University of Durham
Astronomical Instrumentation Group

# NAOMI

**Nasmyth Adaptive Optics for Multi-purpose Instrumentation**

## The Real Time Control System
## *Programmers Guide*

wht-naomi-24

Version: 0.3   24 December 2002

Authors: Richard Myers[1], Stephen Goodsell[2]

1. University of Durham, Department of Physics
2. Isaac Newton Group of Telescopes

## 1. Scope

This document describes how to programme the real-time control system of NAOMI, how the existing real-time programmes are structured and how they can be modified.

A separate document "*NAOMI Hardware Reference Manual*" provides a description of the real-time control electronics rack.

A separate document *"The GP Messaging Library"* describes the general purpose asynchronous messaging system used to configure and monitor the real-time system.

A separate document "*Real Time Control System User's Guide*" provides a description of the use of the low-level engineering level command line access methods, including setting up and executing software configurations as well as the manipulation and monitoring of a running system.

A separate document *" Naomi Engineering and Control Program: TopGui"* provides a description of higher-level control, monitoring and sequencing of real-time processes.

## 2. Acknowledgements

NAOMI's real-time software is derived from Durham University's *Electra* software and many of its files and nomenclature use the word `Electra`. Most of the underlying design of the software and much of the programming is by David Buscher. Other design features were evolved in discussion with Andy Vick. Craige Bevil, David Buscher, Nigel Dipper, Peter Doel, Szilveszter Juhos, Patrick Morris, Richard Myers and Andy Vick evolved the workstation client-side software. (The staff of the UKATC produced the Mechanism and WFS control software but this software has its own documentation and this *Guide* does not describe it further.)

Parts of this *Guide* borrow freely from David Buscher's documentation.

## 3. Overview

NAOMI stands for Nasmyth Adaptive Optics for Multi-Purpose Instrumentation. It is the Adaptive Optics (AO) system on the William Herschel Telescope (WHT).
The purpose of this document is to enable maintenance and development of the NAOMI real-time control system. An understanding of the purposes and the basic construction and operation of NAOMI is assumed. An Introduction to Astronomical Adaptive Optics is available in the book of that name by John. W. Hardy (OUP). Introductory information on NAOMI's overall design requirements may be found on http://aig-www.dur.ac.uk/fix/projects/projects_index.html. The accompanying Technical Description is useful but describes the design rather than the final system. An updated general description of the as-built NAOMI system is in preparation by A.J.Longmore (UK ATC at ROE) and R.M.Myers (Durham) at the time of writing this document.

This *Guide* includes a brief history of the real-time system software, sufficient only to understand the nature of the collected code. It describes the Revision Control system used, how to set up as a developer and gain access to the source code, and where the key files are located in the software directory tree.

Subsequent sections describe the hardware architecture of the real-time control system (briefly) and the corresponding software architecture. The structure of the principal programmes is then explained. The method of building new programmes and generating new real-time configurations is then described, followed by brief descriptions of the nature of the connections to workstation client processes.
The final sections document the software files in detail, and briefly introduce some ways in which the software might develop.

## 4. Glossary

| | | |
|---|---|---|
| AO | Adaptive Optics | partial removal of the effects of atmospheric turbulence on image quality |
| BSP | Bulk Synchronisation Parallelism | Real-time programming methodology used in NAOMI (University of Oxford) |
| C40/C44 | Texas Instruments DSP | DSP optimised for interprocessor communications |
| CCD | Charged Coupled Device | light sensitive detector |
| DM | Deformable Mirror | wavefront phase corrector |
| DSP | Digital Signal Processor | processor optimised for signal processing |
| Electra | Durham University AO system, software and DM | See History section below |
| EPM | Electra process monitor | Process variable database portion of the NAOMI sequencer |
| FSM | Fast Steering Mirror | wavefront tip-tilt corrector |
| GHRIL | Ground-based High Resolution Imaging Laboratory | Nasmyth platform of the WHT |
| GP | General Purpose message protocol | Asynchronous messaging system used to/from and between C40s |
| INGRID | ING infrared camera | Can be used as a science camera for NAOMI |
| ISR | Interrupt Service Routine | Code executed on processor interrupt |
| LoveTrain | Synchronisation packet | Synchronous messaging system used between C40s in the WFS and DM rings during ISR execution. |
| OASIS | Optical Spectrograph | Integral field spectrograph to be used with NAOMI. |
| Python | High-level language | Scripting language for NAOMI |
| Ring Leader | C40 brokering transaction IDs | C40 used to coordinate parameter block transactions within a c40 ring |
| Sequencer | Process launcher/monitor | Processes used by NAOMI supervisory software |
| WFS | Wavefront Sensor | Optical system for wavefront phase distortion measurement |

## 5. Typographic Conventions

`Code is indicated by this type format.`

Future expansion notes in the main text are indicated in this type format.

## 6. History

NAOMI's real-time software is derived from Durham University's *Electra* software and many of its files and nomenclature use the word `Electra`.

The Electra AO system's ThermoTrex 228-degree of freedom mirror, with its internal figure sensing system, was adopted for the NAOMI system in 1996. This followed a PPARC review recommendation and a decision by the NAOMI consortium (then ING, Durham, RGO, ROE). A Memorandum of Understanding between ING and the University of Durham describes the transfer and reciprocal arrangements.

The Electra mirror requires unique control system features because of its internal feedback capability. The design of Electra's real-time control software predated the inception of NAOMI and is essentially a superset of NAOMI's requirements in terms of the flexibility of both the underlying architecture and the visualisation system. On the other hand, it interfaces to different WFS and FSM hardware and does not cover NAOMI's operational requirements. Rather than write wholly new software on this scale, the Electra software was adopted as a whole and new interfaces and a unified engineering GUI (TopGui) were added *within* the Electra software structure. New documentation requirements were added. This document is part of that process.

The NAOMI mechanism and WFS control software is new and was written at UK ATC. It does not need to be within the Electra software structure and, indeed, is not.

The higher-level coordination software (the Sequencer and TopGui) necessarily forms part of the Electra structure because of its need to communicate intensively with the real-time control system (e.g. for WFS image display).

The adoption of the Electra software package as a whole has resulted in the availability of many little-used library routines and applications. Several of these applications (WFSAlign, MirrorMimic, DataDiag) were of great use during NAOMI's commissioning phase but have now been superseded by TopGui. In all these cases it is not proposed to provide extensive documentation of these facilities and it is anticipated that they will be removed from the supported release in due course.

The Electra software system supports development in two languages: C and Python (with Numeric extensions). Python provides both a scripting facility and a rapid development high-level language that is also efficient in execution.

## 7. Accessing and Developing Real-Time Source Code

This section is freely adapted from David Buscher's *Quick Guide to Electra Software Development*. It describes the method for developing Electra workstation and c40 code in general. It also describes the location of the principal source code for the Real-Time control system parts of the Electra. The details of how to build, maintain, modify and enhance this particular code are deferred to subsequent sections.

## *7.1 The Electra Development Process*

Electra uses the *BCS* baseline control system to allow multiple developers to work on one body of code. Not only source code but also documentation is all stored in one central directory tree, called a *baseline*. Software developers maintain their own private copies of this baseline, called *staging areas* and the BCS system takes care of the synchronisation of the private copies and the baseline. For the Electra developer, there are three copies of the source trees to be aware of, which are rooted in

`/software/Electra_src_tree`
`$STAGING` (usually `$HOME/Electra`) and
`/software/Electra`

The first of these is the baseline, which contains the ‘master’ copies of the source files. No object files or executables are ever built in this tree, and manipulation of this tree is mostly indirect via BCS commands.

The second tree, `$STAGING,` is a private staging area for a given user. It can have any name but we have given it the name `$HOME/Electra` for concreteness here. Normally its subdirectories contain pointers (symlinks) to read-only files in the corresponding subdirectories of `/software/Electra_src_tree`. When these files are *staged* real writable copies replace the symlinks. The user can then build modified versions of executables in their staging areas without affecting installed executables and other users. The RCS revision control files within each directory are common to all users, however, and locks are used to prevent two users staging the same file at the same time.

The third (partial) tree is a public staging area. It is like the private staging area, but is used for sharing built versions of the code, e.g. libraries etc.

The normal development process is to develop and test code in the private staging area, and once it has been tested to install the built versions in

`/software/Electra`

and the source code in

`/software/Electra_src_tree.`

The `/software/Electra/bin` directory contains executables, the `/software/Electra/lib` directory contains object libraries and configuration files for the built code. Likewise the `/software/Electra/include` directory contains common include files which are shared between packages.

The files in `/software/Electra/{bin,lib,include}` are the stable versions of these files, while those under the developers' private trees are developmental versions. Compiler search paths for include files and library files may search the user’s private directories first, and then the relevant `/software/Electra` directory. In this way, a developer first picks up the versions of the code s/he is currently developing in preference to the stable version. Clearly though, once development of a package has been completed, the private directories should be cleaned out. This allows the latest version to be picked up from `/software/Electra`, which is helpful if someone else later updates the code. Similar comments apply to the order of `bin` directories in the shell `PATH` variable.

There are several scripts to aid the development process. These scripts are installed in the `/software/Electra/bin` directory, and the originals are in `/software/Electra_src_tree/tools` (and can themselves be accessed from, and staged into, `$STAGING/tools`). The functions of the two most commonly used scripts are as follows:

| | |
|---|---|
| `bcs_mkdirs` | Sets up the directory tree for a developer |
| `bcs_publish` | Checks in source files which have been checked out, and updates the baseline copy of the files. Used to “publish” any updated versions of the source code in a given tree/subtree. |

The `/software/Electra_src_tree/config` directory contains files which are included by makefiles to configure the make process. These files can be accessed from `$STAGING/config`.

The `/software/Electra_src_tree/scripts` directory contains startup scripts for setting up the development process and also for starting daemons which are used at runtime. These files can be accessed from `$STAGING/scripts.`

Historical note: the `/software/Electra_src_tree/docs` directory tree contains some early documentation about the Electra system. It is mostly in the form of LaTeX source files. A makefile in these directories converts these files to HTML and installs them in an HTML tree.

All the rest of the subdirectories of `/software/Electra_src_tree`, and the corresponding directories of `$STAGING/` are C and Python source code trees. These trees contain by convention subdirectories `libsrc` and `appsrc` to hold code for building libraries and executables respectively.

## *7.2 Setting up a user account for Real-Time software development*

Here is a recipe for how to develop a new package. Skip the stages you have already done as necessary.

1. Set up your login files. Edit your `.cshrc` file to include the lines

   ```
   setenv BASELINE /software/Electra_src_tree
   setenv STAGING $HOME/Electra
   source /software/Electra/bin/setup.csh
   ```

   Then log in again or re-source your `.cshrc`. You will obviously have to adjust this process if you do not use `csh` or `tcsh.`

2. Create a private staging area using

   ```
   mkdir $STAGING
   ```

   Type `bcs_mkdirs`. This command should be used any time someone else has created new directories in the baseline.

3. Make a new subdirectory if a totally new package is being made. Type

   ```
   cd $STAGING
   mkdirhier myPackage/appsrc
   cd myPackage/appsrc
   register_file Makefile
   cp ../../c40Comms/appsrc/Makefile .
   ```

   This will create the directory and make a mirror copy in the baseline. The example shows stealing a makefile from another directory as the starting point. This should be edited to suit the package being built.

4. Register any other new files with the BCS system e.g.

   ```
   register_file mysrc.c mysrc.h
   ```

   Edit files in this area, and compile and test them.

5. Once the files at least compile, the source files can be checked into the RCS system for version control

   ```
   bcs ci -l mysrc.c mysrc.h
   ```

6. When the files compile and have been tested, install the built files in the (public) baseline directory and put the latest versions of the source files into the baseline.

   ```
   gmake install
   bcs_publish .
   ```

7. There is a (little-used) process to make a *release*:

   ```
   cd ~/Electra
   bcs_tag_tree myPackage
   ```

   This will tag all the RCS files in the myPackage tree with a tag of myPackage1. The next time you do this the files will be tagged with myPackage2 etc.

