;

    n = (unsigned int) get_char();
    x = (unsigned int) get_short();
    y = (unsigned int) get_short();
    if ( n <= MAX_STANDARD-1 ) Cmake_entry(standards, n, x, y);
    break;

    n = Sreadout(sub_area);
    /* readout operation returns size of report for our convenience only; Overseer must know size and format of the report */
    Hsnd_conf(sub_area, (unsigned long) n);
    break;

case 24:
    /* set_bus_memory_offset */
    n = (unsigned int) get_short();
    frame_area = (long) (n * 1024) + MB_MEM;
    Finit( (FRAME_PTR) frame_area);
    sub_area = (BYTE *) frame_area + NFRAME*ROW*COL;
    break;

case 25:
    /* receive_channel */
    n = (int) get_char();
    if ( (n == 0) || (n == 2) )
    {
        r_chan = n;
        Hinit(r_chan, t_chan);
    }
    break;

case 26:
    /* xmit_channel */
    n = (int) get_char();
    if ( (n == 1) || (n == 3) )
    {
        t_chan = n;
        Hinit(r_chan, t_chan);
    }
    break;

    case 27:
    /* HSSL_busy */
answer = (BYTE) Hsnd_busy();
write(TERMOUT, &answer, 1);
break;
default: /* check for flash or standard subarray request */
if ( !(cmd_char & 0x80) ) break; /* no high bit */
/* figure out if this is a flash subarray request or a standard subarray request */
/* pick up flash or standard number and coords */
n = (int) cmd_char & 0x3f; /* retain lower 6 bits */
if (cmd_char & 0x40) /* standard?? */
{
/* catch bad standard_num */
if ( n > MAX_STANDARD-1 ) break;
w = s_width;
h = s_height;
which_table = standards;
}
else /* catch bad flash_num */
{
/* catch bad flash_num */
if ( n > MAX_CANDIDATE-1 ) break;
w = f_width;
h = f_height;
which_table = flashes;
}
x = Cget_x(which_table, n);
y = Cget_y(which_table, n);
/* send subarrays from current, recent and maybe old */
Fmake_sub_frame(current_frame, x, y, w, h, sub_area);
Hsnd_config(sub_area, (unsigned long) w * h);
while( !Hsnd_done() ) {} /* wait for it to xmit fully */
Fmake_sub_frame(recent_frame, x, y, w, h, sub_area);
Hsnd_config(sub_area, (unsigned long) w * h);
#endif /* OLD_FRAME */
while( !Hsnd_done() ) {}  
Fmake_sub_frame(old_frame, x, y, w, h, sub_area);
Hsnd_config(sub_area, (unsigned long) w * h);
#endif /* OLD_FRAME */
break;

] /* switch */
} /* while(1) */
} /* do_queries() */

static BYTE set_char()
/*
effect Reads TERMIN for a single byte.*/
BYTE c;
read(TERMIN, (char *) &c, 1);
return(c);
]

static WORD get_short()
/*
effect Reads TERMIN for two bytes and combines them into
an unsigned short. First byte read should be
the high order byte of the short; the second
should be the low order byte.
*/
[
WORD answer;

answer = (WORD) get_char() * 256;
answer += get_char();
return(answer);
]
Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.

#include "config.h"
#include "frame.h"
#include "coord.h"
#include "omni_io.h"
#include "stats.h"

#define BOTTOM 15 /* added 12/6/84 by RKV */
#define TOP 293 /* added 10/7/85 by RKV */
#define BYTE unsigned char
#define UINT unsigned int
#define delta(p) (int) *p - (int) *REC(p)
    /* UINT changed to int to allow for negatives by RKV 12/3/84 */

#endif

#endif

#endif

extern int Pcosmic_ray(), Pcentered(), Pgood_profile();
extern int Pbright_star_edge(); /* added 3/18/85 by RKV */
extern int Psaturated(); /* added 11/8/85 */

#endif

extern Sstartingsift(); Smadereport(), Saborting_sift(),
    Sending_sift();
    /* Sreached_level() is now a macro defined in ”stats.h” */
extern Cmake_entry();
extern int Hrec_done();
extern unsigned long Hrec_where();

extern int THget_sky();
extern BYTE *THget_threshold();
#else
    extern BYTE *THget_row_vec();
#endif

extern FRAME_PTR Fget_base(), Fmv_left(), Fmv_right(), Fmv_up(), Fmv_down();
extern int Frow(), Fcol();

extern COORD flashes[];
extern int current_frame, recent_frame, rec_discarded, cur_discarded;
extern int timer_expired;
extern int brightened_delta;
extern int rec_offset;
extern struct stats sp;
extern int test_mode;

int waiting_for_handshake;

sift()
    /* effect Performs six-level sift of frame pair to find
        flash candidates while rejecting noise events. */
requires

Current and recent CCD frames contain valid data. Frame processing proceeds from the beginning of the frame in a sequential fashion. This means that the entire current frame need not be in place when the sift is started -- the incoming data must, however, always be at least a few rows ahead of the sift routine.

All reports (candidates or sift) will look for a handshake byte from the Overseer corresponding to the last report sent in this sift (therefore, the first report will not look for the byte).

side-effects

Signals Overseer (over the RS-232 link) upon each candidate identification (candidate report) and upon completion of frame processing (sift termination report).

terminates

When finished sifting entire image or when sifting time is exhausted.

*/
{
  register BYTE *pixel;
  register BYTE *thresh_ptr;
  #ifdef REF_2_VECS
    char *row_vector;
    register char row_offset;
  #endif
  register short inner, outer; /* loop counters */
  int flash_num;
  int coded_x, coded_y;
  int too_many; /* flag indicating we should quit sifting because limit on number of candidate reports has been reached */

  too_many = 0;
  flash_num = 0;

  inner = COL - 1;
  outer = ROW - 1;

  /* get ptr to current frame */
  pixel = (BYTE *) Fget_base(current_frame);

  /* get ptr to threshold vector or frame */
  thresh_ptr = THget_threshold();
  #ifdef REF_2_VECS
    /* get ptr to row offset vector */
    row_vector = THget_row_vec();
    row_offset = *row_vector++;
  #endif

  if (!(cur_discarded || rec_discarded) && !test_mode)
    {
      /* do not sift -- we do not have two valid frames. Just wait for the sift timer to expire */
      while(!timer_expired) {}  // no_sift(); /* tell overseer */
      return;
    }
Starting_sift(); /* tell stat package */

waiting_for_handshake = 0;

do {
    do {
        /* first level: See if pixel is greater than a pre-set value */
        ifndef REF_2_VECS
            if (*pixel++ > row_offset + *thresh_ptr++)
        else
            if ((*pixel++ > *thresh_ptr++) && (Frow((FRAME_PTR)pixel) > BOTTOM)
                && (Frow((FRAME_PTR)pixel) < TOP))
        endif
        {
            /* second level: Compare pixel to recent frame */
            Sreached_level(2); /* tell stat package */
            if (delta(pixel) > brightened_delta)
            {
                /* third level: see if we are centered on the brightened area */
                Sreached_level(3);
                if (!Psaturated((FRAME_PTR) pixel))
                {
                    /* no more criteria to satisfy; signal overseer */
                    locate(pixel, &coded_x, &coded_y); /* figure out location of flash to
                        better than single pixel res */
                    report_candidate(pixel, flash_num, coded_x, coded_y);
                    Cmake_entry(flashes, flash_num,
                        (int)((float)coded_x/64.0 + 0.5),
                        (int)((float)coded_y/64.0 + 0.5));
                }
                /* correction to Cmake_entry command made 12/3/84 by RKV */
                Smade_report(); /* tell stat package */
                /* see if we reached per-sift limit on reported candidates. If we did, stop sifting */
                if (++flash_num >= MAX_CANDIDATE)
                {
                    too_many = 1; /* indicate condition */
                    break;
                }
            }
        }
    }while(--inner != -1);
    inner = COL - 1;
    ifndef REF_FRAME
        thresh_ptr = THget_threshold();
    endif
} /* inner do */

}
#endif
#endif

if (!test_mode)
{
    /* see if HSSL data is sufficiently ahead of us before
       resuming sifting on next row. But do not wait if we are
       done receiving data or we have run out of time. */
    while((Hrec_where() / COL) < (ROW - 1 - outer)) &&
        !timer_expired && !Hrec_done() ) {} /* wait */

    if (timer_expired || too_many) break;
} /* outer do */
while(--outer != -1);

if (timer_expired)
{
    sift_aborted((ROW - 1) - outer); /* tell Overseer */
    Saborting_sift((ROW - 1) - outer); /* tell stats pack */
}
else /* we finished or issued maximum # of candidate reports */
{
    while(!timer_expired) {} /* wait for sift interval */
    sift_complete(); /* tell Overseer */
    Sending_sift(); /* tell stats pack */
}

/* internal procedures */

#define SIFT_TERMINATION_REPORT 0x40
#define SIFT_ABORTED 0x01
#define REC_ERROR 0x02
#define NO_SIFT 0x04

static sift_complete()
/*
   effect         Tell Overseer that sifting is complete.
   writes          RS-232
*/
{
    if (waiting_for_handshake) get_handshake_byte();
    write_char(SIFT_TERMINATION_REPORT);
    /* DMA should be finished. If it is not, then an error occurred */
    if (!Hrec_done() && !test_mode) write_char(REC_ERROR);
    else write_char(0);

    write_char(0);
    write_char(0);
}

static no_sift()
/*
   effect         Tell Overseer that no sift was done.
   writes          RS-232
*/
/*
if (waiting_for_handshake) get_handshake_byte();
write_char(SIFT_TERMINATION_REPORT);
if (!Hrec_done() && !test_mode) write_char(REC_ERROR | NO_SIFT);
else write_char(NO_SIFT);
write_char(0);
write_char(0);
}

static sift_aborted(row_num)
int row_num;
/*
effect Tell Overseer that sifting has been aborted
writes at the given row number
writes RS-232
*/
{
if (waiting_for_handshake) get_handshake_byte();
write_char(SIFT_TERMINATION_REPORT);
if (!Hrec_done() && !test_mode) write_char(REC_ERROR | SIFT_ABORTED);
else write_char(SIFT_ABORTED);
write_char((BYTE) (row_num / 256)); /* high byte */
write_char((BYTE) row_num); /* low byte */
}

static write_char(c)
BYTE c;
{
write(TERMOUT, (char *) &c, 1);
}

static locate(flash, coded_x, coded_y)
BYTE *flash;
int *coded_x, *coded_y;
/*
effect Locate the given flash to better than single pixel
resolution.
Algorithm used is a parabola fit along the X and
Y axes. For the X axis case, the specified pixel
is examined with its left and right neighbors.
Let L, M, and R be the amount by which the left,
middle and right pixels brighter over the recent
frame. Then the displacement in X from the X
value of the middle pixel is (R-L) / (L+R-2M). 
Similarly for Y.
We then multiply x and y by 64, take the integer
part, and place the result in coded_x and coded_y.
In this manner fractional information is coded in
an integer.
marginal The three pixels along X or Y exhibit an amount
of brightening leading to a zero denominator in
the calculation (they exhibit no curvature).
In this case, we return the X or Y of the
specified pixel.
For best results, the specified pixel should be the one that brightened the most. That is, we should be centered up on the event.