### *7.3 C40 programs*

The c40 DSP processors are described further below. For now it suffices to note that they are the main processors for actual real-time operations.

By convention, c40 executables are denoted by the `.x40` suffix and the object files are denoted by the `.o40` suffix. This allows two versions of a given program, one running on c40s and one on workstations, to be built in the same directory.

In a directory where c40 programs are to be built, the makefile, like all Electra makefiles, should include the line

```
include $(STAGING)/config/Electra.mk
```

A target build line *might* look something like

```
c40Echo.x40 : c40Echo.o40
        $(C40_CC) $(C40_LDFLAGS) -o $@ $^ \
        -L$$STAGING/lib -L/software/Electra/lib \
        -lGPmsg.lib -lUtil.lib
```

The `c40_cc` program is a front-end to the TI C40 compiler that makes it appear much more like a standard Unix compiler.

The library files shown in the make recipe provide GP messaging and utility functions (currently only the exeception-handling facilities). The `c40_cc` compiler front end automatically includes the C run-time system.

### *7.4 Real-Time Program Directories*

After setting up a user account for NAOMI (ELECTRA) RealTime software the STAGING environment variable will be defined (generally as `/home/user/Electra`). The setup procedure should also have produced a subdirectory (amongst others) called:

`${STAGING}/RealTime/` - base directory for real-time source code

This directory in turn has the following subdirectories:

`appsrc/` - real-time workstation client applications (low level engineering level)
`pythonModules/` - workstation python client support libraries
`libsrc/` - c40 libraries
`WFS/` - c40 WFS ring code
`StrainGauge/` - c40 Strain Gauge ring code

#### 7.4.1 RealTime/WFS: key files

The following file in the RealTime/WFS directory are the ones which normally need to be edited to alter real-time WFS processing behaviour or add new control parameters.

| | |
|---|---|
| `AlgNaomiInterleave.c` | Contains the WFS centroid estimation algorithm |
| `AlgMartini.c` | Contains the WFS reconstructor and tip-tilt-piston calibration |
| `NaomiGenericBSP.c` | Hosts AlgNaomiInterleave. This is the main function of `NaomiGenericBSP.x40`, which loads onto all the C44s in the WFS ring apart from the mirror CPU |

| | |
|---|---|
| | (GP number 4). |
| NaomiMirrorBSP.c | Hosts AlgMartini. This is the main function of NaomiMirrorBSP.x40, that loads onto the mirror CPU (GP number 4) of the WFS ring. |
| Makefile | makefile (uses ../Makefile) |

### 7.4.2 RealTime/StrainGauge: key files

| | |
|---|---|
| AlgSGadc.c | Strain Gauge reading, calibration and servo algorithm |
| AlgSGmirror.c | Mirror output algorithm |
| AlgSGtimer.c | Strain Gauge ring timer algorithm |
| SGBSP2.c | Hosts all the above algorithms. This is the main function of SGBSP2.x40 that loads onto all the C44s in the strain gauge ring. |
| Makefile | makefile (uses ../Makefile) |

### 7.4.3 SharedInclude

The ${STAGING}/SharedInclude directory contains ‘master’ python files which are used to generate C and python include files. These ensure that workstation and C40 programmes have the same correspondence between C enum types (and python strings) and command-identifying numbers in GP communication packets.

ParameterBlocks.py - is used to identify new parameter block transactions
Makefile – gmake include will regenerate the include files and install them

# 8. Processor Architecture

## *8.1 Summary of required functions*

The required NAOMI real-time processing functions are:

### 8.1.1 WFS data processing

?? Receive Wavefront Sensor (WFS) pixel data from the NAOMI WFS controllers. The controllers are of SDSU type and have the “Steward” port option, which allows direct access to parallel digital data output (i.e., without the data being transmitted along the VME bus). There are two CCDs in the WFS, which may be synchronised or operated independently. Note that the real-time system can readout from either CCD or from both if they are synchronised. It cannot be run if the CCDs are both running unsynchronised (and indeed if they are both running unsynchronised and the real-time system has data reception enabled for both, it will fail).
The CCDs in the WFS are EEV CCD39 chips and each has frame transfer buffers connected to four separate readout ports. The SDSU controllers interleave pixels from the quadrants of each CCD. The data from each quadrant therefore become available concurrently, starting at each corner and progressing by rows towards the centre.
For further details of the SDSU controllers and the NAOMI readout modes see

NAOMI WFS CCD CAMERA CONTROL (V3 or higher) by D. Ives (ATC) and NAOMI ICD 101 or higher by X. Gao (ATC).

?? Determine the current readout mode of the CCD from the header data preceding each frame. Adjust the size of the expected data transfer and the parameters of the signal-processing algorithm (see below) accordingly. Different modes support different numbers of subapertures and have different on-chip binning and skipping. The mode also determines if the two CCDs are synchronised.

?? Process the WFS pixel data to produce centroid estimates of the WFS spot positions, and therefore, estimates of the local wavefront slopes. This involves removing a background level and, optionally, a sky gradient, and dividing the pixels into “boxes” (e.g., a 4x4 pixel square for each subaperture).

?? Generally there is one WFS subaperture per deformable mirror segment but in some modes a subaperture covers several mirror segments. If this is the case then copy the centroid estimates as required.

?? Remove an optional offset from the WFS centroid estimates. This is to account for the shape of the “starting” figure on the DM.

?? Apply a servo algorithm to the current estimator in order to update the current mirror segment x,y slope demands.

?? Perform a matrix multiply on the vector of x,y slope demands. There is one x,y measurement per subaperture. The result of the matrix multiply is a vector of piston values: one per subaperture.

?? Convert the x,y,piston command for each mirror segments into the A,B,C equilaterally sectored actuator commands that are actually required to drive the mirror. The resulting command vector is the mirror input demand.

### 8.1.2 DM data processing

?? When strain gauge feedback is operating, compare the actuator input demand to a calibrated digitised sample of the strain gauge voltages and adjust the final demand to the deformable mirror using a servo algorithm. This algorithm must sample and update substantially faster than the WFS servo algorithm.

### 8.1.3 Latency

?? All the above WFS and strain gauge processing must be performed with timing uncertainties of no more than a few microseconds (i.e. it must have deterministic latency). There are also stringent upper limits on the magnitude of the processing latency. If it substantially exceeds 1 ms then performance will in general degrade. In practice, the sensible upper limit to the latency of the WFS processing depends on the current integration time of the WFS. If it is already 10 or more milliseconds then the relative effect of another 0.2 milliseconds of processing latency may not be decisive to the level of performance obtained. Similar arguments may be applied to the time taken to read out in the WFS CCD modes where little skipping (windowing) and binning takes place. Reducing processing latency is always beneficial but the actual magnitude of the benefit needs to be evaluated for the likely operational conditions. A NAOMI model exists (by Richard Wilson, Durham) which may be used to estimate operational benefits of any planned change of this kind. For the present purpose of describing the existing implementation, it may be taken that some form of interleaving of the readout of the CCD and the estimation of the WFS centroids is highly desirable in most cases. That is to say that, so far as possible, WFS readout and WFS data processing, should be concurrent.

### 8.1.4 Commands

?? All key run-time parameters of the above processing must be capable of being updated on-the-fly. This means that updates can take place with all the control loops closed and with all parameters changing synchronously. There must be no missed samples (of WFS or SG data) or delayed processing of samples.

### 8.1.5 Status

?? The state of run-time parameters must be able to be retrieved at any time.

### 8.1.6 Diagnostics

?? Diagnostic samples of input, intermediate and output data must be available from the real-time controls system in a streaming fashion without interfering with the real-time data flow.

## *8.2 C40 processors*

The Texas Instruments TMS320C40 Digital Signal Processors (DSPs) was selected as the main processor for implementation of the real-time control system for NAOMI. It has the following features:

?? Optimised for digital filtering. A Multiply-Accumulate (MAC) and two data moves can be executed by a single instruction and performed in a single processor cycle. The MAC is also the key instruction for servo algorithms and for matrix multiplication, which are key operations for NAOMI.

?? Interprocessor communications ports. Each processor has six processor-to-processor bi-directional links. The link hardware is 8-bit 20MBs$^{-1}$ parallel but from the programme point of view the minimum data quantum is 32 bits. Each C40 has an 8x32bit FIFO buffer on input and output. Interprocessor communications therefore have 16-deep 32-bit FIFOs in each link.

?? Each communications port is connected to a separate on-chip Direct Memory Access (DMA) engine which may be programmed to move data between the communications port and memory without further programme intervention. With suitable hardware the communications ports can be used to transfer data to or from external systems as well as between c40s.

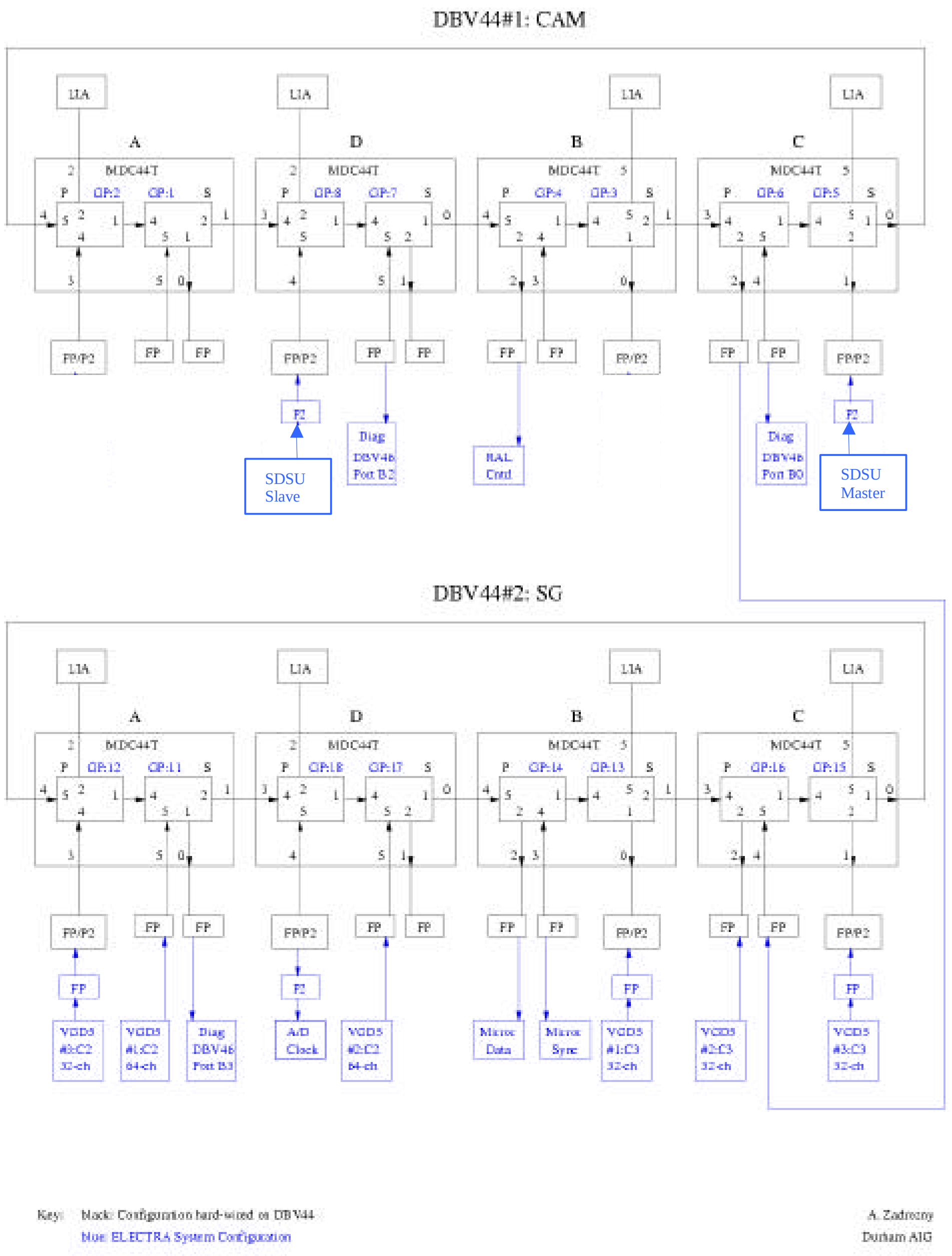
?? Separate instruction and data busses.

?? JTAG scan chain hardware for external debugging access to processor registers and hardware breakpoints. It is possible to set up synchronised hardware breakpoints on several processsors.

Note that the TMS320C44 is actually used in the NAOMI control system. This differs from the c40 by having 4 instead of 6 communications ports. A full c40 is used for the diagnostics processor.

## 8.3 Interconnections

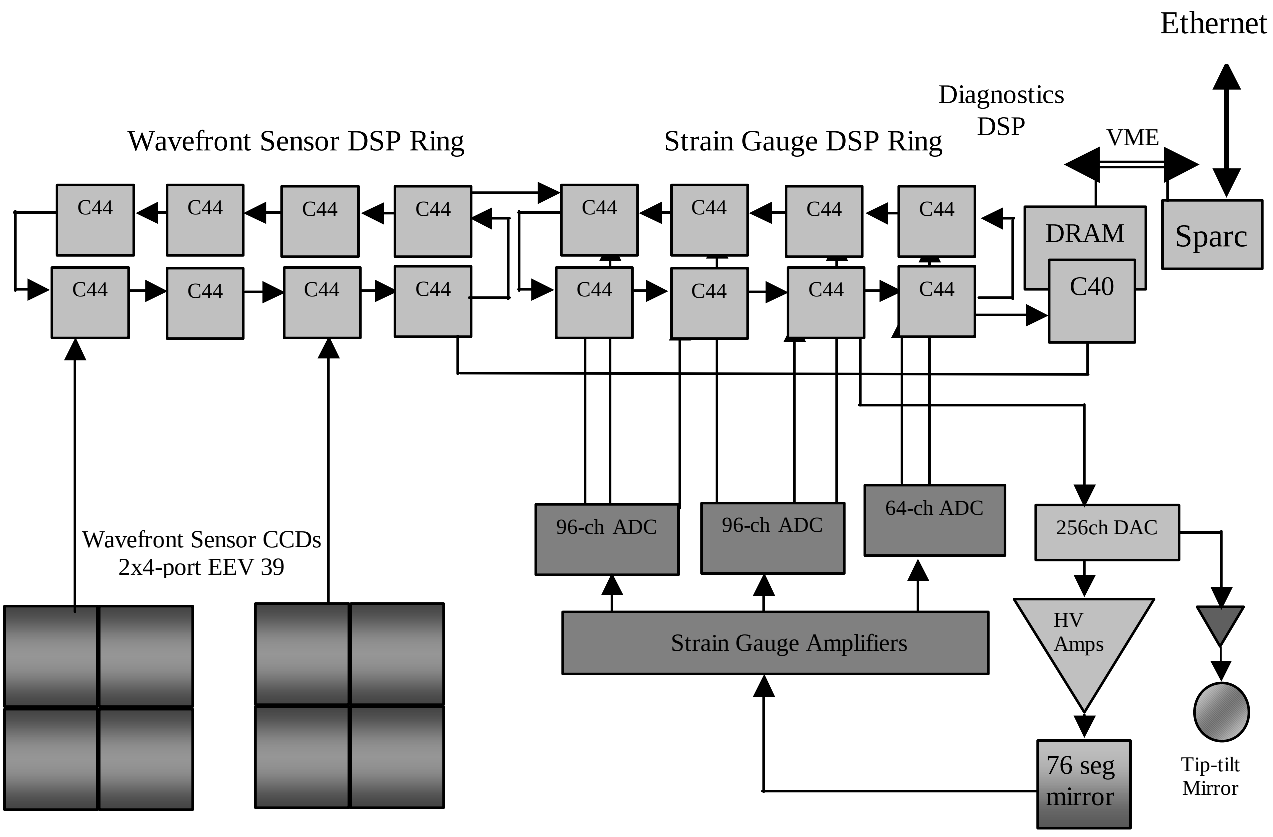
The figure below shows the connections of the c44 communications for NAOMI. Interprocessor connections and external connections are indicated, as are the assigned GP numbers for each processor.



ELECTRA/NAOMI Comm-Port Configuration Diagram

## *8.4 Ring structures*

16 of the c40s have their communications ports interconnected so as to form two rings of eight processors each. One ring is used for the WFS algorithms and one ring for the Strain Gauge processing.  The figure below shows how the c40 rings connect to the external sensing, actuation and diagnostics systems.



Together with the software architecture described below, the ring structures enable the real-time processing requirements to be fulfilled with fewer DSPs than with an equivalent farm architecture. The development overheads are also reduced.

## *8.5 WFS ring*

The WFS ring contains 8 c44 processors (GP numbers 1 to 8). Two are connected via communications ports and interface casrds to the data output from the “Steward” parallel output ports of the SDSU controllers for the WFS CCDs. The interface cards, produced by Durham, buffer the incoming data and respond to synchronisation and control signals from  further communications part which carries command from the c40s. P. Clark describes the interface cards in the *NAOMI Hardware Reference Manual*.

The 8 processors of the WFS ring are hosted on a single VME card: a Blue Wave Systems (UK) DBV44 card. This card is a motherboard, which hosts 4 TIM processor daughtercards, each of which carries 2  C44s. The DB44 card routes communications ports to the VME P2 bus as well as to the panel front, and it is P2 ports which carry the WFS interfaces. Panel front connections are used for the communications ports that link the WFS ring to the strain gauge ring (one port) and to the diagnostic CPU.

The DBV44 has Link  Interface Adapters (LIAs) which connect some of the communications ports to the VME bus, where they appear as memory mapped registers. The four LIAs are used to provide communications with the VME host processor (see below) and thence, via Ethernet to the outside network.

## *8.6 Strain Gauge Ring*

The Strain Gauge (SG) processing ring is designed to be as similar possible in both hardware and software to the WFS ring (above). It too consists of 8 C44 processors (GP numbers 11 to 18) hosted on a DBV44 VME card. In this case the sensor data come from the strain gauge Analogue to Digital Convertors (ADCs). The ADCs are contained in 3 VME cards produced by Pentland, UK, which are specialised in that they have C40 communications port outputs. There are 256 16-bit ADC channels altogether and each of them is capable of sampling at 85kHz and delivering the data via a c40 communications port. The configuration of these ADC cards is done via the VME bus from the `c40Comms` process on the VME host processor (see below). They are configured to deliver the digitised data via 6 communications ports to 6 processors (the ADC CPUs) of the strain gauge ring: 2 ports carry 64 strain gauge channels each and 4 ports carry 32 strain gauge channels each. The readout order is a little complicated and downloadable tables are used by the c40s to reorder the data. The connection to the processors is organised so that the 2 64-sample channels could have half of their respective data transferred to unloaded  neighbour C44s in order to achieve a better load balance of 32 channels per processor (future upgrade).

An output port from one of the C44s (the timer CPU) is connected to a Durham interface card, which produces a trigger signal for the ADCs. Therefore the SG ring must provide its own source of interrupts, as it is responsible for the conversion trigger to the ADCs. This is in distinction to the WFS ring where the SDSU controllers free-run and provide the interrupts.

The output from the SG ring is carried by a panel front communications port to a Durham DAC interface card and thence to the Durham DAC rack. A synchronisation signal is carried to the same interface on a separate communications port. There are 256 13-bit DAC channels of which 228 are used by the DM. Two DAC channels (30 and 31) are used for the FSM and are connected to the Zeiss (Jena) driver rack. The DM analogue signals go to the Durham drive amplifier rack.

Like the WFS ring, the SG ring uses its LIAs to communicate with the VME host processor. Likewise it has a panel front communications port connection with the diagnostic CPU.

## *8.7 Diagnostic CPU*

The diagnostic CPU is a C40 (GP number 9) located on a Blue Wave Systems (UK) DBV46 VME card. The CPU is one of two fitted as standard to the DBV46. The other

CPU (GP number 10) is spare capacity. The DBV46 can also host TIM cards carrying additional processors but none have been fitted.

The DBV46 has dual-ported memory, which is read/write accessible both from the C40s and from the VME32 bus. It is this channel that is used for downloading diagnostics data to the VME host.

Unfortunately the DBV46 has no LIA connections and therefore must be booted indirectly from one of the WFS ring CPUs via its front panel communications port connection.

### *8.8 VME Host*

The current VME host is a Force VME card carrying a SPARC 5V processor. It runs the Sun Solaris operating system and can therefore mediate between the C40s and the external network. It has its own disk and is an autonomous computer. A console may be attached if required via an RS232 connection. Normal communication goes via a 10BT Ethernet port.

For mostly historical reasons this processor currently carries the C40 cross-development software. The reason is that one of the C40 development tools, the DB40 debugger, must run on this computer in order to access the JTAG scan chain via the VME bus. This debugger is only used very rarely now and there is no fundamental reason why this computer should continue to host the other C40 cross-development tools.

The current Force card will probably be replaced with an updated one based on an UltraSPARC processor. Such a card will be fitted with a 100BT Ethernet port.

### *8.9 Workstations*

The control and monitoring processes of the remainder of the NAOMI control system may be distributed anywhere on the Internet, at least in theory. Some of them have been ported to SGI and Linux hardware, for example. In practice they are run on the NAOMI workstation, `navis`, a dual processor UltraSPARC system.

# 9. Software Architecture

### *9.1 Summary of required functions*

The software architecture fulfils the overall requirements given in the hardware architecture section (above) and within the constraints imposed by the selected hardware (above). It is important to also bear in mind the history of the design (above), i.e., that the NAOMI software is derived from the Electra software and that the Electra requirements were in some respects a superset of those for NAOMI. Some software features are therefore not strictly traceable to NAOMI’s requirements. These features generally take the form of additional flexibility and diagnostic capabilities. An example of additional flexibility is the ability to perform synchronised switching of interrupt processing algorithms as well as just run-time parameters.

One important example of an Electra-specific feature is a code design that allows the WFS data from a given CCD to be transmitted as separate quadrants to more than one CPU. This is indeed partially exploited by NAOMI in its support for more than one

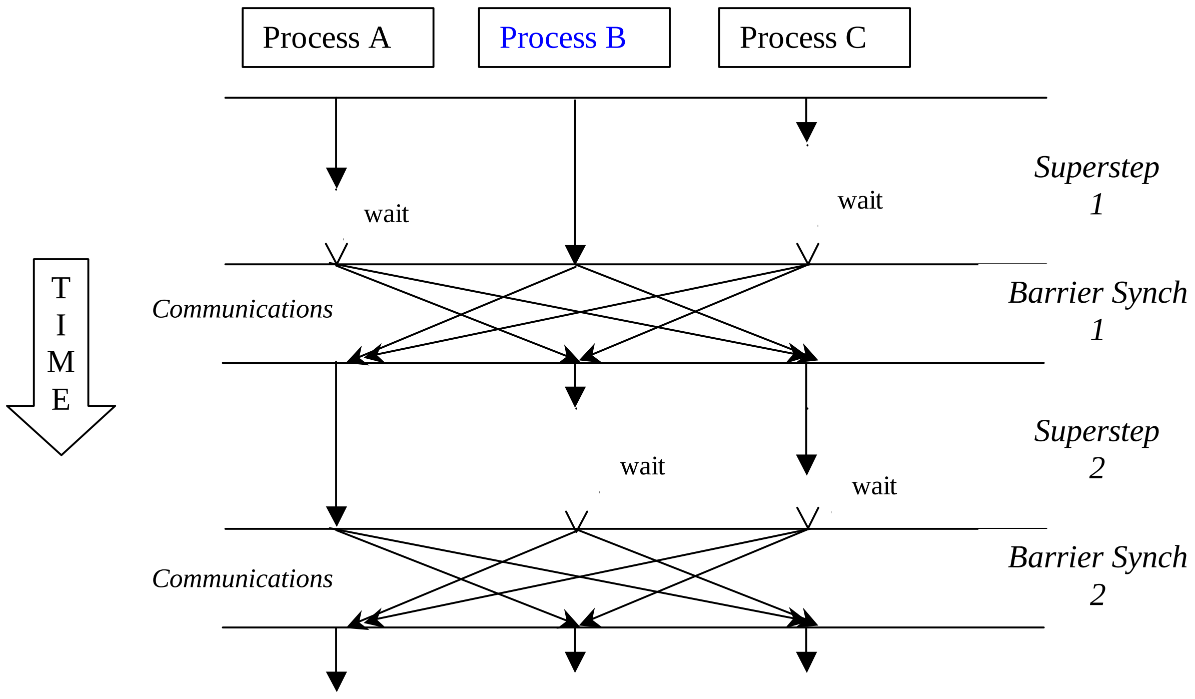
WFS CCD. Also some per-quadrant processing is really necessary because of the quadrant dependence of the bias and the (interleaved) read order. It would therefore be difficult to untangle in retrospect how much of the per-quadrant processing is really superfluous to NAOMI's requirements.