/*
FRAME_PTR middle, left, right, up, down;
int L, M, R, U, D;

middle = (FRAME_PTR) flash;
if ( (left = Fmv_left(middle)) == NULL_FP ) ;
   (right = Fmv_right(middle)) == NULL_FP ) ;
   (up = Fmv_up(middle)) == NULL_FP ) ;
   (down = Fmv_down(middle)) == NULL_FP ) )
   return;        /* specified pixel at edge of CCD */

L = delta(left);
M = delta(middle);
R = delta(right);
U = delta(up);
D = delta(down);

if ((L + R - 2*M) == 0)
   *coded_x = 64 * Fcol(middle);
else
   *coded_x = (64 * Fcol(middle)) + ( (32 * (L - R)) / (L + R - 2*M) ) ;
   /* correction to dotyfit algorithm made 11/13/84 by roland */

if ((U + D - 2*M) == 0)
   *coded_y = 64 * Frow(middle);
else
   *coded_y = (64 * Frow(middle)) + ( (32 * (U - D)) / (D + U - 2*M) ) ;
   /* correction to dotyfit algorithm made 11/13/84 by roland */

#define CAND_REPORT 0x80

static report_candidate(pixel, flash_num, coded_x, coded_y)
BYTE *pixel;
int flash_num, coded_x, coded_y;
/*
effect Tells Overseer all about this candidate.
requires flash_num between 0 and 63, inclusive.
writes RS-232
*/
{
   if (waiting_for_handshake) get_handshake_byte();
   write_char(CAND_REPORT);
   write_char((BYTE) flash_num);
   write_char((BYTE) (coded_x / 256)); /* high order */
   write_char((BYTE) coded_x);        /* low order */
   write_char((BYTE) (coded_y / 256));
   write_char((BYTE) coded_y);
write_char(*pixel);       /* current value */
write_char(*REC(pixel));  /* recent */
write_char((BYTE) THget_sky(pixel)); /* expected sky value */
waiting_for_handshake = 1;
}

get_handshake_byte()
{
  char byte;
  read(TERMIN,&byte,1);
  waiting_for_handshake = 0;
}
/*
Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.
*/

#include "config.h"
#include "stats.h"

#define BYTE unsigned char

/*
stats = Sinit, Sstarting_sift, Sending_sift, Saborting_sift,
Smade_report, Sreadout, (replaced Sreached_level)

Statistics package for sifting process. We accumulate data on the following:
1. How many sifts are started, ended, and aborted.
2. Total of row numbers at time of abort for each aborted sift.
3. Total number of times each sifting level (past first level) is reached. Done with a macro defined in stats.h
4. Total number of candidate reports made to Overseer.

This package provides for the initialization of the statistics data, the collection of the data, and the reporting of the data.
*/

/* the rep */

extern struct stats sp;

/* rep invariants:
1. sp.started_sifts >= 0
2. sp-ended_sifts >= 0
3. sp.aborted_sifts >= 0
4. sp.t_reported_candidates >= 0

****
NOTE: Although space is allocated in t_level_reached for levels 0 and 1,
**** this package only keeps track of levels reached beginning with level 2 (the first sifting level is level 1 and is reached by every pixel). And there is no level 0. So these extra spaces in the array are unused.

****
NOTE: No provision is made for guarding against overflow in the
**** t_level_reached vector, or in t_abort_row. If the stats package is used to much before readout and an re-initialization, these values may overflow.
*/

/* operations */
Sinit()
    /*
    effect Initializes the state of the statistics package.
    All recorded data (if any) is lost.
    This operation MUST be performed before any other
    operations of this package are invoked.
    */
    [int n;
     sp.started_sifts = 0;
     sp.ended_sifts = 0;
     sp.aborted_sifts = 0;
     sp.t_abort_row = 0;
     for (n = 2; n <= MAX_LEV; n++) sp.t_level_reached[n] = 0;
     sp.t_reported_candidates = 0;
    ]

Sstarting_sift()
    /*
    effect Prepares to receive statistics for a new sift.
    requires A sift must not already be in progress.
    */
    [sp.started_sifts++;
    ]

Sending_sift()
    /*
    effect Records that this sift has been ended.
    requires A sift must have been started (with the Sstarting_sift
call).
    */
    [sp.ended_sifts++;
    ]

Saborting_sift(row_num)
    int row_num;
    /*
    effect Records that this sift was aborted at the given row
    number, updating the average abort row value.
    requires A sift must have been started.
    */
    [sp.t_abort_row += row_num;     /* update abort row total */
     sp.aborted_sifts++;
     /* record aborted sift */
    ]

#if 0
    /* replaced by macro in stats.h */
Sreached_level(level_num)
int level_num;
/*
effect record that sifting has reached the specified level.
requires level_num must be between 2 and MAX_LEV (inclusive).
*/
{
sp.t_level_reached[level_num]++;
}
#endif

Smade_report()
/*
effect records that a candidate report was made to the overseer.
*/
{
sp.t_reported_candidates++;
}

int Sreadout(dest)

register BYTE *dest;
/*
effect Contents of accumulated statistics are read into the supplied destination buffer. Handy to prepare statistics report to Overseer.
returns Size of statistics report.
requires dest should be big enough to (sizeof sp) bytes.
*/
{
register int n;
register char *stats;
stats = (char *) &sp;

for (n = 0; n < (sizeof sp); n++) *dest++ = *stats++;
return(sizeof sp);
}
Please put your name next to any non-trivial changes/bug-fixes.

#include "config.h"
#include "frame.h"

#define BYTE unsigned char

eextern FRAME_PTR Fget_base();
eextern int Frow(), Fcol();

eextern int global_threshold_offset;

#include "confie.h"

int reference-frame;

if defined REF_FRAME

eextern int reference_frame;
eextern long ref_offset;
#endif

threshold = THcalc_threshold, THget_threshold, THget_row_vec,

THget_col_vec, THget_sky

Module to calculate and disseminate sky information. The sky plus a fixed offset is used during the first sifting level to determine if the pixel under consideration is bright enough to be examined further.

There are three modes of use, depending on which of REF_COL_VEC,
REF_2_VECS, or REF_FRAME are defined in "config.h". Note only one may be defined.

If REF_COL_VEC is used, then we calculate a vector of COL elements, each of which is the median value of a column of the CCD plus a fixed offset. In this case, THget_row_vec is undefined.

If REF_2_VECS is used, then we calculate a second vector in addition to the vector of COL elements described above. This vector has ROW elements, each of which is the median of a row of the CCD (each element has had the median of its column subtracted).

By using median values, we capture sky brightness information (and not star brightnesses, assuming a sparse population of stars in the field).

If REF_FRAME is defined, the two vectors discussed above are used to create a reference frame, which contains spatial sky information plus an offset. Each point in the reference frame is the sum of the corresponding entries in each vector.

*/
/* the rep */

static int offset_from_sky;
static BYTE thresh_col_vec[COL];
#endif /* two vectors or reference frame */
static char row_vec[ROW];
#endif

/* operations */

THcalc_threshold(frame_num, offset)
int frame_num, offset;
/
"effect Use the specified frame to calculate reference 
vector(s) or frame, depending on the system 
configuration.
requires The specified frame_num must be between 0 and 
NFRAME-1.
*/
{
    offset_from_sky = offset;
    global_threshold_offset = offset;
    calc_col_vec(Fget_base(frame_num));
#endif /* that is, two vectors or ref frame */
    calc_row_vec(Fget_base(frame_num));
#endif
    #ifdef REF_FRAME
    calc_ref_frame(Fget_base(reference_frame));
    #endif
}

BYTE *THget_threshold()
/*
"effect Returns a pointer to the column threshold vector 
or to the reference frame (if REF_FRAME is set).
*/
{
    #ifdef REF_FRAME
    return( (BYTE *) Fget_base(reference_frame) );
    #else
    return(thresh_col_vec);
    #endif
}
#endif /* two vectors or reference frame */

char *THget_row_vec()
/*
"effect Returns a pointer to the row vector
*/
{
    return(row_vec);
}
#endif

BYTE *THget_col_vec()
>Returns a pointer at the column vector
 */
{
    return(thresh_col_vec);
}

int THget_sky(pixel)
FRAME_PTR pixel;
/*/     effect Gets the sky value to be expected at the given pixel. */
{
#ifdef REF_COL_VEC
    return( (int) thresh_col_vec[Fcol(pixel)] - offset_from_sky);
#endif
#ifdef REF_2_VECs
    return( (int) thresh_col_vec[Fcol(pixel)] - offset_from_sky +
            (int) row_vec[Frow(pixel)] );
#endif
#ifdef REF_FRAME
    /* return( (int) *REF(pixel) - offset_from_sky ); */
    return( (int) offset_from_sky );        /* introduced 11/11/84 by rkv */
#endif
}

/* Internal operations */

static calc_col_vec(frame)
register FRAME_PTR frame;
/*/     effect Finds the median of each column of the given frame, adds offset_from_sky, and places the result into thresh_col_vec. */
{  
    register short x, y;
    register FRAME_PTR from;
    register short *to;
    short column[ROW];

    for (x = 0; x < COL; x++)
    {
        to = column;
        from = &frame[x];
        y = ROW;
        while( --y != -1 )
        {
            /* Collect pixels to take median of */
            *to++ = *from;
            from += COL;
        }
        thresh_col_vec[x] = (BYTE) nths( column, ROW, (ROW + 1) / 2 ) + offset_from_sky;
    }
}
#ifndef REF_COL_VEC    /* that is, two vectors or ref frame */
static calc_row_vec(frame)
    register FRAME_PTR frame;
/*
   effect Finds the median of each row of the given frame, using column-median corrected pixel values.
   requires thresh_col_vec and offset_from_sky must have been calculated.
*/
[
register short x, y;
register FRAME_PTR from;
register short *to;
register BYTE *thptr;
short row[COL];
for (y = 0 ; y < ROW ; y++)
[
    to = row;
    from = &frame[ y * COL ];
    thptr = thresh_col_vec;
    x = COL;
    while( --x != -1 )
    {
        *to++ = (short) *from++ - (short) *thptr++;
    }
    row_vec[y] = (char) (nths( row, COL, ( COL + 1 )/ 2 ) +
        (short ) offset_from_sky);
    /* changed to + sign by Roland 3/15/85 -------------- */
}
#endif