The software User Requirements Document for NAOMI as a whole lists various direct requirements for real-time processing and indirect ones for control, status display and visualisation. These are effectively covered by the requirements given in the above hardware architecture section and the Electra capabilities. Note, however, that some requirements, for example, 3D diagnostics, though met in general by Electra's capabilities, have not yet been commissioned for NAOMI use due to time constraints and their low priority in practice.

In developing a parallel real-time processing system, a very major issue is controlling the development time required to generate a stable structure and to subsequently maintain and enhance it. Following studies of parallel operating systems and an experimental evaluation of one proprietary language, it was decided to adopt selected aspects of the Bulk Synchronisation Parallelism (BSP) methodology developed by Oxford University, UK. The aims of this methodology are ease of development and stability. Its adoption has been successful. Consequently there is no proprietary code within the real-time software. The aspects of BSP not adopted have normally been omitted for obvious reasons: e.g., random process placement would not work with fixed interface connection nodes.

### 9.2 Introduction

The BSP methodology requires that all processes wait to communicate until a global synchronisation step. All processes wait for this step to complete regardless of whether they have anything to transmit or receive. The step is not complete until all interprocess communications have finished. The global progress of the parallel processing system is therefore divided into supersteps by these synchronisation barriers. The figure below illustrates the general idea.



The advantage of all this is that it makes parallel software development much easier in practice. Each process ‘knows’ that if it is in superstep *n* then every other process must be in superstep *n* too. It also ‘knows’ that the data available to every other process is precisely that which is dictated by its being in superstep *n*. The developer can safely program the message passing of any process at any stage knowing the states of all other receiving and transmitting processors. Furthermore, if an error were reported by process *X* whilst in superstep *m* and this were thought to be due to the activities of another process, then the developer can ascertain, simply by inspecting code, the program and data states of all other processes within the system at the time of the error to within the resolution of one superstep: they will all be at their respective superstep *m*. Without such a scheme, complex and time-dependent webs of interprocess dependency can develop.

In practice, the BSP methodology is applied separately to the two rings of C44 processors, and then, in fact, only to the software within their Interrupt Service Routines (ISRs). The use of ISRs for the time-critical processes is pretty well dictated by the requirement that diagnostic/visualisation/logging activity should have no effect on the timing or dataflow within the latency-critical processing. We do not have multi-ported memory on each CPU so there must be C44 involvement in diagnostic/visualisation/logging activity. If this activity is not to affect time-critical processing then it follows that it must be interrupted by signals indicating the arrival of new real-time data. In the case of the WFS ring, the signal is the arrival of a new frame of WFS data. In the case of the SG ring, the signal is the arrival of newly digitised strain gauge data from the Pentland ADCs. There is of course the issue of processor interrupt latency but the effects of that are effectively eliminated by the use of DMA transfers for the interrupting WFS and SGS data, combined with the “cowcatcher” system (below).

The diagnostics/visualisation activity effectively forms a background activity and clearly needs to use an interruptible, and therefore asynchronous, communications system to exchange data with external workstation processes. For example a WFS processor sending pixel data to a workstation GUI for display must be interrupted mid-message by the arrival of new pixel data coming from the CCD controllers. The physical medium for this background communication is via the network of interconnecting communications ports. An asynchronous message-passing protocol is therefore required. This is called GP for General Purpose and is described in detail by David Buscher’s document. It is called “General Purpose” because it is also used for transmission of command and status information to/from the C40/C44 processors. Such a protocol is clearly well suited to retrieving status data and can also be used for sending commands to the C44s provided there is some means of synchronising the actual changes in real-time parameter states on different processes. This requirement is fulfilled by the transaction system, which is described below.

GP message packets originating from a C40/C44 processor are forwarded from processor to processor according to a destination address embedded within the packet. Other internal data fields, as described below, further identify their contents. Diagnostic data packets arriving at the diagnostic C40 (GP number 9) are placed in a shared memory buffer, ready for transmission via the VME bus to the VME host.

Other packets typically travel via LIAs to the VME host. Either way, the server process, `c40Comms`, running on the VME host, embeds the GP packets within appropriately addressed TCP/IP packets for onward transmission via the internet. The `c40Comms` process also performs a reverse procedure, extracting GP packets from TCP/IP wrapping and forwarding them via LIAs into the C40/C44 network.

The ISR code cannot use the GP system directly for interprocessor communications. Firstly, this is because GP is not synchronous and would not meet the processing latency requirement. Secondly, in the context of an ISR the GP system will have been interrupted by the ISR code and such a protocol could hardly be made re-entrant without a significant loss of efficiency. A second, synchronous, interprocessor communications protocol, and one which embodies the Barrier Synchronisation idea, is therefore required for use by the ISRs. The unit of this protocol is called a `LoveTrain`[?], primarily for memorability.

LoveTrains use the ring of interconnections of the C44s to broadcast information between them. Each processor in the ring sends its own output information for broadcast (if any) to its downstream neighbour and then copies information from its upstream neighbour to downstream. The processor's output information is therefore copied right around the ring to the processor immediately upstream, which does not copy it further (i.e., information does not return, redundantly, to its origin). The LoveTrain implements the BarrierSynch because communication is global and cannot finish until all processes have started communicating. The expected quantity of data to be sourced, copied and removed by each processor is coordinated near the beginning of each ISR. This is done by an initial, special BarrierSynch which uses the first LoveTrain after the cowcatcher to broadcast the anticipated number and size of LoveTrain contributions from each processor. The final BarrierSynch of each ISR is also special. It contains the StopMessage, which is used to decide if RealTime processing should cease at the current interrupt. The StopMessage also implements the transaction system, whereby real-time parameter changes, which have been scheduled on several processors, all become effective at the next ISR.

The communication port links used by the ISRs to transmit LoveTrains can also be used by the GP system outside of the ISRs. This is achieved by insisting that the two protocols travel in different directions on the bi-directional links. This is possible because each link direction has its own FIFO buffer system.

### *9.3 Summary of GP system*

The General Purpose Message system (GP) allows asynchronous communications between any 2 C40/C44s and between a C40/C44 and an external processor. This is achieved by a simple message passing and forwarding system within the C40 network and by wrapping the GP messages in TCP/IP packets (accessed using the DTM library) for transmission via the internet.

David Buscher in "The GP Messaging Library" deals with the GP system in detail. Here we summarise a few key features.

GP messages consist of fixed-length header and a variable length body. The header fields include a length and source and destination address fields. The body carries most of the actual data and can be up to 3300 32-bit words long. This is the C declaration for the header structure:

```
typedef struct {
  int32 length;          /* Length of the entire packet in words -1 */
  uint32 protocolID;     /* Synchronisation word */
  uint32 destAddress;    /* Destination machine  port address */
  int32 hopCount;        /* Incremented on every hop - used to trap
                            messages which never reach
                            their destination */
  int32 command;         /* Command/acknowledge verb */
  int32 sequenceID;      /* Unique message tag for multiple
                            messages of the same type */
  uint32 replyAddress;   /* Reply_to machine + port address */
  int32 arg1             /* Optional command argument –
                            pads to 8 words */
} GPmsgHdr;
```

The `replyAddress` field is used so that a reply can be sent if appropriate. The 32-bit address fields have the following format:

| | |
|---|---|
| Bits 31:16<br>Port number | Bits 15:0<br>CPU number |

For a C40/C44 address, the port number is always 0, whilst the CPU number is the GP number assigned to the CPU at boot time. The C40s/C44s in NAOMI are numbered 1 to 18 with the c40Comms process on the VME host having the special number 0.

For a workstation process address, the CPU number is always 0. So, as far as a C40/C44 routing such a packet is concerned, the correct thing to do is to forward the packet to the `c40Comms` process on the VME host. This is the correct behaviour. The port number is non-zero in this case and this identifes to `c40Comms` the final destination of the packet on the internet. The c40Comms process maintains a table translating port numbers to DTM ports. DTM ports are identified by strings, which include both an IP address and an IP port number or symbolic name. When a client workstation process opens its communications with `c40Comms` its first action is to request (via a GP command) that `c40Comms` make an association between the client's DTM port for GP replies and a new GP port number. The client sends the DTM port name as an 'argument' to the GP command and receives the generated port number as a reply. It can then use this port number as a reply address in the header of subsequent GP commands directed to c40 CPUs. The GP commands from workstation clients are wrapped in DTM packets addressed to `c40Comms`, which then extracts the GP command and forwards it via an LIA to the C40 network. The reverse process happens to a a reply GP message from the C40s to a workstation client. In this case c40Comms wraps the GP message in a DTM packet addressed to the DTM port corresponding to the GP port number in the destination field of the GP packet. It is also possible for a workstation client to request a mapping from a port number to a DTM port attached to a "third party" workstation process. This is intended for directing diagnostic GP packets to a display or logging process. In practice, however, many processes request their own diagnostic packets.

## 9.4 Command formats

GP command headers contain a `command` field, which, like the destination and reply addresses, is also divided into two sub-fields. In this case the fields are a command class and an actual command indentifier. The class field identifies to a standard C40 program which GP callback function it should invoke in order to further identify and process all commands of that class. An example is found in the file

```
${STAGING}/SharedInclude/c40RealTimeCommands.py
```

This contains the python statements which define the python strings associated with a new class and its commands. The installation process uses these statements to generate a C include files which contains `enum` statements that make the same association.

## 9.5 DTM

The Data Transfer Mechanism (DTM), developed by the NCSA, is the system that is used for message passing. It is layered on the TCP socket libraries and provides a relatively simple message passing API for use between (potentially remote) processes. The DTM API allows the destination port to be named either via an IP address/port (which can be in symbolic or numeric form) or using a *name* which is not known to the TCP/IP dns system. Such names are translated using a *Nameserver* process, which also allows names to be registered (by association with IP address/ports). One Nameserver process can serve several machines. All the machines which share the Nameserver must “know” the IP/port address on which it can be contacted.

DTM communication always begins with a freeform text *header* message followed by a number of binary *data* messages. The header generally serves to describe the format and contents of the data packets. Although the header is freeform in a sense, its contents must normally be parseable in order to achieve automatic data description. Several simple conventions and code for header interpretations are distributed with DTM can some of these are employed with NAOMI. GP messages are very straightforwardly copied into DTM data packets as described above. DTM is also used for communication between workstation applications and display tools and these re-use some of the DTM header conventions. New-style commands are sent over DTM as python embedded into the data packet with “meta data” in the header describing the reply and acknowledge addresses. The *Sequencer* processes use this system. The sequencer and associated *EPM* (Electra Process Monitor) are used to coordinate control of NAOMI systems at the high level (for example using TopGui).

## 9.6 GP main loop

The following C statements form a typical main C40 programme configured to use the GP system.

```
  GPsetup(status);
  GPaddCallback(RT_CLASS, &RTcallback, status);
  GPaddCallback(MIRROR_CLASS, &MirrorCallback, status);
  GPmainLoop(status);
```

The first statement sets up the GP system whilst the next two associate call-back C functions with command classes. Finally the programme enters the `GPmainLoop` where it remains until the CPUs are reset. The `RT_CLASS` contains the special commands which configure ISRs, and the transaction system whereby Parameter Block data may be communicated between ISRs and the GP system. The status structure pointer is used to track errors. By convention each function will return immediately if an error status has been set.

### *9.7 GP callbacks*

Basic C40 programs are normally set up as simple message-driven programs. The actions of the GPmainLoop consist of waiting for a message, decoding the header, performing whatever action the header specifies, and returning a reply or error message. The program then loops back and looks for another message.
The convention is that all messages sent to port `PORT_BOOT` are handled by calling the `BootServices` function. This is essential for the messaging system to function properly.

All other commands are handled in the appropriate callback. The callback contains a switch statement which decodes the command verb and performs the appropriate action. This processing frequently involves extracting additional argument data from the message buffer. By convention, all commands put the reply in the same message buffer used to receive the message, and indicate the length of the reply by adjusting the header appropriately. At the end of the switch statement, control is returned by the callback to GPmainloop and any reply in the message buffer is sent back to the workstation, with the `header.arg1` value set to zero to indicate success, and the `header.command` value incremented by `GP_ACK` to indicate which command is being acknowledged.

If an error is encountered, using the `ExcRaise` macro will store an error message in the status structure (defined in `exception.h`). At the end of the switch statement, if an exception has been raised, a reply with a non-zero `header.arg1` value is sent to indicate an error. The reply contains the error message in its body, in case the receiving program wants to print it out. In addition, the error message is sent to the `C40_STDERR` port. If an error message printing process is listening on that port, it will print it for the user. (A `fprintf(stderr,..)` statement on the C40s will also cause a message to be printed by any such error-logging processes.)

### *9.8 ISRs and cowcatchers*

Interrupt Service Routines (ISRs) perform all latency critical data processing and are the only context in which a Barrier Synchronisation can take place. They are therefore the only context in which the components of a LoveTrain can be transmitted, copied and received. ISRs are normally invoked by an external interrupt signal. In the case of the WFS ring this will be ultimately caused by the arrival of a new frame of WFS data. In the case of the SG ring the ultimate cause is a timer interrupt. However, even though these are the ultimate causes of ring-wide interrupts, most of the CPUs in the ring are actually interrupted by communications port traffic from the neighbour (upstream) CPU in the ring. This is because only the timer CPU is actually interrupted in the SG ring case and only the WFS frame reception CPU or CPUs are interrupted in the WFS ring case. The interrupted CPUs then begin to send data to their neighbours which then in turn begin to send data to *their* neighbours and so on. The CPUs are armed to interrupt on receipt of communications port activity so this has the desired effect of eventually putting all of the ring CPUs into their ISRs. However if we relied upon waiting for the first processed data output from the interrupted CPUs to accomplish this, we would have the unfortunate side effect that we would then have an additional wait whilst the interrupted CPU actually arrived in its ISR code. This delay, the interrupt latency, is in-part a hardware delay caused by the processor saving the interrupted code context on the stack, and partly a software delay caused by the execution of the ISRwrapper assembly language code which prepares for execution of a C function and ensures that certain global data may be accessed from the ISR. Incurring these delays in a system-critical fashion as each CPU in the ring were sequentially interrupted would be most undesirable.

The deleterious effects of most of the interrupt latency are overcome by arranging for a high degree of concurrency. Each interrupted CPU, whatever the cause of the interrupt, immediately sends to its downstream neighbour a single word on their connecting communications port. This has the effect of interrupting the neighbour promptly so that it will almost certainly have completed its interrupt latency before useful data could be sent to it. Because these interrupting words precede the first LoveTrains carrying operational data they are dubbed "cowcatchers" for memorability. The interrupt latency of the WFS ring is concurrent with the arrival and decoding of the WFS header and the processing of the first row of centroid data. The interrupt latency of the SG ring is concurrent with the conversion time of the SG ADCs. The ADC conversion is initiated by the SG timer CPU immediately on its (timer) interrupt.

### *9.9 ISR debug logging*

The `LOG(val)` macro records a programme file, programme line number, frame number, time, and an integer argument `val` in a circulating buffer. This may be retrieved at any time and is useful for tracing code execution and for obtaining time-synchronised data samples simply. The command `RT PrintDebugLog nCPU` will print logging data from CPU (GP) number `nCPU`. Care must be taken when examining these logs as the circular buffer will typically contain output from several ISR interrupt frames and will have recycled and overlapped to a seemingly arbitrary point at the time of download and printing. A careful examination of the frame numbers and times for the lines in the log output will reveal which is the most recent log record.

### *9.10 Panics*

When an error is detected within an ISR the standard action is to invoke the C macro `Panic()`. This causes the following actions:

1. The C program jumps (!) to an exception processing label at the end of the ISR code. No further Real-Time processing therefore takes place.
2. An exception is raised which records the following information in a status message: the time, frame number, super-step, program line, and program file at the which the panic occurs.
3. Future ISRs are disabled for this CPU.

Because of this last action then no other CPUs in the same ring can pass further BarrierSynchs, because no LoveTrain can pass the Panic'ed CPU. Ring-wide real-time activity therefore halts at this point. Note, however, that the panicking CPU has not crashed. It can still perform GP processing in its foreground task, including displaying its Panic status message, normally in response to the `WFS Status` and `SG Status` commands for the respective C40 rings.

## 9.11 Barrier Synchronisation

Barrier Synchronisation is both the method by which the supersteps of ISRs within a CPU ring are synchronised and also the method by which they communicate. Each CPU which participates in a Barrier Synchronisation sources a certain amount of data to its downstream neighbour, copies a certain amount of data from its upstream neighbour to its downstream neighbour, and sinks a certain amount of data that has already been all the way around the ring and would, if it were copied further, be returning (inefficiently) to its originator. This ring-wide movement of data, collectively constitutes a LoveTrain.

For efficient operation, careful checking that the expected quantity of data arrives and does not cause a timeout or overrun is *not* carried out for every LoveTrain. Instead the each CPU publishes a *plan* of its subsequent LoveTrain activity at the start of the ISR. The exchange of plans *is* carried out with careful checking. Each CPU then compares all received plans with its own intentions and if they are inconsistent, panics.

A typical plan is defined by the following C statement:

```
static BarrierSyncPlan myPlanWFS[] = {
    {  5, NUM_SUBAP_X*2*NUM_CCD_PORT, NUM_SUBAP_X*2*NUM_CCD_PORT, 0 },
    {  1,            3,            3*NUM_RING_CPU, 0 },
    { -1, -1, -1, -1 }
  };
```

Each line of the plan specifies a set of similar BarrierSyncs that the CPU intends to perform in turn. The first field specifies the number of similar BarrierSyncs and the next field specifies the number of 32-bit words which the CPU plans to *source* (i.e., to add to the LoveTrain) at each of these BarrierSyncs. The second field contains the total number of 32-bit words it anticipates there to be in each LoveTrain. The final field is always initially zero but is set during a successful exchange of plans. The final line of the plan simply signals that this is the end of the plan, but the penultimate line is more interesting. This always specifies the exchange of a *StopMessage*, which is always the last LoveTrain to be exchanged in each ISR. It establishes if real time activity is scheduled to stop on completion of the current frame, or if a change of real-time parameters or algorithm has been scheduled ring-wide at the next interrupt. This last function implements a key part of the *Transaction* system (see below).

The plan is exchanged using the C function

```
    BSPbegin(myPlanWFS, ISRglobals.status);
```

where the plan array forms the first argument. The BarrierSyncs themselves are carried out by the following function:

```
     BarrierSync(iSuperStep,(int *)centroid,
                    NUM_SUBAP_X * 2 * NUM_CCD_PORT,
                    0);
```

The first argument is the superstep counter (which is incremented by the invoking program after the BarrierSync), the second is a pointer to the LoveTrain buffer, the third is the number of words to source and the fourth is the number of words to copy.

The StopMessage is exchanged and acted on by the following code fragment:

```
  /* Transfer stop frame info around the ring - barrier synchronisation */
  if (StopMessage(iSuperStep))Panic();
  iSuperStep++;

  /* Do algorithm shuffle */
  if (SwapAlgorithm())Panic();
  LOG(0);
```

Note that some older ISR code uses a direct call to `BarrierSync` to deal with the StopMessage.

## *9.12 Parameter Block transactions*

Parameter block transactions implement the synchronised changing of real-time parameters and even algorithms ring-wide. Essentially a set of changes can be queued up in advance on each CPU in a ring and then made active simultaneously on a particular interrupt. The queuing of changes is accomplished by the GP commands of the RT_CLASS and can be activated by either C or Python workstation programs but is most elegantly implemented in python, where a single python function call can carry out a very complex ring-wide transaction.

In order to be available to the standard function `RTCallback,` which processes the `RT_CLASS` commands, each C40 program must include a code fragment along the following lines in its `main()` function:

```
/* Table to hold the set of available algorithms. Used in RTcallback()
 * to define the real-time algorithms available in this executable.
 */
extern struct AlgorithmMethods SGstarterMethods;
extern struct AlgorithmMethods SGmirrorMethods;
extern struct AlgorithmMethods SGadcMethods;
extern struct AlgorithmMethods SGtimerMethods;

const struct AlgorithmMethods *algorithmMethods[] = {
  &SGstarterMethods,
  &SGmirrorMethods,
  &SGadcMethods,
  &SGtimerMethods,
  NULL /* Required to mark the end of the table */
};
```

The standard array of pointers to `AlgorithmMethods` structures, `algorithmMethods`, establishes a global record of available algorithms and their associated parameter manipulation functions. This is available to `RTcallback` which is then able to invoke particular methods (functions) according to `RT_CLASS` command parameters. The methods themselves are defined externally to the main program and are in fact, most conveniently, defined in the same program files as the

ISR algorithms themselves. Consider the following example from an algorithm C file (the names of these files conventionally begin with Alg):

```
static void ISR(void);  /* The interrupt service routine */
static Algorithm *Create(const Algorithm *, ExcStatus *);
                      /* Create/copy an algorithm instance */
static void Destroy(Algorithm *, ExcStatus *);
                      /* Release resources used by an instance */
static void SetParameters(Algorithm *, int32, int32 *, int32, ExcStatus *);
                      /* Set/alter the parameters of an instance */
static void GetParameters(const Algorithm *, int32, GPmsgBuffer *,
                            ExcStatus *);
                      /* Return a instance parameter set in a binary format
*/
static void PrintParameters(const Algorithm *, int32, ExcStatus *);
                      /* Print parameter set values to stderr */

struct AlgorithmMethods SGadcMethods = {
  ALG_SG_ADC,
  "$Id: AlgSGadc.c,v 1.21 2000/05/24 10:11:59 rmm Exp $",
  &ISR,
  &Create, &Destroy, &SetParameters,
  &GetParameters, &PrintParameters
};
```

Note that the AlgorithmMethods structure definition includes the following:

1. an algorithm ID. This is made available to both C40 and workstation programs (C and python) using the SharedInclude system. It is used by workstation programs to identify which algorithm is to be swapped in or to have its parameters manipulated or interrogated.
2. The second is an RCS string which can be used to identify which algorithm versions are currently running. This is typically accomplished using the workstation command:
   RT GetAlgorithmVersion
3. The remaining fields are pointers to functions. The first is a pointer to the ISR function itself, which is defined in the same file.
4. The Create function (one per algorithm file) is used to set up a new algorithm, Destroy removes its resources.
5. SetParameters is a function made available to RT_CLASS to manipulate the parameters of an algorithm whilst GetParameters is used to retrieve them. PrintParameters is a future extension which may be used to 'print' the parameters to an error-logging process. SetParameters does not directly manipulate the parameters of the currently active algorithm but rather manipulates a set of duplicated parameters waiting to be swapped into active use by the Transaction system.