#ifdef REF_FRAME
static calc_ref_frame(frame)
    register FRAME_PTR frame;    /* destination */
/*
   effect Using thresh_col_vec and row_vec, build up a reference frame by adding elements of the two vectors.
   requires thresh_col_vec and row_vec must have been calculated.
*/
[
register short x, y;
register BYTE *col;
register char *row;
register char row_val;
row = row_vec;
for (y = 0 ; y < ROW ; y++)
[
    row_val = *row++;
    col = thresh_col_vec;
    x = COL;
    while( --x != -1 )
# enditf

/* Find the nth from the minimum value in an array */
/* Monte Carlo method intended for finding medians */
/* 2/13/85 jpd */

/* For random data, this routine takes about */
/* 2.6*numdata + O( log( numdata ) ) comparisons */
/* If the data is tightly clustered about the mean, */
/* there is a speedup; it may take as few as */
/* 0.5*numdata comparisons. */
/* There is a slight penalty if the array is completely */
/* or partially sorted; it is at most 25%. */

/* NTH will be nthi, nths, etc., depending on DATATYPE */
#define NTH nths
#define DATATYPE short

NTH( data, numdata, n )
DATATYPE data[]; /* Data array (will be scrambled on return)
int numdata; /* length of data array */
int n; /* index if item to find:
1 <= n <= numdata */
{
    register DATATYPE boundary, thisdata;
    register DATATYPE *lowp, *highp;
    DATATYPE v1, v2;
    int nlowbin;

    lowp = data; /* Init data pointer */

    v1 = data[ ixrand( numdata )];
    {
        register DATATYPE v1r = v1;
        int nc = 1 + numdata - n; /* "Complement" of n */

        if( nc > n )
            highp = lowp + nc;
        else
            highp = lowp + n; /* Limit to test for done */

        /* Scan for the first point which doesn't match the boundary point */
        /* If we encounter enough matching points, */
        /* the boundary is the answer */
        while( *lowp++ == v1r ) { /* Back up to get point */
            if( lowp >= highp ) return( v1r );
        }

        v2 = v1; /* Back up to set point */
    }
boundary = ( v1 >> 1 ) + ( v2 >> 1 ); /* Beware overflows */

highp = data + numdata; /* Now process the whole thing */
thisdata = *lowp; /* Prime the pump */

if( v2 < v1 ) { /* Bin 2 is low bin */
  for( ; lowp < highp; thisdata = *lowp ) {
    if( thisdata <= boundary ) { /* Bin 2 */
      *lowp = **--highp; /* Exchange */
      *highp = thisdata;
      } else ++lowp; /* Data point in right place */
  }
}

nlowbin = numdata - ( lowp - data );
if( nlowbin >= n ) return( NTH( highp, nlowbin, n ) );
else return( NTH( data, lowp - data, n - nlowbin ) );

else { /* Primary bin is low bin */
  for( ; lowp < highp; thisdata = *lowp ) {
    if( thisdata > boundary ) { /* Bin 2 */
      *lowp = **--highp; /* Exchange */
      *highp = thisdata;
      } else ++lowp; /* Don’t move point */
  }
}

nlowbin = ( lowp - data );
if( nlowbin >= n ) return( NTH( data, nlowbin, n ) );
else return( NTH( highp, numdata - nlowbin, n - nlowbin ) );
}
Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.

#include "ptm6840.h"

/*
timer = Tinit, Tset, Tabort, timer_expired

Timer chip abstraction. Counts in intervals of 10 milliseconds. When the programmed count expires, the global timer_expired flag is set. Note that a global variable is used to communicate the expiration of the timing interval (as opposed to a procedure) to bypass the overhead of a procedure call.

*/

int timer_expired, h_t_int();

#define L4AUTOVECTOR Dx70
#define COUNT ((E_FREQ / 100) - 1) /* for 10 MS */

Tinit()
/*
effect Sets up interrupt handler, and configures 6840 chip to interrupt every 10 ms. We use counter 1 (of 3); the rest are inactive.
*/
{
    int lev;
    int (**vector)();

    lev = spl4(); /* disable timer interrupts */

    vector = (int (**)) L4AUTOVECTOR;
    *vector = h_t_int; /* install interrupt handler */

    *CR2 = REG3;
    *CR3 = 0; /* disable interrupts from counter 3 */
    *CR2 = REG1; /* disable interrupts from counter 2 and point to control register 1 */
    *MSB1 = (char) (COUNT / 256); /* high order of COUNT */
    *LCH1 = (char) (COUNT % 256); /* low order of COUNT */

    *CR1 = INT_CLK | BIT_16 | CONT | INT_EN;

    splx(lev);
}
Tset(cs)
unsigned int cs;

 effect
Sets up a delay of cs "centiseconds" (0.01 sec).
The global timer_expired is reset and will be
set upon the completion of the time interval.
Note that the time interval may be shorter than
the given cs by as much as one 10 millisecond
interval.

 requires
 cs must be non-zero

 if (cs == 0) return;
 int lev;

 lev = spl4();
time_count = cs;
timer_expired = 0;
splx(lev);
}

Tabort()

 effect
Shut timer chip down to disable interrupts.

 if (*STAT != (INT_PENDING : CT1_STAT)) return;  /* not me */
dummy = *CTR1;  /* reset interrupt source */
/* note: "lint" complains that I am setting but not using dummy.
 This is OK. Note further that I must assign the value of *CTR1
 into a variable or the compiler may "optimize" the *CTR1 away.
*/
if (time_count == 0) return;
else if (--time_count == 0) timer_expired = 1;
Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.

/*
 top level Trigger program. This is it, folks! */

/* NOTE: When loading the entire trigger package, this should be the first module loaded. */

#include "config.h"
#include "frame.h"
#include "omni_io.h"
#include <sgtty.h>

#define BYTE unsigned char

extern Tinit(), Tabort(), Tset();
extern Hinit(), Habort(), Hrec_config();
extern int Hrec_done();
extern unsigned long Hrec_where();
extern Finit(), Sinit();
extern FRAME_PTR Fget_base();
extern sift(), init omni_io();
extern int do_queries();

/* system globals accessed here */
extern int r_cha, t_cha;
extern unsigned int sift_time;
extern long frame_area;
extern BYTE *sub_area;
extern int current_frame, recent_frame;
extern long rec_offset;
#endif OLD_FRAME
extern int old_frame, old_discarded;
extern long old_offset;
#endif REF_FRAME
extern int reference_frame;
extern long ref_offset;
#endif
extern int was_current, was_recent;
extern int cur_discarded, rec_discarded;
extern int test_mode;

Trigger()
/* effect
entered
exited */
{ Initializes all modules in the system, initializes certain global variables and runs the trigger from the top-most perspective.
 Via ETRM after being down-loaded into the Omnibyte.
 Via the go_to_ETRM() call during queries. }
struct sgttyb ttyi;

spl7();  /* inhibit all interrupts (except
 * NMI which we don't use) */

r_chan = 0;
* receive on DMA channel 0 by default */
t_chan = 1;
* xmit on DMA channel 1 by default */
Hinit(r_chan, t_chan);
* set up interrupt handler and vector
 * for HSSL, and program channel
 * numbers */

Tinit();  /* initialize 10 ms Timer and interrupt
 * handler, vector */
init_omni_io();  /* initialize RS-232; install interrupt
 * handler for omni_io */
Habort();  /* abort any HSSL activity */
spl2();  /* allow interrupts now */

/* set up tty mode; Raw, no echo */
sgtty(TERMIN, &tty);
tty.sg_flags |= RAW;
tty.sg_flags &= ~ECHO;
sgtty(TERMIN, &tty);

frame_area = MB_MEM;  /* use base of Multibus by default */
Finit((FRAME_PTR)frame_area);
Sinit();  /* clear statistics */

test_mode = 0;  /* not test mode */

/* set up initial frame numbers */
current_frame = 0;
recent_frame = 1;
rec_offset = Fget_base(recent_frame) - Fget_base(current_frame);
#endif OLD_FRAME
old_frame = 2;
old_offset = Fget_base(old_frame) - Fget_base(current_frame);
#endif REF_FRAME
reference_frame = 3;
ref_offset = Fget_base(reference_frame) - Fget_base(current_frame);
#else
reference_frame = 2;
ref_offset = Fget_base(reference_frame) - Fget_base(current_frame);
#endif

was_current = current_frame;  /* initialize with */
was_recent = recent_frame;  /* sensible values */

/* all images initially have no valid data */
cur_discarded = 1;
rec_discarded = 1;
#endif OLD_FRAME
old_discarded = 1;
#endif
/* use area above frame storage as area for subarray preparation */
sub_area = (BYTE *) frame_area + NFRAME*ROW*COL;

while(1)
{
    if ( do_queries() == -1 ) break;  /* return value of -1 means
    we should return to ETRM */

    /* set up to receive DMA image into current frame */
    Hrec_config( (BYTE *) Fget_base(current_frame),
                (unsigned long) FRAME_SIZE);
    /* do it twice to gobble up spurious data (if any) */
    Hrec_config( (BYTE *) Fget_base(current_frame),
                (unsigned long) FRAME_SIZE);

    if (!test_mode)
    {
        /* wait for image data to start flowing into trigger */
        while( Hrec_where() == 0 ) {}  /* wait */
    }
    /* set sift timer */
    Tset(sift_time);

    if (!test_mode)
    {
        /* wait for HEADSTART rows to accumulate */
        while( (Hrec_where() / COL) < HEADSTART ) {}  /* wait */
    }
    /* show current contains valid data */
    cur_discarded = 0;

    sift();  /* sifts, and issues sift termination report */
    /* remember which frames were used for this past sift */
    was_current = current_frame;
    was_recent = recent_frame;
    /* discard current frame if there was an error receiving the frame. An incomplete transfer is the only error we can detect. If no error was detected, we claim that the current frame now contains valid data. */
    if ( Hrec_done() )  
        cur_discarded = 0;  
    else  
        cur_discarded = 1;
    }  /* while(1) */

    /* return to ETRM. First shut down HSSL and Timer to thwart their interrupts */
    Habort();
    Tabort();
}
/*
Original Author: George Mitsuoka
Adapted by Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.
*/

/* DMAC chip description file */

#define BYTE unsigned char
#define WORD unsigned short
#define LONG unsigned long

/* note: The Omnibyte OB68K1A does not swap bytes on byte accesses. However, bytes are swapped within words and longwords. So when using word or longword access to read or write WORD or LONG locations within the 68450, you must swap bytes */