The functioning of the algorithm methods themselves is dealt with in the next section below. The principal commands of the RT_CLASS are summarised below:

RT_INIT – initialises the real-time transaction system
RT_STATUS – enquires about the status of the ISRs and transaction system,
RT_START – starts interrupt processing
RT_STOP – stops interrupt processing
RT_BEGIN_TRANSACTION – obtains a **transactionID** from the *ringleader*
RT_END_TRANSACTION – instructs the *ringleader* to release a transaction

`RT_BREAK_TRANSACTION` – aborts and unlocks a transaction setup
`RT_SET_PARAMS` – invokes `SetParameters` for a named algorithm ID
`RT_GET_PARAMS` - invokes `GetParameters` for a named algorithm ID
`RT_PRINT_PARAMS` - invokes `PrintParameters` for a named algorithm ID
`RT_GET_ALGORITHM_VERSION` – Gets the Algorithm RCS ID

Workstation programs may invoke these commands either directly with the GP `rpc` (remote procedure call) function which is available in both the C and Python versions of the workstation GP support libraries, or in the case of python workstation programs, they will probably chose to invoke them via the `GPtransaction` library.

The actual mechanism of a transaction is as follows:

1. The workstation requests a *transactionID* from the ringleader CPU of a particular ring. There is only one ringleader per C40 ring and the each CPU 'knows' via a global variable whether or not it has been assigned as a ringleader. If there is already a transaction in progress on a ring, that is a transactionID has previously been allocated and never released, then the ringleader will refuse to allocate another. In this case the workstation programme or user may choose to break the transaction. In the case of the WFS ring the workstation user command `WFS BreakTransaction` will accomplish this.
2. Assuming a transactionID was successfully obtained, the workstation process can then begin to setup the parameters of its chosen algorithm on an CPU. To do this it invokes `RT_SET_PARAMS`, specifying an algorithmID and a transactionID. It identifies exactly which parameter that is to be manipulated using a *ParameterBlock Section* ID. These Ids are available to both workstation (C and python) and C40 programmes through the `SharedInclude` system.
3. Following the manipulation of ParameterBlocks on all CPUs where changes are required, the workstation process send the `RT_END_TRANSACTION` command to the ringleader. This unlocks the transaction and instructs the ringleader to send the transactionID as part of its next ISR StopMessage LoveTrain. This instructs all the CPUs to swap in the modified parameter blocks (and potentially algorithms) at the next interrupt.

The ParameterBlock section Ids identify a particular parameter for replacement or retrieval. Many are actually arrays of variables. The section IDs are defined in `Electra/SharedInclude/ParameterBlocks.py` and are summarised below. A `PB_SG_` prefix indicates an SG ring CPU parameter block section. All others are WFS ring CPU parameter block sections.

| | |
|---|---|
| `PB_FLAT` | unused |
| `PB_WFS_OFFSET` | RAL WFS offset |
| `PB_SEG_GAIN` | WFS segment TT gains (X,Y) |
| `PB_DECIMATE` | WFS centroid and pixel diagnostic decimation values |
| `PB_RECON_GAIN` | usused |
| `PB_LOCK_DAC` | open/close WFS loop |
| `PB_CCD_TEST` | WFS test mode |
| `PB_FRAME_DELAY` | WFS test mode rate |
| `PB_SG_TIMER_INTERVAL` | SG sample interval (in 66ns clocks) |

| | |
|---|---|
| PB_SG_DEMAND_PORT | identify demand communications port for SG ring |
| PB_SG_TIMER_TRIGGER_PORT | identify timer trigger communications port |
| PB_SG_MIRROR_DATA_PORT | mirror data output port |
| PB_SG_MIRROR_SYNC_PORT | mirror sync output strobe port |
| PB_SG_ADC_PORT | SG ADC data input port |
| PB_SG_ADC_BLOCK_SIZE | size of SG ADC data block for this SG CPU |
| PB_SG_ADC_CAL_GAIN | SG ADC calibration gain vector |
| PB_SG_ADC_CAL_OFFSET | SG ADC offset gain vector |
| PB_SG_ADC_SERVO_GAIN | SG servo loop gain |
| PB_SG_PASS_THROUGH | mirror actuator flags controlling SGloop |
| PB_SG_ADC_REORDER_TABLE | vector indicating SG ADC channel order |
| PB_SG_INITIAL_DEMAND | mirror demand received from WFS ring |
| PB_SG_DAC_REORDER_TABLE | table for reordering LoveTrain actuator values |
| PB_SG_WAVEFORM | test actuator waveform |
| PB_SG_CAPTURE | SG rapid sample diagnostic mode control |
| PB_SG_SNAPSHOT | SG synchronised sample diagnostic mode control |
| PB_SG_ACCUM_ZERO_HOLD | SG open/closed loop (normally use PASS_THROUGH) |
| PB_SG_DIAGNOSTIC | |
| PB_CENTROID_WEIGHT | X and Y pixel weights for centroiding |
| PB_SG_TABLE_OFFSET | identify block of SG ADC data |
| PB_MATRIX | Reconstructor matrix |
| PB_MATRIX1 | Reconstructor section |
| PB_MATRIX2 | Reconstructor section |
| PB_MATRIX3 | Reconstructor section |
| PB_MATRIX4 | Reconstructor section |
| PB_TT_FLAT | Zero level for global tip-tilt |
| PB_TT_GAIN | Gain matrix for global tip-tilt |
| PB_SG_ADC_SNAPSHOT_DECIMATE | frame-wise decimation for SG ADC snapshot (all channels) diagnostics |
| PB_SG_ADC_CAPTURE_DECIMATE | buffer-wise decimation for SG waveform (single continuously-sampled channel) diagnostics |
| PB_SG_DAC_SNAPSHOT_DECIMATE | frame-wise decimation for SG DAC snapshot (all channels) diagnstics |
| PB_XYZ_TO_ABC | Geometry matrix for segment XYZ algorithm |
| PB_I_AM_A_DUMMY | Set into dummy interrupt mode |
| PB_PIXEL_TIMEOUT | Timeout for waiting for pixels (in clock ticks) |
| PB_FLUX_MEMORY | Decay constant for flux low-pass filter - zero is no memory, unity is no learning |
| PB_SG_SYNCHRONISE | set synchronisation to WFS ring demands |
| PB_SG_FEEDFORWARD | send new demand deltas direct to the mirror |
| PB_SEGMENT_TILT_LIMIT | Limit of tilt during closed-loop operation |
| PB_BACKGROUND_WEIGHT | Pixel weights for background estimation |
| PB_QUICK_PISTONS | Compute pistons immediately after the tilts |
| PB_CENTROID_BIAS | Bias for x, y and flux sums |
| PB_LONG_WFS_OFFSET | Full-frame WFS offsets for use with NAOMI |
| PB_LONG_CENTROID_BIAS | Full-frame WFS biases for use with NAOMI |
| PB_SDSU_CURRENT_CAM | Sets current UNSYNCHED loop controlling camera - for use with NAOMI |
| PB_ACCEPT_FRAMES | Sets WFS CPU to accept SDSU frames |
| PB_ROUTE_CENTROIDS | Sets WFS CPU to route SDSU centroids to mirror |
| PB_ROUTE_PIXEL_DIAGS | Sets WFS CPU to route SDSU pixel diagnostics |
| PB_ROUTE_CENTROID_DIAGS | Sets WFS CPU to route SDSU (direct) centroid diagnostics |
| PB_LONG_CENTROID_BIAS_4x4 | 4x4 WFS bias for use with NAOMI |
| PB_LONG_CENTROID_BIAS_2x2 | 2x2 WFS bias for use with NAOMI |
| PB_CENTROID_WEIGHT_4x4 | 4x4 X and Y and flux pixel weights for centroiding |
| PB_BACKGROUND_WEIGHT_4x4 | 4x4 Pixel weights for background estimation |
| PB_CENTROID_WEIGHT_2x2 | 2x2 X and Y and flux pixel weights for centroiding |

| | |
|---|---|
| PB_BACKGROUND_WEIGHT_2x2 | 2x2 Pixel weights for background estimation |
| PB_SDSU_STATUS | WFS interface status (Get only) |
| PB_HEADER_TIMEOUT | Timeout for waiting for SDSU header (in clock ticks) |
| PB_BACKGROUND_FLUX_MEMORY | Flux memory specific to background calc |
| PB_SURROGATE_WFS_APP | Set a WFS mode to simulate from full frame data |
| PB_TT_ONLY_GAIN | Gain for use in tip-tilt only (app 10/doublet) mode |
| PB_TT_ONLY_LIMIT | Limit for use in tip-tilt only (app 10/doublet) mode |
| PB_SELECT_APP10 | Assume SDSU App 10 is in use rather than App 8 |

## *9.13 Algorithm methods*

The `Create` function of each algorithm performs certain activities by convention when it is invoked. It allocates space for a parameter block structure which will contain all the parameter data (and some of the operational data) that the algorithm will use. Where rapid write-access to data is required for particular parameters or operational buffers, it will only allocate space for a pointer in the parameters structure itself and will instead allocate the actual memory in on-chip RAM. This is a limited resource in the C40 architecture but provides faster access than off-chip RAM. The `Create` function can ascertain whether or not it is the first invocation for this type of algorithm. If it is the first then it will initialise the parameter block with default values. If it is not the first invocation then it will copy the values from the last invocation into the newly allocated structure. In this way, the `SetParameters` function need modify only the selected parameters of an existing configuration and does not have to upload a full set of data each time a change is required.

The `SetParameters` function identifies which parameter block section is actually being manipulated using a switch statement. Normally a number of additional arguments are then decoded from the GP body. The precise number and nature of these arguments depends on the Parameter Block section which is being manipulated. Floating point arguments may be unpacked in-situ from the GP message body following this example code fragment:

```
floatArg = (float *) arg;
afrieee(floatArg, SG_UNPACKED_DEMAND_BUF_SIZE);
```

The `GetParameters` function also uses switch statement to identify which ParameterBlock Section is being addressed. It then formats a buffer body to deliver the requested information. Again the number and nature of the elements in the body depends on the ParameterBlock section in question.
Floating point arguments may be packed for GP transmission as follows:

```
floatArg = (float *) buffer->body;
for (i = 0; i < SG_UNPACKED_DEMAND_BUF_SIZE; i++) {
  floatArg[i] = parameters->ADCcalGain[i];
}
atoieee(floatArg, SG_UNPACKED_DEMAND_BUF_SIZE);
```

## *9.14 Misc support libraries*

There are a number of other support libraries and macro collections which support the writing of C40 programmes. Import examples are the `Exc` exception handling system and the `AIO` asynchronous I/O system which manipulates the on-chip DMA engines.

### *9.15 WFS Programme structure*

Two algorithms are involved in WFS ring processing: `AlgNaomiInterleave`, which deals with SDSU WFS frame reception, and runs on all but one of the WFS ring CPUs and `AlgMartini`, which takes the centroid data produced by `AlgNaomiInterleave` and derives drive signals for segmented mirror and fast steering mirror.

### 9.15.1 AlgNaomiInterleave

`AlgNaomiInterleave.c` contains the ISR itself, its associated methods such as `Create`, `SetParameters` and `GetParameters`, and numerous helper functions invoked from the ISR. Some of these helper functions are tagged for inline compilation for speed of execution. The file begins with the including of the various required include files for utility c40 and real-time support libraries (in `Electra/RealTime/libsrc`), definitions of working constants, and the declaration of the parameter block structure and `AlgorithmMethods`.

The ISR function of `AlgNaomiInterleave` has several possible modes of operation. The most fundamental condition determining which mode to execute is whether or not the processor is attached to a wavefront sensor readout, i.e., whether it is a “WFS CPU”. If not, then its role in the current implementation is simply to pass on data: it is a “slacker” representing spare processing capacity. (The distributed reconstructor under development as `AlgParallelSISO` is designed to exploit this capacity.) The processor determines whether it is attached to a WFS output port using the `WFS.in` structure which is set up before the RT system is started using ordinary GP commands rather than transactions. The reason for this is simply that the WFS interrupts are driven by the reception of WFS frames and the configuration of the WFS system must precede the use of the transaction system, which depends on interrupts.

The attachment of a WFS channel to a CPU does not necessarily mean that it will always be desirable for that CPU to process frames. Consider, for example, the use of only one of the two NAOMI WFS CCDS when both are reading out in synched mode. This is important in system alignment, when it can be desirable to switch rapidly between both cameras. In this case it is necessary to switch off frame processing on one of the CPUs even though a WFS is attached and data frames are arriving. This switching is achieved using the parameter `acceptFrames`.

When a WFS CPU is receiving and processing frames there are a number of options available for routing output data and diagnostics. Using the `routeCentroids` parameter it is possible to control whether output centroid data are passed from this CPU to the mirror control CPU (running `AlgMartini`) or not. Similarly `routePixelDiags` and `routCentroidDiags` control whether or not the CPU send diagnostics packets to the diagnostic CPU containing pixel and centroid data respectively. Note that it is not usual to send centroid diagnostic data from a WFS CPU. Rather it is the mirror CPU in the WFS ring which is normally responsible for this. The routing of centroid Data to the mirror CPU and the routing of centroid diagnostics are therefore coupled in normal operation. The facility to decouple them using the data routing parameters enables a configuration where one WFS CCD is responsible fore closing the DM control loop whilst the other is providing pixel and centroid diagnostics for alignment. The purpose of this configuration is to allow the WFS CCDs to be mutually aligned with the DM control loop closed.

There are two other special configurations available in `AlgNaomiInterleave`. Firstly *dummy mode* allows a CPU to generate interrupts without WFS frames actually arriving (or even the WFS being switched on or attached). This is accomplished using a c40 timer to trigger interrupts instead of a WFS communications port. Dummy mode is controlled by the parameter `IamAdummy`. The second configuration is *surrogation* whereby the WFS actually provides frames in NAOMI SDSU mode ('application') 1 but the CPU reformats the data to simulate some other mode and then processes the data as if it were actually provided in that format. The purpose of this configuration is to debug the processing of data for various WFS modes.

With one exception the processing of the data from the various modes ('applications') of the SDSU WFS is *dynamically determined*, that is to say that the WFS mode of each individual WFS data frame is determined from self-describing header data and the processing algorithm is adjusted accordingly. The exception is mode 10 which shares the same application-identifying bit pattern as mode 8 and so must currently be enabled by the previous transmission of a parameter `selectApp10`. A future development might be to determine this distinction using auxiliary header data (row/column) instead.

The following sequence describes the detail of the operation of AlgNaomiInterleave essentially in its conventional processing mode, whilst passing comments about the other configurations. The detailed description of the helper functions is deferred to later discussion.

1. the definition of local static and automatic variables
2. cache configuration (generic):
   ```
   CACHE_ON();
   CACHE_DEFROST();
   ```
3. Setup timer and frame numbering (generic):
   ```
   /* Synchronise watches on first frame */
   if (ISRglobals.iFrame == 0) RunWallClock();

   /* New frame number */
   parameters->wfsFrame = ISRglobals.iFrame++;
   LOG(0);
   if (!ExcOK(ISRglobals.status)) Panic();

   /* Record start of frame time */
   start = C40ticksStart();

   /* keep track of inter-frame interval */
   parameters->frameInterval = start - lastStart;
   lastStart = start;
   ```
4. Setup heap variables (generic)
   ```
   /* Set up heap variables */
   parameters = ISRglobals.currentAlgorithm->parameters;
   centroid = parameters->centroid;
   ```

5. Start the DMA for the SDSU WFS header data. First allocate a pointer to the DMA engine control registers:
   ```
   wfsData = WFS.in.AIO;
   ```
   On the basis of receive_frames parameter start the actual header read:
   ```
   AIOread(wfsData, parameters->naomiHeader, NAOMI_HEADER_SIZE - 1);
   ```

6. Now that the time-critical DMA processing has been initiated, complete the generic ISR startup:
   ```
     /* Unblock comports so that messages and wakeups can occur */
     GPunblock(ISRglobals.status);

     /* Start timer for timeouts etc */
     RunTimer(ISRtimer, 100);

     /* Remove wakeup (cowcatcher) word from comport */
     if (CowCatcherRemove()) Panic();
   ```
7. Agree the BarrierSync plan
8. If required, start the WFS frame processing. Begin by initialising the frame processing variables if we are on the first iteration then examine the first Header word to see is if it is a start-of-frame word (`0x8000`). It need not be as it could have been absorbed at the end of the previous frame depending on SDSU timing.
9. Check that the header DMA is complete. If not, check that we have not arrived at the end of the timeout interval. Once we have the whole header begin decoding it. We first extract the `naomiOpMode` and `naomiCameraID` header fields. We then switch to format specific code based on the application subfield (this is known as the SDSU *application* or *mode*, eg., mode/application 1 is full-frame readout):
   ```
   switch(naomiOpMode & NAOMI_APPLICATION_MASK) {
   ```
10. Each case statement of the switch statement deals with a particular frame readout format and sets the `naomiApplication` variable accordingly. Bases on this another larger DMA of the actual pixel data is initiated:
    ```
    AIOread(wfsData, parameters->pixel, numPixelsDMA);
    ```
11. At this point we decide if we are doing WFS *surrogation*, a debugging mode (not used much, if at all) where a full frame of real CCD data is actually read, then reformatted by the receiving C40 to look like one of the other windowed (and perhaps binned) modes. Let us assume we are not using this mode.
12. Based on the `naomiApplication` variable we enter another switch case statement where a number of pixel processing control variables are set up. These determine how many pixels there are per WFS subaperture box, what its geometry is, how many pixels there are per quadrant line, how many guard pixels there are between boxes and how many subapertures there are per quadrant. Everything is processed on a quadrant basis even though SDSU data is delivered in an interleaved format (you get one pixel from each quadrant in turn). We also set up pointers to the pixel weighting matrices used in deriving the centroids and background.
    Note that there is special background processing used for applications 8, 9, and 10. The first two are 4x4 subaperture modes, the latter is a tip-tilt only mode with only a single aperture. These formats have special background monitoring pixels in the corner of each quadrant and these require special processing. All other modes call the helper function `DeriveNaomiBackground` for each CCD quadrant. This function takes

as arguments the pixel geometry and background weight variables as well as a quadrant ID and, of course, the pixel data buffer pointer, and an output variable such as `&background1`.

13. We next begin the processing of the WFS centroids. This is done for each row of subapertures in each quadrant. We therefore enter a `for` loop which indexes each row of subapertures in turn:
    ```
    for (iSubApY = 0; iSubApY < numSubapY; iSubApY++)
    ```
    Remember that the rows for each quadrant count from the outside of the chip inwards as this is the quadrant read order. Note the dependency on the `skipTable` as to whether a particular row of centroids is processed or not. This table, together with the `copyTable`, is used for processing SDSU modes (‘applications’) 8 and 9. These modes operate with larger than normal WFS lenslets which cover ~4 segments. The required grouping together of segments is achieved by copying the centroid values to the adjacent segments within each group. This copying is controlled by the two tables, the indices of which can either centroid row or column numbers within each quadrant (because of the row-column symmetry in each quadrant). A non-zero entry in the `skipTable` means do not do any processing for this row/column of centroids at all and leave the pixel buffer unchanged. Apart from the case of the first row/column, which corresponds to disabled segments in NAOMI, such processing will not be required because data will already have been copied into the current buffers by an entry in the `copyTable` for the *previous* row/column.
    The processing of each row consists of the following operations:
    a) wait for the DMA of the row to complete. Note that this DMA transfer actually contains a row of subapertures **from each quadrant** as the quadrant pixels are read out in an interleaved format.
    b) call the helper function `DeriveNaomiCentroidRow` four times to process a row of pixels from each quadrant. Many of the arguments to this helper correspond to those used with the background calculating functions. There are more in this case, however, because, we need to pass in the offset and bias control vectors and the centroid XY weight tables. We also pass in the `skipTable` and `CopyTable` so they can be applied to the individual centroids within a row.
    c) perform centroid row copy operations as required by the `copyTable`.
    d) perform a BarrierSync to transmit the newly calculated centroids to the other CPUs of the WFS ring. Each of the 5 LoveTrains will contain a row of X and Y centroid coordinates (WFS spot locations) for all four quadrants.
    e) increment the current `pixel` processing pointer unless we skipped the current row of centroids because of an entry in `skipTable`.
14. A DMA is scheduled to absorb any trailing pixels that may be in the hardware input buffer following a bad/incomplete WFS data frame. This should allow the reading of the next frame to be properly resynchronised.
15. If the CPU is not doing WFS processing then a series of ‘slacker’ BarrierSyncs are performed after a short delay.
16. The `StopMessage` is transmitted around the ring. This marks the end of the real-time communications phases of the ISR. The remaining processing within the ISR is concerned with the interpretation of the `StopMessage`, re-

initialisation of subsequent ISR processing, and the scheduling of the transmission of diagnostics.

17. A particular CPU is designated as the WFS control CPU via the previously-configured `WFS.out` structure. This CPU requests the next frame of data form all CCD interfaces using the following output statement: `WFS.out.comport->outData = 0x02;`
18. If diagnostic pixel data frames are to be sent then we must determine the type. This can be either the older ELECTRA-compatible full-frame data, which was sent on a per-quadrant basis and actually re-assembled into frames in the diagnostic CPU, or the newer NAOMI type which includes a self-describing header. In the case of surrogation we can alternate between raw full-frame and surrogated data.
19. If we are sending the old pixel data format then we extract the quadrant data and rotate it if it originates from the 'master' CCD (the 'master' and 'slave' CCDs are rotated with respect to each other to allow on-chip row and column binning). We achieve this using the helper function `ExtractNaomiQuadrant` for the 'slave' CCD data and `ExtractNaomiQuadrantTrans` for the 'master'. Old-style pixel data is transmitted with individual quadrants being sent on each of a series of subsequent ISR invocations. There is therefore a minimum sensible *decimation* value for pixel data diagnostics. Diagnostic decimation values within ISRs refer to the number of real data values which must be processed for each corresponding diagnostic value transmitted to the diagnostic CPU. This decimation value is set up as a parameter.
20. If the new type of pixel diagnostic data are required then the helper function `SendNaomiPixelData` is invoked, or, in the case of mode 10 (tipl-tilt only correction) `SendNaomiTTpixelData`.
21. Centroid data may then be optionally transmitted. As described in the introduction to this section, this is rather an unusual situation. Generally the centroid diagnostics come from the mirror CPU running AlgMartini.
22. In the case of dummy mode operation the interrupting timer is re-initialiased.
23. If we are receiving frames then the DMA to remove extra pixels is stopped. Note the rather unusual fact that this will include the Start of Frame Word for the next transmission, which is sent rather early.
24. If we are not a receiving (i.e. pixel processing) CPU but are nevertheless attached to a WFS we complete our flushing operation by waiting to see that the flow if pixels seems to have completed.
25. If no exceptions were raised we return from the ISR, otherwise we raise an exception and disable future interrupts (i.e., we 'panic').

#### 9.15.1.1 AlgMartini

The Algorithm `AlgMartini` runs on only one CPU and that is in the WFS ring. This is known as the `mirrorCPU` although in fact it interfaces to the SG ring and not directly to the mirror. Its ISR is responsible for receiving centroid LoveTrains from the WFS CPUs. It maintains servo loops controlling the mirror segment tip-tilts and conducts a reconstruction of the mirror pistons using a matrix. The XYZ (tip-tilt-piston) segment commands are converted to ABC (triaxial actuator) format and then added to a 'flat' buffer containing the base actuator commands and transmitted to the

SG ring. Various helper functions are invoked to accomplish these procedures, and the description of these is postponed to later.