/* registers within a 68450 channel. The 68450 contains four such channels. An appropriate declaration would be:
"struct channel dma_chip[4]"
*/

struct channel
{
    BYTE csr; /* channel status register */
    BYTE cer; /* channel error register */
    WORD nr10; /* null register */
    BYTE dcr; /* device control register */
    BYTEocr; /* operation control register */
    BYTE scri; /* sequence control register */
    BYTE ccri; /* channel control register */
    WORD nr1000; /* null register */
    WORD mtc; /* memory transfer counter */
    LONG mar; /* memory address register */
    LONG nr10000; /* null register */
    LONG dar; /* device address register */
    WORD nr11000; /* null register */
    WORD btc; /* base transfer counter */
    LONG bar; /* base address register */
    LONG nr100000; /* null register */
    BYTE nr100100; /* null register */
    BYTE niv; /* normal interrupt vector */
    BYTE nr100110; /* null register */
    BYTE eiv; /* error interrupt vector */
    BYTE nr101000; /* null register */
    BYTE mfc; /* memory function codes */
    WORD nr101010; /* null register */
    BYTE nr101100; /* null register */
    BYTE cpr; /* channel priority register */
    WORD nr101110; /* null register */
    BYTE nr110000; /* null register */
    BYTE dfc; /* device function codes */
    LONG nr110010; /* null register */
    WORD nr110110; /* null register */
}
BYTE nr111000;  /* null register */
BYTE bfc;      /* base function codes */
LONG nr111010; /* null register */
BYTE nr11110;  /* null register */
BYTE scr;      /* general control register on */
                   /* channel 3, null on 0, 1, 2 */

};

/* values of bit fields within registers */

/* channel status register */
#define COC_CSR 0x80  /* channel operation complete */
#define BTC_CSR 0x40  /* block transfer complete */
#define NDT_CSR 0x20  /* normal device termination */
#define ERR_CSR 0x10  /* error */
#define ACT_CSR 0x08  /* channel active */
#define PCT_CSR 0x02  /* pcl/ transition */
#define PCS_CSR 0x01  /* state of pcl/ input line */
#define RST_CSR OxFF  /* reset status */

/* channel error register */
#define NOERR_ERR 0x00  /* no error */
#define CONF_ERR 0x01  /* configuration error */
#define OP_ERR 0x02  /* operation timing error */
#define ADOM_ERR 0x05  /* address error: memory address or counter */
#define ADDD_ERR 0x06  /* address error: device address */
#define ADBB_ERR 0x07  /* address error: base address or counter */
#define BUSM_ERR 0x09  /* bus error: memory address or counter */
#define BUSD_ERR 0x0A  /* bus error: device address */
#define BUSB_ERR 0x0B  /* bus error: base address or counter */
#define CNTM_ERR 0x0D  /* count error: memory address or counter */
#define CNTD_ERR 0x0E  /* count error: device address */
#define CNTB_ERR 0x0F  /* count error: base address or counter */
#define XAB_ERR 0x10  /* external abort */
#define SAB_ERR 0x11  /* software abort */

/* device control register */
#define BTM_DCR 0x00  /* burst transfer mode */
#define CSMH_DCR 0x80  /* cycle steal mode without hold */
#define CMH_DCR 0x00  /* cycle steal mode with hold */
#define EXPOO_DCR 0x00  /* 68000 compatible device, explicitly addressed */
#define EXPOO_DCR 0x10  /* 6800 compatible device, explicitly addressed */
#define IMPA_DCR 0x20  /* device with ACK/,, implicitly addressed */
#define IMPAR_DCR 0x30  /* device with ACK/,, READY/,, implicitly addressed */
#define BIT0_DCR 0x00  /* 8 bit port */
#define BIT16_DCR 0x00  /* 16 bit port */
#define STAT_DCR 0x00  /* PCL as status input */
#define STATI_DCR 0x01  /* PCL as status input with interrupt */
#define STRT_DCR 0x02  /* PCL as start pulse */
#define ABRT_DCR 0x03  /* PCL as abort input */

/* operation control register */
#define M2O_OCR 0x00 /* transfer from memory to device */
#define D2M_OCR 0x80 /* transfer from device to memory */
#define BYTE_OCR 0x00 /* byte operations */
#define WORD_OCR 0x10 /* word operations */
#define LONG_OCR 0x20 /* long word operations */
#define NCH_OCR 0x00 /* chain operation disabled */
#define ACH_OCR 0x08 /* array chaining */
#define LCH_OCR 0x0C /* linked chaining */
#define ARQ_OCR 0x00 /* auto-request limited by GCR */
#define MARQ_OCR 0x01 /* auto-request at maximum rate */
#define XRO_OCR 0x02 /* REQ/line initiates a transfer */
#define AXRQ_OCR 0x03 /* auto request first operand, external request on rest of operands */

/* sequence control register */
#define MNC_SCR 0x00 /* memory address does not count */
#define MINC_SCR 0x04 /* memory address counts up */
#define MDEC_SCR 0x06 /* memory address counts down */
#define DNC_SCR 0x00 /* device address does not count */
#define DINC_SCR 0x01 /* device address counts up */
#define DDEC_SCR 0x02 /* device address counts down */

/* channel control register */
#define STR_CCR 0x80 /* start operation */
#define CNT_CCR 0x40 /* continue operation */
#define HLT_CCR 0x20 /* halt operation */
#define SAB_CCR 0x10 /* software abort */
#define INT_CCR 0x08 /* interrupt enable */

/* channel priority register */
#define PRI0_CPR 0x00 /* highest priority */
#define PRI1_CPR 0x01
#define PRI2_CPR 0x02
#define PRI3_CPR 0x03 /* lowest priority */

/* function codes (68000 definitions) */
#define USR_DAT 0x01 /* user data space */
#define USR_PRG 0x02 /* user program space */
#define SUP_DAT 0x05 /* supervisor data space */
#define SUP_PRG 0x06 /* supervisor program space */
#define IACK_FC 0x07 /* interrupt acknowledge cycle: dma chip should not assert this */
#define ACIA1 (char *) 0x3FF01  /* "terminal" serial port */
#define ACIA2 (char *) 0x3FF21  /* "host" serial port */
#define RESET_ACIA 3
#define DIV16 1  /* 16x clock */
#define BIT8S1 0x14  /* 8 bits, no parity, 1 stop bit */
#define TINTEN 0x20  /* enable xmit interrupt */
#define RINTEN 0x80  /* enable receive interrupt */
#define CONFIG (DIV16 : BIT8S1)
#define RDRF 1  /* receive data register full */
#define TDRE 2  /* transmit data register empty */
Nov 19 01:02 1985 config.h Page 1

/*****************************************
Original Author: Steve Rosenthal
Please put your name next to any non-trivial changes/bug-fixes.
*/

#define VERSION 2
#define COL 400
#define ROW 292
#define FRAME_SIZE (ROW*COL)

#define SAT_PIX -255 /**< value of negatively saturated pixel, 6/19/84 by */
#define NFRAME 4 /**< current, recent, old, reference */
#define OLD_FRAME /**< define to support an old frame */

/** different flavors of reference for first level of sifting. 
   Exactly one of these should be #define'd. */
#define REF_COL_VEC /**< reference done from a vector of medians 
   of columns (plus an offset). */
#define REF_2_VECS /**< reference done with two vectors that 
   capture variations along both axes. 
   Better model, but somewhat slower than 
   REF_COL_VEC */
#define REFFRAME /**< same model as REF_2_VECS, except information 
   is contained in an entire image, so it 
   is faster than REF_2_VECS (indeed, even 
   a tiny bit faster than REF_COL_VEC). */

#define MB_MEM 0x40000 /**< where Multibus memory starts (default) */
#define MB_M_SIZE 0x80000 /**< how much Multibus memory we can use */

/** leftover Multibus memory devoted to subarray construction buffer */
#define MAX_SUB_SIZE (MB_M_SIZE - NFRAME*ROW*COL)

#define MAX_CANDIDATE 64 /**< maximum number of candidates that 
   may be reported in a single sift. Absolute 
   maximum is 64 */
#define MAX_STANDARD 64 /**< maximum number of standard stars that 
   may be defined per Trigger. Absolute maximum 
   is 64 */

#define HEADSTART (ROW / 25) /**< number of rows to accumulate before 
   beginning sift */
/*
   Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.
*/

typedef struct
{
   short x;
   short y;
} COORD;
/*
Original Author: Steve Rosenthal
Please put your name next to any non-trivial changes/bug-fixes.
*/

#include "confs.h"

typedef unsigned char *FRAME_PTR; /* used to point at or within a frame */

#define NULL_FP ((FRAME_PTR) 0)

/* macros to take us from a current pixel to a recent or old or reference pixel */
#ifdef REC(fp) (fp + rec_offset)
  #ifdef OLD_FRAME
    #define OLD(fp) (fp + old_offset)
  #endif
  #ifdef REF_FRAME
    #define REF(fp) (fp + ref_offset)
  #endif
#else
  #define REC(fp) (fp)
  #define OLD(fp) (fp)
  #define REF(fp) (fp)
#endif
Original Author: Steve Rosenthal

Please put your name next to any non-trivial changes/bug-fixes.

/*
* this file replaces <stdio.h> for the Omnibyte environment *
*
* #define TRIGGER
* define if used in Trigger software. Used
to restrict omni_io to a very restricted
subset of stdio. Don't define it here;
define it in the compiler line
(say cc -DTRIGGER ...)
*/

#define BUFSIZ 512
#define _NFILE 4

ifndef TRIGGER

extern struct _iobuf {
    char  *_ptr;
    int    _cnt;
    char   *_base;
    char   _flag;
    char   _file;
} _iob[ _NFILE];

define _IOREAD 01
#define _IOWRT 02
#define _IONBF 04
#define _IOMYBUF 010
#define _IOEOF 020
#define _IOERR 040

#define NULL 0
#define FILE struct _iobuf
#define EOF (-1)
#endif

#define STDIN 0
#define TERMIN STDIN
#define STDOUT 1
#define TERMOUT 1
#define MODIN 2
#define MODOUT 3

ifndef TRIGGER

#define stdin _iob[STDIN] /* input from Omnibyte terminal port */
#define termin stdin /* termin is alias for stdin */

#define stdout _iob[STDOUT] /* output to Omnibyte terminal port */
#define termout stdout /* termout is alias for stdout */

#define modin _iob[MODIN] /* input from Omnibyte modem port */
#define modout _iob[MODOUT] /* output to Omnibyte modem port */
#define getc(p) (--(p)->_cnt>=0? *(p)->_ptr++&0377:_filbuf(p))
#define getchar() getc(stdin)
#define putc(x,p) (--(p)->_cnt>=0? ((int)(*(p)->_ptr++=(unsigned)(x))):_filbuf((p)))
#define putchar(x) putc(x,stdout)
#define feof(p) (((p)->_flag&_IOEOF)!=0)
#define ferror(p) (((p)->_flag&_IOERR)!=0)
#define fileno(p) (p)->_file

FILE *fopen();
FILE *freopen();
FILE *fdopen();
long ftell();
char *fgets();

#endif
/*
Original Author: Steve Rosenthal
Please put your name next to any non-trivial changes/bug-fixes.
*/

/* describes 6840 timer chip as used on the Omnibyte OB68K1A */

#define E_FREQ 1000000 /* frequency of E input to 6840 */
#define PTM 0x3ff61 /* base address */

#define CR3 ((char *) PTM) /* w */
#define CR1 CR3 /* w */
#define CR2 ((char *) PTM + 2) /* w */
#define STAT ((char *) PTM + 2) /* r */

#define CTR1 ((char *) PTM + 4) /* r */
#define LSB1 ((char *) PTM + 6) /* r */
#define CTR2 ((char *) PTM + 8) /* r */
#define LSB2 ((char *) PTM + 10) /* r */
#define CTR3 ((char *) PTM + 12) /* r */
#define LSB3 ((char *) PTM + 14) /* r */

#define MSB1 ((char *) PTM + 4) /* w */
#define LCH1 ((char *) PTM + 6) /* w */
#define MSB2 ((char *) PTM + 8) /* w */
#define LCH2 ((char *) PTM + 10) /* w */
#define MSB3 ((char *) PTM + 12) /* w */
#define LCH3 ((char *) PTM + 14) /* w */

/* values for control registers (partial list) */
#define INIT 0x1
#define REG1 0x1
#define REG3 0x0
#define INT_CLK 0x2 /* E input used as clock */
#define BIT_16 0x0
#define CONT 0x0
#define INT_EN 0x40

/* status masks */
#define INT_PENDING 0x80
#define CT1_STAT 0x1
#define CT2_STAT 0x2
#define CT3_STAT 0x4
/* Original Author: Steve Rosenthal */

Please put your name next to any non-trivial changes/bug-fixes. */

#define MAX_LEV 6 /* number of sifting levels */

/* macro to record a reached level (instead of function call to stats.c -- saves time */
#define Sreached_level(n) (sp.t_level_reached[n]++)

struct stats
{
    long started_sifts;
    long ended_sifts;
    long aborted_sifts;
    long t_abort_row; /* total of row numbers at each abort */
    long t_level_reached[MAX_LEV+1]; /* total number of times each level is reached */
    long t_reported_candidates; /* total number of candidates reported */
};
APPENDIX C

The Instrument Control Electronics Software

Introduction

This Appendix contains the operating system, instrument control code and Sequel code used by the ICE in the ETC test unit. The operating system and instrument control code are together in the first file in this Appendix, while the Sequel code stands alone in the second file.
forget kx
kx :ascii Kernel Extensions

*/
(* tools to compile from host *)
(* compile time utilities *)
-emit : i>compile \A+ ;
-emit8 : i>compile \A!8+ ;

(* Array stuff *)
bytes : 2 - idictop \A + dup idictop \A \A swap ! idictop ! ;
words : 1 asl bytes ;

(* Block structure stack *)
;i>blks :variable ;
32 words ;
-push : i>nest \A dup 64 - ifpz -errnest dup 2 + ;i>nest ! ;i>blks + ! i
-pop : i>nest \A dup 2 - ifn -errnest 2 - dup ;i>nest ! ;i>blks + \A ;

(* ops for block structure *)
-3skip :code
13 *( inc r3 )
13 *( inc r3 )
;

-brf :code
43 *( lda r3 )
52 *( str r2 )
83 *( slo r3 )
F4 *( add )
A3 *( pla r3 )
93 *( shi r3 )
7C00 *( adci 0 )
B3 *( phi r3 )
;
\begin{verbatim}
find -3skip -emit find -brf -emit ;>compile @ -push 0 -push -\textbackslash{}zero ;
\end{verbatim}

\begin{verbatim}
} : -pop if -errnest -pop dup ;>compile @ swap -1 - swap !8 ;
\end{verbatim}

\begin{verbatim}
find [ _immediate ;
find ] _immediate ;
find ][ _immediate ;
\end{verbatim}

-\texttt{init :code}

\begin{verbatim}
60 * ( irx
60 * ( irx
F0 * ( ldx
57 * ( str r7
27 * ( dec r7
22 * ( dec r2
72 * ( ldxax
57 * ( str r7
27 * ( dec r7
F800 * ( ldi 0
57 * ( str r7
27 * ( dec r7
57 * ( str r7
27 * ( dec r7
;       -- init counter
\end{verbatim}

-\texttt{do\texttt{init :code}}

\begin{verbatim}
F800 * ( ldi 0
57 * ( str r7
27 * ( dec r7
57 * ( str r7
27 * ( dec r7
57 * ( str r7
27 * ( dec r7
;       -- push four bytes of zero on return stack
\end{verbatim}

-\texttt{test :code}

\begin{verbatim}
13 * ( inc r3
87 * ( glo r7
A9 * ( plo r9
97 * ( shi r7
B9 * ( phi r9
E9 * ( sex r9
60 * ( irx
72 * ( ldxax
60 * ( irx
F3 * ( xor
CA+17 * ( if zero
29 * ( dec r9
72 * ( ldxax
60 * ( irx
F3 * ( xor
CA+10 * ( if zero
\end{verbatim}
Nov 20 05:07 1985 Opsys Page 3

E2 *(sex r2) -- do arith on usual stack
17 *(inc r7) -- pop counters
17 *(inc r7)
17 *(inc r7)
23 *(dec r3) -- point to offset
43 *(lda r3)
52 *(str r2) -- convenient place for off
83 *(glo r3) -- point r3 at loop end
F4 *(add)
A3 *(plo r3)
93 *(shi r3)
7C00 *(adci 0)
B3 *(phi r3)

E2 *(sex r2)

--[notest :code

13 *(inc r3) -- always fall thru
;
--] :code

87 *(slo r7)
A9 *(plo r9)
97 *(shi r7)
B9 *(phi r9) -- copy rsp
E9 *(sex r9)
60 *(irx)
F801 *(ldi 1)
F4 *(add)
59 *(str r9) -- inc count
60 *(irx)
F800 *(ldi 0)
74 *(adc)
59 *(str r9)
E2 *(sex r2)
43 *(lda r3) -- get offset of loop top
52 *(str r2) -- save it
83 *(glo r3)
F7 *(sm) -- find loop top
A3 *(plo r3)
93 *(shi r3)
7F00 *(smbi 0) -- borrow
B3 *(phi r3)
;
--it :code

87 *(slo r7)
A9 *(plo r9)
97 *(shi r7)
B9 *(phi r9) -- copy rsp
19 *( inc r9
19 *( inc r9
09 *( ldn r9
73 *( stxd
29 *( dec r9
09 *( ldn r9
73 *( stxd
;
- :code
87 *( glo r7
A9 *( plo r9
97 *( gh r7
B9 *( phi r9
19 *( inc r9
19 *( inc r9
09 *( ldn r9 -- transfer current count
73 *( stxd
29 *( dec r9
49 *( lda r9
52 *( str r2
19 *( inc r9 -- point to limit
F800 *( ldi 0
F6 *( shr -- set borrow
49 *( lda r9 -- get count limit lsbs
77 *( smb -- subtract count
52 *( str r2
60 *( irx
09 *( ldn r9 -- msbs
77 *( smb
73 *( stxd
22 *( dec r2
;
-return :code
43 *( lda r3 -- get loop nest count
A9 *( plo r9
*( do
C2+OA *( until zero
17 *( inc r7
17 *( inc r7
17 *( inc r7
17 *( inc r7 -- pop loop counts off rs
29 *( dec r9
89 *( glo r9
C0-0A *( repeat
17 *( inc r7
47 *( lda r7 -- return
A3 *( plo r3
07 *( ldn r7
B3 *( phi r3
;
-break :code
B7  *( glo r7
A9  *( plo r9
97  *( shi r7
B9  *( phi r9  -- copy rsp
E9  *( sex r9
60  *( irx r7
F801 *( idi 1
F4  *( add
59  *( str r9  -- inc count
60  *( irx r7
F800 *( idi 0
74  *( adc
S9  *( str r9
E2  *( sex r2
43  *( ida r3  -- get offset of loop top
52  *( str r2  -- save it
83  *( glo r3
F7  *( sm
A3  *( plo r3
93  *( shi r3
7F00 *( smbi 0  -- borrow
B3  *( phi r3
17  *( inc r7  -- pop counters
17  *( inc r7
17  *( inc r7
43  *( ida r3
52  *( str r2  -- convenient place for offset
83  *( glo r3
F4  *( add
A3  *( plo r3
93  *( shi r3
7C00 *( adci 0
B3  *( phi r3

blastreg :code

F87F *( idi 7F
B7  *( phi r7
F8FE *( idi 7E
B2  *( phi r2

[ : ;>loops A1+ find -[init -emit find -[test -emit ];>compile A -push 1 -pl -\zero ;
do[ : ;>loops A1+ find -do[init -emit find -[notest -emit ];>compile A -push 1 -push -\zero ;
] : ;>loops A1- find -] -emit -pop ifz -errnest -pop dup ];>compile A swap -dup 3 + -emit8 swap !8 ;
return : find -return -emit ;>loops A -emit8 ;
-looptop : -pop -pop swap ifz [ drop repeat ] ;
br ek : i;nest A -looptop swap ;>nest !
    find -break -emit ];>compile A swap -1 + -emit8 ;
Nov 20 05:07 1985 Opsys Page 6

```plaintext
find [ immediate ;
find do [ immediate ;
find ] immediate ;
find return immediate ;
find break immediate ;
-?loop : i>loops @ ifz -errnest ;
i+ : -?loop find -i+ -emit ;
i- : -?loop find -i- -emit ;
find i+ immediate ;
find i- immediate ;

-err" : repent 201 error ;

-" : code

93 *( shi r3 -- push literal addr
73 *( stxd
83 *( glo r3
73 *( stxd
*( do
43 *( lda r3 -- look for terminating null
CA-02 *( repeat nonzero

" : find -" -emit ;text @i+ do[ ;text @i+ dup ifz -err" dup \ 22 -
    ifz [ break ] -emit8 ] drop 0 -emit8 ;
find " immediate ;
# : blastree qmon ;

*(----------------------------------------------------------------------
*( Bit manipulation Routines
*----------------------------------------------------------------------

not : code 60 60 F0 FBFF 73 F0 FBFF 73 ;
*( .BCLR : .NOT .AND )
bclr : code 60 72 60 FBFF F2 73 72 60 FBFF F2 73 22 ;
*( Mask maker )
bits : 0 swap [ 1 asl 1 + ] ;
*( return bit <n> )
bit : 1 swap asl ;
*( Word subscripting )
[] : dup + + ;

*(----------------------------------------------------------------------
*( Low Level Arithmetic Routines
*----------------------------------------------------------------------

mul : code 82 BF 60 72 AB 72 BB 72 AA F0 BA F800 73 73 73 52 AF 9B B9 BB A9
     BA F6 AA CB+0F 89 F4 52 60 99 74 52 60 8F 74 52 22 22 BA C2+0D
     89 FE A9 99 7E B9 8F 7E AF C0-21 9A C2+09 AA F800 BA 60 C0-32 9F A2 ;
div : code F800 AB BB 1B AA BA 60 72 A9 72 B9 99 F6 B9 89 76 A9 9A 76 BA AA
     BB F6 BA CB+11 FS 52 60 9A 75 52 60 89 75 52 60 99 75 C0+0E
     F4 52 60 9A 74 52 60 89 74 52 60 73 88 8E AB 9B 7E C3+07
     BB 22 22 C0-3C 60 73 BB 73 FS C3+09 22 BA F4 52 60 9A 74 52 22 22
lt? : code 60 72 60 F5 22 72 60 75 C3+06 FBFF C0+03 F800 73 73 ;
```

(* Square root routine *)

i-sqh :variable ;
i-sql :variable ;
i-sqlast :variable ;
-sqloop : dup i-sqlast ! dup i-sqh @ swap i-sql @ swap div drop +/2 dup
i-sqlast @ lt? if repeat i-sql ! i-sqh ! 65535 -sqloop drop i-sqlast @ ;

(* Composite Arithmetic *)

i-sign :variable ;
i-atemp :variable ;
abs : dup ifn csign ;

(* Double precision change sign *)
chsd : dup ifz [ swap csign ][ csign swap -1 xor ] swap ;

(* Make sign and magnitude of number *)
sgn : dup ifn [ csign -1 ][ 0 ] ;

(* Extend sign single -> double precision *)
snx : dup ifn [ -1 ][ 0 ] swap ;

(* Signed multiply with double precision result *)
smul : sgn i-sign ! swap sgn i-sign @ xor i-sign ! mul i-sign @ if [ chsd ]

(* Signed multiply with single precision result *)
* : smul swap drop ;

(* Signed divide - double precision dividend, single divisor, quotient, rem *)
sdiv : sgn i-sign ! i-atemp ! swap dup ifn [ -1 i-sign @ xor i-sign ! -1 xor swap csign dup ifz [ swap 1 + swap ] ][ swap ]
i-atemp @ div i-sign @ if [ csign swap csign swap ] ;

(* all single signed divide, no remainder *)
/ : i-atemp ! snx i-atemp @ sdiv drop ;

(* sdiv with no remainder *)
sdnr : sdiv drop ;

(* Faster version of positive divide with remainder for num conversion *)
/plr : i-atemp ! 0 swap i-atemp @ div swap ;

(* Formatting routines *)

": \ 2E %emit ;
": \ 20 %emit ;
-s1z : swap dup if [ return ] drop 1 - dup if repeat swap ;
-prest : swap \ 30 + %emit 1 - dup if repeat drop ;
-btbcdd : 4 [ 10 /plr ] ;
-psign : "sp dup ifn [ "- csign ] ;
= : -psien -btbcd 4 -slz swap 1 + -prest ;
=.2 : -psien -btbcd 2 -slz swap 1 + -prest " .2 -prest ;
=onoff : if [ " on" ][ " off" ] ,t ;

(*( *******************************************
*( PROGRAMMABLE SEQUENCER STUFF
*( *******************************************

quit ;
forget stacks
hostcom
stacks : " Stacks" ,t "nl ;
&init : 0 swap ! ;
&push : dup dup @ 2 + swap ! dup @ + ! ;
&pop : dup dup @ + @ swap dup @ 2 - swap ! ;
&depth : @ 1 asr ;
&top : dup @ + @ ;
&drop : dup @ 2 - swap ! ;
stacks ;
quit ;

forget seqc
hostcom
seqc : " Sequencer commands" ,t "nl ;
iseqbase : 57344 ; *( Loc of sequencer memory in the COSMAC address space ;
iseqcs : :iseqbase 4095 + ;
seqgo : :iseqcs @0 \ CO or :iseqcs !8 ;
seqstop : :iseqcs @0 \ 3F and :iseqcs !8 ;
pss : :iseqcs @0 dup 7 and " Gain =" ,t = "nl not \ CO and dup " Sequencer i
  ifz [ drop " running" ][ \ 80 - ifz [ " halted" ][ " reset" ] ] ,
    "nl ;
seqc ;

8xasm : " The 8X300 assembler" ,t "nl ;
*( Instruction counter ;
.I :variable ;
*( Stuff for loading the 8X300 from the COSMAC ;

*( Assembler support ;
emit : .I @ 2 asl :iseqbase + ! .I @ 1 + .I ! ;
3args : \ 1f and swap \ 07 and 5 asl + swap \ 1f and 8 asl + ;
2args : \ ff and swap \ 1f and 8 asl + ;
op : 13 asl + emit ;
errpage : " Branch boundary violation." ,t "nl quit ;

*( The 8X300 instructions ;

MOVE : 3args 0 op ;
ADD : 3args 1 op ;
AND : 3args 2 op ;
XOR : 3args 3 op ;
XMITIO : 3args 6 op ;
XMITR : 2args 6 op ;
NZTR : dup \ FF00 and .1 A \ FF00 and - if \ errpage \ Zargs 5 op ;
JMP : \ 1FFF and 7 op ;
8xasm ;

ss : " Sequencer support" ,t "nl ;
*( Handy macros for sequencer ;

*( Make a status register field specification given the (Signetics) bit ;
*( number of its LSB ;
STAT : 16 + ;
*( Auxiliary register specification ;
AUX : 0 ;
*( Control instructions ;
HALT : 0 STAT 1 1 XMITIO ;
RESET : 1 STAT 1 1 XMITIO ;
NOP : 0 0 0 MOVE ;
ss ;

sequel : " The sequence compiler" ,t "nl ;

*( The action location counter ;
.A :variable ;
*( The current action ;
.action :variable ;
*( Stacks for loop control ;
.forstack :variable ;
10 words ;
.idostack :variable ;
20 words ;
*( temp buffers for deferred action statements ;
alist :variable ;
50 words ;
idlist :variable
50 words ;
alp :variable ;
idlp :variable
.idlp :variable
.idlfp :variable
.dlist :variable
50 words ;
.*( Error mesages ;
errcount : " Loop count <1 or >256." ,t "nl quit ;
errnest : " Loops too deep or not nested properly." ,t "nl quit ;
errfield : " Illegal action." ,t "nl quit ;

*( Action compiler front end ;
*( If control instructions have been emitted ahead of the action counter ;
*( the specified action will be emitted immediately. Otherwise, the action ;
*( and delay are placed in a buffer for later processing when there is ;
*( something to merge with. This code also merges duplicate actions into ;
*( single actions with summed delays (making "2 DELAY" equivalent ;
*( to "1 DELAY 1 DELAY") ;
emita : .A @ 2 asl 2 + iseqbase + ! .A @ 1 + .A ! ;
backfill : do[ dup ifz { break } .A @ 1 @ - ifpz { break }
 .action @ emita 1 - ] ;
mergelist : idip 3 2 - @ + idip 3 2 - ! ;
emitlist : iaction @ ialp @! + idip @! + ;
fillist : ialp @ ialist - if [
  @ ialp 3 2 - iaction @ - ifz { mergelist ][ emitlist ][ emitlist ][ emitlist ]
  DELAY : backfill dup if { dup idsum @ + idsum ! fillist }[ drop ] ;
EXECUTE : 1 DELAY ;
microsec : 2 asl ;

*( Loop basics are defined here so that the compiler can use them: *
  *( for building delays. They are also used later by the regular (user) loop *
  *( code. ;

*( Check and reformat loop count (256 -> 0) (also negate) ;
getcount : dup 1 - \ FF00 if errcount csign \ FF and ;
*( Figure out which 8300 register to use for a loop counter (1-6 or 9) ;
get8xreg : iforstack & depth dup 1 - ifn errnest dup 8 - ifpz 8 - ifpz ;
getnext : iforstack & top add test ;
*( Delays ;
delayloop : dup 515 -
  ifpz { 1 - 0 swap 257 div swap qfor 127 qfor qnext qnext }
  dup 3 - ifn [ [ NOP ] ][ dup 1 - 1 asr qfor qnext 1 and ifz [ NOP ] ] ;

*( } sync emptys the deferred action list and aligns the location ;
*( counters so that the instruction counter is exactly } locations ;
*( behind the action counter. Most action code is actually emitted ;
*( by sync. It is invoked before generating code for each action statement.

delwarn = " Warning --" , t = " DELAY inserted. " , t " nl ;
delneed : dup . 1 @ 0 @ - ;
xtrad : delneed idsum @ swap lt? if [ delneed idsum @ - dup delwarn DELAY ]
*( nmid determines the number of extra states to occur *
*( before the new control statement ;
nmid : dup idsum @ swap - ;
emitdelay : dup delayloop idsum @ swap - idsum !
  do [ . A @ . A - ifz [ delay ] ialfp @ @ emit ] ;
  delwhole : idfp @ @ + emitdelay ialfp @ @ drop ;
delsplit : nmid dup idfp @ @ swap - idfp @ @ ! emitdelay ;
midfill : do [ nmid ifz [ delay ]
  nmid idfp @ @ lt? if [ delsplit ][ delwhole ] ] ;
nonull : do [ idfp @ @ if [ delay ] idfp @ @ drop ialfp @ @ drop ] ;
*( merge emits the action code to be merged with the new control statement *
merge : [ nonull ialfp @ @ emit ialfp @ @ dup @ 1 - swap ! ] ;
initlist : idlist idfp @ @ ialist ialfp @ 0 idsum ! ;
sync : idlist idfp @ @ ialist ialfp @ xtrad midfill merge initlist ;

*( backstate fills extra actions for control statements with out *
*( of line stuff in them), given the distance back to look for state ;
backstate : . A @ swap - 2 asl 2 + iseqbase + @ emit ;

*( Start compiling sequencer code. Initialize compiler variables and ;
BEGIN: 0 .1 ! 0 .A ! 0 iaction ! iforstack &init ido stack &init initlist
AUX 1 XMITR ;

End up a sequencer program;
END: 2 sync RESET .1 @ JMP ;

Loops;
FOR: 2 sync .1 @ 2 + iforstack &push
csign dup 8 asr get8xreg swap XMITR
iforstack &push
\ FF and get8xreg swap XMITR ;

End of loop;
NEXT: 5 sync .1 @ 6 + addtest iforstack &pop if errnest
     .1 @ 3 + addtest .1 @ 4 + JMP iforstack &pop JMP NOP .1 @ 2 - JMP
     1 backstate 4 backstate 4 backstate ;

Indefinite loops;
DO: 0 sync .1 @ 3 + ido stack &push 0 ido stack &push ;
   backpatch, given an address, puts a jump to the current loc + 1 at it ;
   backpatch: .1 @ swap .1 ! dup 1 + JMP .1 ! ;
   REPEAT: 1 sync do[ ido stack &pop dup ifz { drop break } backpatch ]
            ido stack &pop JMP ;

WHILE, UNTIL, and SET invert their data 'cause the 8X300 does.

Control register bit fields;
HALT/ : 0 STAT 1 ;
RESET/ : 1 STAT 1 ;
SYNC : 2 STAT 1 ;
IDLE : 4 STAT 1 ;
CNTL : 5 STAT 1 ;
GAIN : 7 STAT 2 ;

Action compiler;

Check for legal action;
fieldcheck: - if [ errfield ] ;

Action setup words;
on and off are for one bit controls;
high and low work with anything (equivalent to on, off if field is 1 bit
mid is for tri-level drivers only;
ON : 1 fieldcheck bit iaction @ or iaction ! ;
OFF : 1 fieldcheck bit -1 xor iaction @ and iaction ! ;
HIGH : bits swap asl -1 xor iaction @ and iaction ! ;
LOW : bits swap asl iaction @ or iaction ! ;
MID : 2 fieldcheck dup 2 bits swap asl -1 xor iaction @ and swap 1 + bit
     or iaction ! ;
seqnames : " Sequencer bit assignments" , t " nl ;
*( NAME  LSB  #BITS ;
 DP :  0  2 ;
 OM :  2  2 ;
 OT :  4  2 ;
 OS :  6  2 ;
 ENCOD :  8  1 ;
 HOLD0 :  9  1 ;
 HOLD1 : 10  1 ;
 HOLD2 : 11  1 ;
 SP1 : 12  1 ;
 SP2 : 13  1 ;
 SP3 : 14  1 ;
 SP4 : 15  1 ;
seqnames ;

*( **********************************************
*( COSMAC tools for ccd system control )
*( **********************************************

*(dac tools)

!dac1 : \ 73d0 + !8 ; *(set value of dac)
!dac2 : \ 73dc + !8 ; *( ""
!dac3 : \ 73e8 + !8 ; *( ""
!dac4 : \ 73f4 + !8 ; *( ""

@dac1 : \ 73d0 + @8 ; *(return value of dac)
@dac2 : \ 73dc + @8 ; *( ""
@dac3 : \ 73e8 + @8 ; *( ""
@dac4 : \ 73f4 + @8 ; *( ""

clks1 : 47 98 130 49 96 130 67 75 137 77 10 [ i- !dac1 ] ;
clks2 : 47 98 130 49 96 130 67 75 137 90 10 [ i- !dac2 ] ;
clks3 : 70 75 100 70 75 100 50 95 100 50 10 [ i- !dac3 ] ;
clks4 : 47 98 130 49 96 130 0 120 100 87 10 [ i- !dac4 ] ;
pdac1 : 12 [ i+ dup = "sp @dac1 = "nl ] ; *(print dac adu values ) ;
pdac2 : 12 [ i+ dup = "sp @dac2 = "nl ] ; *( ""
pdac3 : 12 [ i+ dup = "sp @dac3 = "nl ] ; *( ""
pdac4 : 12 [ i+ dup = "sp @dac4 = "nl ] ; *( ""

daczero : 48 [ 0 i+ !dac1 ] ;
dczr1 : 12 [ 0 i+ !dac1 ] ;
dczr2 : 12 [ 0 i+ !dac2 ] ;
dczr3 : 12 [ 0 i+ !dac3 ] ;
dczr4 : 12 [ 0 i+ !dac4 ] ;
offs1 : 10 !dac1 ;
offs2 : 10 !dac2 ;
offs3 : 10 !dac3 ;
offs4 : 10 !dac4 ;
poff1 : 10 @dac1 %emit ;
poff2 : 10 @dac2 %emit ;
poff3 : 10 @dac3 %emit ;
poff4 : 10 @dac4 %emit ;

send : iseqcs 08 \ 3B and iseqcs !8 ; *( sets CTL1 low to enable hss! )
notsend : iseqcs 08 \ 04 or iseqcs !8 ; *( sets CTL1 high to disable hss

a0 : variable ; -208 a0 ! ;
house : 8 [ i + dup = a0 a3 + 3B = "nl ] ;

*( temperature control tools )
temp1 : \ ff36 08 , \ ff37 08 ;
heat1 : \ 73C8 08 , \ 73CC 08 ;
temp2 : \ ff34 08 , \ ff35 08 ;
heat2 : \ 73C9 08 , \ 73CC 08 ;
temp3 : \ ff30 08 , \ ff31 08 ;
heat3 : \ 73CB 08 , \ 73CC 08 ;
temp4 : \ ff32 08 , \ ff33 08 ;
heat4 : \ 73CA 08 , \ 73CC 08 ;
temp : temp1 temp2 temp3 temp4 ;
tempwatch : temp "nl 30000 delay ;

sethc : \ 73CC !8 ;
sett1 : \ 73C8 !8 ;
sett2 : \ 73C9 !8 ;
sett3 : \ 73CB !8 ;
sett4 : \ 73CA !8 ;

heaten : \ ff \ ff00 !8 ;
heatdis : \ 00 \ ff00 !8 ;
heatdis ;
O sethc ;

gain : iseqcs !8 ;
time : variable ;

*( hostcom temperature routines )
scurrents : \ ff37 08 %emit \ ff35 08 %emit \ ff31 08 %emit \ ff33 08 %emit
stemps : \ ff36 08 %emit \ ff34 08 %emit \ ff30 08 %emit \ ff32 08 %emit
st_settings : \ 73C8 08 %emit \ 73C9 08 %emit \ 73CB 08 %emit \ 73CA 08 %
s_setting : \ 73CA 08 %emit ;

************************************************************************
*( Telescope mount controls )
************************************************************************

ha : 65528 a , ;
habyte : 65528 0B %emit 65529 0B %emit ;
slewon : \ ff 65530 !8 ;
trackon : 0 65530 !8 ;

*( hostcom telescope control routines )

s : 2 %out7 %out3 ;
s1 : 114 s ;
s2 : 54 s ;
s3 : 39 s ;
s4 : 99 s ;
s0 : 0 s ;
cw : s1 s2 s3 s4 ;
ccw : s1 s4 s3 s2 ;
westslew : cw iterate s0 ;
eastslew : ccw iterate s0 ;

wesT : 10000 westslew 1 %emit ;
wesM : 1000 westslew 1 %emit ;
wesC : 100 westslew 1 %emit ;
wesX : 10 westslew 1 %emit ;
wesI : 1 westslew 1 %emit ;
easT : 10000 eastslew 1 %emit ;
easM : 1000 eastslew 1 %emit ;
easC : 100 eastslew 1 %emit ;
easX : 10 eastslew 1 %emit ;
easI : 1 eastslew 1 %emit ;

*( Dome control )*

nenw_dome : 0 \ ff23 !8 ;
right : nesw_dome ;
nwse_dome : \ ff \ ff23 !8 ;
left : nwse_dome ;
start_dome : \ ff \ ff22 !8 ;
stop_dome : 0 \ ff22 !8 ;
azimuth : \ ff20 0 @8 %emit \ ff21 0 @8 %emit ;
enc : \ ff20 0 256 * \ ff21 0 + ;

micro_pulse : start_dome 50 delay stop_dome 500 delay 1 %emit ;
short_pulse : start_dome 100 delay stop_dome 500 delay 1 %emit ;
half_pulse : start_dome 500 delay stop_dome 500 delay 1 %emit ;
medium_pulse : start_dome 1000 delay stop_dome 500 delay 1 %emit ;
long_pulse : start_dome 5000 delay stop_dome 500 delay 1 %emit ;
spin_dome : start_dome 30000 delay 30000 delay 30000 delay stop_dome ;

*( define sequencer clock variables )

thp : variable ; *( parallel clock -- time high ) ;
tmp : variable ; *( parallel clock -- time mid ) ;
ths : variable ; *( serial clock ---- time high ) ;
tms : variable ; *( serial clock ---- time mid ) ;
 tls : variable ; *( serial clock ---- time low ) ;
tvarset : 0 0 0 40 4 thp ! tmp ! ths ! tms ! tls !
tvarset ;

status : temp1
    heat1
"Integration Time (mins.) = " ,t time = "n!

pclocks : " OP high = " ,t 2 adac4 = "n!
" mid = " ,t 1 adac4 = "n!
" low = " ,t 0 adac4 = "n!
" OM high = " ,t 5 adac4 = "n!
" mid = " ,t 4 adac4 = "n!
" low = " ,t 3 adac4 = "n!
" OS high = " ,t 8 adac4 = "n!
" mid = " ,t 7 adac4 = "n!
" low = " ,t 6 adac4 = "n!
" Vref = " ,t 9 adac4 = "n! ;

Vph : " VPhigh = " ,t 700 7 2 adac4 * - = .2 "sp " volts " ,t "n! ;

Vmh : " VMhigh = " ,t 700 7 5 adac4 * - = .2 "sp " volts " ,t "n! ;

Vsh : " VShigh = " ,t 700 7 8 adac4 * - = .2 "sp " volts " ,t "n! ;

Vpm : " VPmid = " ,t 450 7 1 adac4 * - 7 2 adac4 * - = .2 "sp " volts " ,t "n! ;

Vmm : " VMmid = " ,t 450 7 4 adac4 * - 7 5 adac4 * - = .2 "sp " volts " ,t "n! ;

Vsm : " VSmid = " ,t 450 7 7 adac4 * - 7 8 adac4 * - = .2 "sp " volts " ,t "n! ;

Vpl : " VPlo w = " ,t 300 7 2 adac4 * - 7 1 adac4 * - 7 0 adac4 * - = .2 "sp " volts " ,t "n! ;

Vml : " VMlow = " ,t 300 7 5 adac4 * - 7 4 adac4 * - 7 3 adac4 * - = .2 "sp " volts " ,t "n! ;

Vsl : " VSlow = " ,t 300 7 8 adac4 * - 7 7 adac4 * - 7 6 adac4 * - = .2 "sp " volts " ,t "n! ;

pvols ts : Vph Vpm Vpl "ni Vmh Vmm Vml "ni Vsh Vsm Vsl ;

seqw : iseqqs 38 6 bit and if repeat ;
go : seqgo seqw ;
g : go repeat ;

flush : notsend go go go go go ;
dark : flush send go ;
sec : 60000 * delay ;

csnt1 : 0 gain seqstop clks1 0 sethc heatdis 0 offs1 ;
csnt2 : 0 gain seqstop clks2 0 sethc heatdis 0 offs2 ;
csnt3 : 0 gain seqstop clks3 0 sethc heatdis 0 offs3 ;
csnt4 : 0 gain seqstop clks4 0 sethc heatdis 0 offs4 ;
cosinitall : csnt1 csnt2 csnt3 csnt4 slewon ;

quit ;
hostcom
BEGIN;
3 DELAY;
OS LOW EXECUTE;
10 FOR;
  2 DELAY;
  30 FOR;
    OM MID EXECUTE;
    tmp @ DELAY;
    OM HIGH EXECUTE;
    thp @ DELAY;
    OM LOW EXECUTE;
    16 DELAY;
  NEXT;
  1 DELAY;
  OS MID EXECUTE;
  800 FOR;
    OS HIGH EXECUTE;
    OS MID EXECUTE;
    2 DELAY;
    OS LOW EXECUTE;
    OS MID EXECUTE;
    2 DELAY;
NEXT;
5 DELAY;
NEXT;
2 DELAY;
OS LOW EXECUTE;
292 FOR;
  OP MID OM MID EXECUTE;
  tmp @ DELAY;
  OP HIGH OM HIGH EXECUTE;
  thp @ DELAY;
  OP LOW OM LOW EXECUTE;
  16 DELAY;
NEXT;
2 DELAY;
OS MID EXECUTE;
800 FOR;
  OS HIGH EXECUTE;
  OS MID EXECUTE;
  6 DELAY;
  OS LOW EXECUTE;
  OS MID EXECUTE;
  6 DELAY;
NEXT;
2 DELAY;
292 FOR;
  OM MID EXECUTE;
  tmp @ DELAY;
  OM HIGH OS LOW EXECUTE;
  thp @ DELAY;
  OM LOW EXECUTE;
  20 DELAY;
  OS MID EXECUTE;
2 DELAY;
400 FOR;
  OS HIGH EXECUTE;
  OS MID EXECUTE;
  8 DELAY;
  OS LOW EXECUTE;
  OS MID EXECUTE;
  2 DELAY;
  HOLDO ON EXECUTE;
  1 DELAY;
  HOLDO OFF EXECUTE;
  4 DELAY;

NEXT;
5 DELAY;
NEXT;
2 DELAY;
END;
quit;
hostcom
clks1 : 75 70 85 70 85 70 85 110 77 10 [ i- !dac1 ] ;
clks2 : 50 75 90 50 75 90 65 75 98 90 10 [ i- !dac2 ] ;
clks3 : 70 75 100 70 75 100 65 75 72 53 10 [ i- !dac3 ] ;
clks4 : 47 98 130 49 96 130 0 120 100 87 10 [ i- !dac4 ] ;
tempall : 0 sett1 0 sett2 0 sett3 0 sett4 0 sethc heatdis ;
offall : 4 offs1 60 offs2 26 offs3 7 offs4 ;
checkload : 10 delay ;
cosinitall : 2 gain seqstop clks1 clks2 clks3 clks4 tempall offall slewon ri

cosinitall ;
quit ;
hostcom
cلكs1 : 75 70 85 75 70 85 70 85 110 77 10 [ i- !dac1 ] ;
cلكs2 : 70 75 70 75 70 65 75 98 90 10 [ i- !dac2 ] ;
cلكs3 : 70 75 100 70 75 100 65 75 72 53 10 [ i- !dac3 ] ;
cلكs4 : 47 98 130 49 96 130 0 120 100 87 10 [ i- !dac4 ] ;
tempall : 0 sett1 0 sett2 0 sett3 0 sett4 0 sethc heatdis ;
offall : 4 offs1 6 offs2 3 offs3 7 offs4 ;
cosinitall : 3 gain seqstop cلكs1 cلكs2 cلكs3 cلكs4 tempall offall slewon r
cosinitall ;
quit ;
APPENDIX D

ETC-specific Driver Code

Introduction

This Appendix contains the Overseer computer driver code that is unique to the ETC. This Appendix has no relevance to the random reader of this thesis: it is meant solely as a reference for future workers on the ETC.
APPENDIX E

ETC Computational Algorithms

E.1. Centroiding

The location of a stellar image on a CCD is determined to a precision of a fraction of a pixel by a simple interpolation algorithm proposed by John Doty of MIT. The interpolation is the two-dimensional fit of an inverted parabola to the peak of a stellar image, using the value of the peak pixel of the image and the values of the two points on either side of the peak pixel. Implementation of the algorithm to the image profile in the x-direction and in the y-direction leads to a specification of the image location on the CCD.

The algorithm fits the three peak points to the formula \( z = ax^2 + bx + c \), where the x-values of the three points in the fit are \(-1, 0\) and \(1\). The maximum of this function is at \(-b/2a\), which can be calculated to be

\[
X_{\text{max}} = \frac{Z_{-1} - Z_1}{2(Z_{-1} + Z_1 - 2Z_0)}
\]

E.2. Median Filtering

The first level of Trigger sifting requires an image frame which represents to level of the sky brightness plus the bias level (the
The sky+bias level across the CCD is not constant, due to the effects of the vignetting of the lens and possible gradients in sky brightness; therefore, a simple averaging process for determining sky+bias level is not acceptable.

The median filtering process, adapted from use on MIT's Small Astronomical Satellite (SAS-3), is a straightforward method of calculating a representation of the sky without stars from an image of the sky with stars. In median filtering, a linear array of numbers is represented by the median value of that array, a value which can be considered typical for that array. In image median filtering, the image is median filtered first by rows and then by columns. The median values of the rows of an image are stored in a column vector. This column vector is then subtracted from each column in the image. The resultant image is median filtered along each column, the values being stored in a row vector.

The column vector is a representation of the typical sky+bias levels in the CCD, while the row vector represents deviations from typical values. The vector sum of these two (orthogonal) vectors yields the median-filtered image, which is essentially the original image with all stars, hot pixels and cosmic rays removed.
APPENDIX F

Thermal Analysis of the ETC

Introduction

The operating temperature of the CCD is determined by the cooling rate of the closed-cycle refrigerator and the ambient warming of the chips and how each varies with temperature. The rate of cooling is determined by the cooling power of the MFC-100 and the thermal resistance of the metal strap between the cold copper rods and the CCD cold sink. The ambient warming of the chip is due to heat gain through conduction and radiation from the walls of the camera. The final temperature of the CCDs cannot be calculated exactly because the variation of the cooling power with final temperature is not known. However, from the analysis that follows, it can be seen that the CCDs will reach a temperature well below -80°C.

F.1. Ambient Warming by Radiation

The rate of heat gain of the cold surfaces of the interior of the camera is calculated by comparing the rate of energy radiation by the cold surface with the rate of absorption by the cold surfaces of radiation from the camera walls. The rate of radiation of the cold surfaces is

\[ P_{\text{emitted}} = \sigma T_{\text{cold}}^4 A_{\text{cold}} \]

where \( T_{\text{cold}} \) is the temperature (° Kelvin) of the cold surface, \( A_{\text{cold}} \)
its area, $\varepsilon_{\text{cold}}$ its emissivity (and therefore also its emissivity), and $\sigma$ is the Stefan-Boltzmann constant. The radiation flux from a single surface of the wall is $\sigma T_{\text{wall}}^4$; the flux of radiation in a cavity (such as an empty camera body) is just $\sigma T_{\text{wall}}^4$. The geometry of the cold surfaces inside the camera will make the actual radiation flux hitting the surface somewhere between these values: the value of cavity radiation flux will be used here as an upper limit. Thus the rate of radiation absorption by the cold surfaces, $P_{\text{absorbed}}$, fulfills

$$P_{\text{absorbed}} < \sigma T_{\text{wall}}^4 A_{\text{cold}} \varepsilon_{\text{cold}}$$

The net rate of heat gain by the cold surfaces due to radiation follows

$$P_{\text{net}} < \sigma A_{\text{cold}} \varepsilon_{\text{cold}} (T_{\text{wall}}^4 - T_{\text{cold}}^4)$$

For $T_{\text{wall}} = 288^\circ K$, $T_{\text{cold}} = 173^\circ K$, $\varepsilon_{\text{cold}} = 0.08$ and $A_{\text{cold}} = 100 \text{ cm}^2$, $P_{\text{net}} < 0.27$ Watts. A rough estimate of the effects of geometry gives the value of heat absorption by the cold surfaces to be

The rate of heat transfer to the cooling probe can be calculated in a similar fashion. The net rate of heating is

$$P_{\text{net}} = \sigma \varepsilon_{\text{probe}} A_{\text{probe}} (T_{\text{manifold}}^4 - T_{\text{probe}}^4)$$

For $A_{\text{probe}} = 200 \text{ cm}^2$, $\varepsilon_{\text{probe}} = 0.6$, $T_{\text{manifold}} = 288^\circ K$ and $T_{\text{probe}} = 173^\circ K$, the rate of radiation heat input to the probe $P_{\text{net}} = 11.1$ Watts.
It should be noted that these calculations are based on the assumption that the walls of the camera and manifold are in thermal contact with an infinite heat bath. In actual fact, they are not, and their temperature is observed to drop when the system is cooled. This difference will simply reduce the rate of heat flow to the cold surface; thus, the numbers calculated above are good upper limits on the heat flow to the camera's cold surfaces.

F.2. Ambient Warming by Conduction

The rate of heat gain of the cold surfaces due to conduction can be roughly calculated from a simple kinematic model of the gas inside the camera. At the operational pressures inside the ETC cameras (\(\sim 10^{-5}\) mm Hg), the mean-free-path of an air molecule is several meters, so thermal molecules from the walls will strike another wall or a cold surface before colliding with another molecule. The spectrum of velocities striking a cold surface can therefore be considered a Maxwellian, based on the temperature of the walls.

The rate of energy deposition by conduction into a cold surface equals the rate of collision of molecules into the surface times the average energy deposited per collision. The rate of collision of molecules of number density \(n\) and velocity \(v\) onto area \(A\) is \(nvA\). If the average energy deposited is half the molecule's kinetic energy, the rate of energy deposition for particles of velocity \(v\) is

\[ P(v) = nvA(\frac{mv^2}{4}).\]
By integrating over the Maxwellian velocity distribution, the total rate of energy deposition by all molecules becomes

\[ P_{\text{conduction}} = \frac{\int_0^\infty P(v)v^2e^{-mv^2/2kT}}{\int_0^\infty v^2e^{-mv^2/2kT}} \]

This reduces to

\[ P_{\text{conduction}} = 0.799m^{1/2}A(\kappa T)^{3/2} \]

For \( T = T_{\text{wall}} = 288^0\text{K} \), \( m = 14m_{\text{proton}} = 2.3 \times 10^{-23}\text{g} \) and \( A = 100\text{ cm}^2 \),

\[ P_{\text{conduction}} = 1.3 \times 10^{-14}n \text{ Watts} \].

The value of \( n \) at 1 \( \mu \text{m} \) Hg pressure is \( 1.5 \times 10^{-15} \), so \( P_{\text{conduction}} = 20.0\text{ Watts/\mu m} \). At a pressure of \( 10^{-2}\mu \text{m} \), this rate is 0.2 Watts.

F.3. Cumulative Warming Effect due to Conduction and Radiation

The net rate of warming of the ETC CCD cameras and manifold at a pressure of 0.01 \( \mu \text{m} \) Hg is <15 Watts at -80\text{C}, according to the calculations in the sections above. Since the MFC-100 closed-cycle refrigerator is rated to deliver >"200 Watts of cooling power at -80\text{C}, the ETC CCD cameras should reach temperatures between -85\text{C} and -95\text{C}.\]