`AlgMartini` is contained in the file `AlgMartini.c`, which also contains declarations, the ISR itself, the helper functions, and the other algorithm methods. This is the same situation as for other algorithms.

The following sequence summarises the operation of the ISR of `AlgMartini`:

1. After the usual initialisations, which follow the same form as `AlgNaomiInterleave` stages 1-7 we derive the piston and global tip-tilt values from the centroids processed in the previous frame. The logic is that these computations can be efficiently interleaved with the WFS pixel readout latency for the current frame and are therefore conducted before the first centroid `BarrierSync` for the current frame. The computed pistons for the previous frame are then transmitted along with the piston values for the current frame. This may sometimes involve an undesirable delay, however, and the parameter `quickPistons` can be use to enable immediate processing of piston values.
2. We next perform the 5 barrier syncs in order to get all the centroid data from the WFS CPUs We note that there is currently no interleaved processing in this phase and this represents a possible avenue for future expansion along with the implementation of the distributed `AlgParallelSISO`.
3. We next examine the received centroid buffer to see if the WFS processors have flagged the frame as 'bad' and if the frame originated from the 'master' or 'slave' CCDs. The two cameras readout in different formats and this is accounted for by switching a pointer between two different tables according to a `cenID` enumerator embedded in the centroid message. `cenID` can also indicate a tip-tilt only SDSU 'application' 10 frame.
4. On the first iteration we initialise various servo variables and set the output mirror DAC buffer to the current flat value.
5. If the `cenID` indicate a tip-tilt only frame then special processing is carried out for this mode only. This includes setting an overall tip-tilt limit.
6. In full AO mode the `segment` tip-tilt servo variables are updated using new centroid values and the `gain` parameter table. The global tip-tilt variable is added to the tip-tilt slots of the mirror output DAC buffer.
7. The mirror output DAC buffer is translated from its XYZ format using the helper function `XYZtoMirrorBuffer` and transmitted to the SG ring using the helper function `WriteDACs`. Note the use of the parameter `lockDAC` that is passed to `WriteDACs` to determine if the loops are actually to be closed.
8. If the `quickPistons` parameter is set then the new piston values are computed immediately and the full DAC data **re-transmitted** to the SG ring.
9. Following the `StopMessage`, centroid and DAC diagnostics are transmitted to the diagnostic CPU with a rate determined by a single common `decimate` parameter.
10. As with other algorithms the ISR returns if no exception has been raised or otherwise performs panic processing.

# 10. SG Programme Structure

The Strain Gauge (SG) ring runs a single compiled c40 programme on all 8 CPUs within the ring: `SGBSP2.x40`. This programme therefore contains all the algorithms required for operation on the various CPUs: `AlgSGstarter`, `AlgSGtimer`, `AlgSGmirror` and `AlgSGadc`. The source code for each algorithm is contained in a `.c` file of the same name and the same organisational convention as `AlgNaomiInterleave` on the WFS ring is followed: each file contains a parameter block definition, an ISR, and the code for its associated methods. Linking all the algorithms into a single monolith is inefficient in terms of space because each CPU runs only one algorithm. A possible future extension is to produce separate programs for each CPU function as in the case of the WFS ring. Note however that `AlgSGstarter` is required in the initialisation phase and also that a further future rebalancing of the SG processing load may well involve increasing commonality of the algorithms running on different CPUs.

The basic function of the SG ring is to accept the demand output from the WFS ring and compare it repeatedly to sample data from the Strain Gauge ADCs. On the basis of this comparison the final output demand to the mirror control electronics is adjusted using a servo algorithm such that the demand and ADC values become identical. For this purpose the ADC values are subjected to a calibrated transformation to nominal mirror DAC units. Mirror DAC values are represented by 13-bit unsigned integers and the raw Strain Gauge ADC values are 16-bit unsigned integers.

#### 10.1.1.1 AlgSGstarter

The algorithm `AlgSGstarter` is a minimal ISR with essentially placeholder methods. It executes no BarrierSyncs apart from the Stop Message. Its purpose is to allow the main algorithms to be configured using parameter block transactions. The main algorithms can therefore swap in with a full set of parameters at the first frame. It is actually a little more complicated than this because the main algorithms then perform one ‘frame’ with the ADCs being triggered but no data actually being read. This is to deal with an artefact of the ADC initialisation whereby data are not produced after the first trigger.

#### 10.1.1.2 AlgSGtimer

This simple algorithm acts as a ‘slacker’ during data processing when it simply copies the LoveTrains containing the demand and output data. Its main purpose is to schedule the timer interrupt which will cause the next ISR invocation. Once invoked the timer algorithm automatically wakes up its neighbour by transmitting the cowcatcher in its ISR wrapper (as do all algorithms). Shortly after invocation the ISR sends a trigger pulse to the ADC trigger module. The re-scheduling of the interrupt takes place after transmission of the Stop Message. The parameter `interruptInterval` is used to control the timer interval.

#### 10.1.1.3 AlgSGadc

This is the main algorithm of the SG ring and runs on the six CPUs that have ADC data inputs. The structure of the ISR is as follows:

1. after the usual initialisation stages which follows the first few stage of `AlgNaomiInterleave`, a number of servo parameters are initialised on the first iteration.
2. If a parameter change is detected then the `waveform` parameters are checked for changes. The `waveform` system is used inject programmed motion (for example, a sine wave) onto a particular DAC channel (which could include the FSM tip-tilt control channels).
3. The `snapshot` variables are then initialised if required. The purpose of the `snapshot` system is to allow a full set of Strain Gauge ADC and output values sampled at the same time to be retrieved by a workstation process. This is achieved by setting a number of pointers to the snapshot buffers where copies of these data are held. The pointers normally point to dummy buffers where the copies are made anyway. This arrangement prevents the taking of a snapshot from interfering with timings by introducing additional copying.
4. The next step is to read the Strain Gauge ADCs via a DMA transfer. The number of channels to be transferred is controlled by ADCblockSize, which is configured to the value 32 or 64 depending on the ADC port which the CPU is attached.
5. The `waveform` playback system is linked to the developmental `capture` diagnostic system. The `capture` system is the antithesis of the `snapshot` system in that it captures a whole contiguous sequence of data values but only from one channel per CPU at any one time. The purpose of the linkage to the waveform system is to enable measurement with fine time resolution of the response of an actuator to a stimulus. Such a method would be valuable to the future implementation of Smith Compensation (feed forward) in the Strain Gauge control loop.
6. The next stage is to agree on the BarrierSync plan.
7. If the CPU has the demand interface that conveys data from the WFS ring then a DMA must be scheduled to read the data. This is done if the DMA is completed or not started. There is also a test system for copying in simulation data from the `initialDemand` parameter. The SG ring typically operates much faster than the WFS ring so several executions of the ISR may be expected to take place between each demand. There is an optional synchronisation system that may be used to delay the further execution of the ISR at the read point if this is the ISR execution in which a new demand is expected to arrive (based on the previous interval between demands). The purpose of the synchronisation arrangement is to allow each new WFS ring demand to be processed with the minimum of delay. If there is or is not a new demand present then this is signalled in a field of the `demandBuffer` parameter prior to the data being transmitted around the WFS ring using a BarrierSync:
   ```
   /* signal new demand */
   parameters->demandBuffer[SG_NEW_DEMAND_SIGNAL_CHANNEL] = 1;
   ```
8. If the current CPU does not have the demand interface then demand data are simply copied during the BarrierSync.

9. We now set up pointers into various parameter tables, including the demand block, the calibration slope (gain) and offset, the servo gain and the `passThrough` table. These parameter tables are the same on all ADC CPUs and include entries for all channels. Each CPU therefore needs an offset parameter, `tableOffset`, within these tables in order to select the entries for the channels which that CPU reads out.
10. The *feed forward* system is intended as the first stage of a developmental Smith Compensation system. It is controlled by the `feedforward` parameter and allows a new demand to be transmitted promptly without feedback to the mirror. It performs this by setting the pointers to the unpacked demand buffers to a special buffer for this purpose.
11. We now unpack the demand. It is transmitted from the WFS ring with two of the 13-bit values packed into each 32-bit word.
12. If we are using the feedforward system with the waveform system then we need to make sure that the current waveform value is copied into the feed forward demand buffer at this point.
13. We now check that the ADC data have all been read and panic if there is too long a delay in it arriving.
14. The data arrives from the ADCs in a fairly obscure order and each ADC CPU has a `reorderTable` parameter to re-order the ADC data into DAC channel order.
15. We start an ADC DMA to remove any additional unexpected data.
16. If the `feedforward` variable is switched on then we calculate how the new demand differs from the last demand and adjust the output accordingly, whilst checking the output value for exceeding saturation limits. The `feedforward` variable indicates that the `feedforward` parameter is set and a new demand has just arrived.
17. If we are not performing feedforward processing then we perform the servo processing for the CPU's data block in DAC channel order:

    ```
        for (iWord = 0; iWord < blockSize; iWord++) {
    ```

    The servo processing is performed using floating point arithmetic. The ADC value is converted into floating point format and then transformed into nominal DAC units using calibration tables loaded using the `ADCcalGain` and `ADCcalOffset` parameters. A simple servo algorithm is then executed and the integer format `output` buffer updated.
18. It is at this point that we use the `passThrough` parameter table to override servo processing on selected channels and to write the demand value directly to the channel in the `output` buffer.
19. The `output` buffer contents are then checked against the saturation limits.
20. The contents of the output buffer are now packed into a buffer for transmission around the ring by BarrierSync. The BarrierSync is then performed. This has the effect of distributing all the output data for all ADC CPUs around the SG ring. The mirrorCPU runs the algorithm `AlgSGmirror` which deals with outputting data to the mirror electronics.
21. We now check for extra data in the ADC input buffer and panic if any is found (as this represents a serious inconsistency).
22. It is at this point that the snapshot data are copied, whether or not they are required (in order to ensure constant timing)

23. The StopMessage is then transferred and processed.
24. Snapshot and capture data are then scheduled for transmission to the diagnostic CPU according to separate decimate parameters: `ADCsnapshotDecimate`, `ADCcaptureDecimate`. It is also possible to retrieve the snapshot data using the Parameter Block system.
25. The usual return or panic processing is then performed.

#### 10.1.1.4 AlgSGmirror

The ISR function of the `AlgSGmirror` algorithm is responsible for transmitting output data to the mirror electronics. The ISR processing follows the following sequence:

1. The initial processing corresponds to the `AlgNaomiInterleave` and other algorithms.
2. We initialise the `finalDemand` and `output` values to midrange (4096) on the first iteration.
3. the sync pulse is sent on the mirror syncPort. The mirror `finalDemand` is then written to the mirror electronics. Note that the output data from the last iteration (or from initialisation) is written to the mirror electronics at the *start* of the current ISR invocation.
4. We now agree on the BarrierSync plan using a call to `BSPbegin`.
5. The snapshot system of `AlgSGadc` extends to `AlgSGmirror` and in this case the data which are copied are the contents of the `finalDemand` and `inputDemand` buffers. Pointers to diagnostic or dummy buffers are adjusted in the same way as for `AlgSGadc`.
6. The BarrierSyncs for the demand data, the snapshot copy (or dummy) and the BarrierSync for the `AlgSGadc` output data are then performed.
7. The `StopMessage` is then transmitted.
8. We now unpack the demands from the output BarrierSync and unpack the data using the `DACreorderTable` parameter to determine which order the data blocks arrive from the ADC CPUs.
9. We now extract the tip-tilt data and check it for exceeding the saturation limits.
10. The algorithm swap processing is then performed if required.
11. Snapshot diagnostic data are then transmitted according to the `DACsnapshotDecimate` parameter. Note that snapshot data can also be retrieved using the Parameter Block system.
12. Finally we perform the usual return or panic processing.

# 11. Building the principal c40 programs

GNU `gmake` is used to build the programs from the `RealTime` directory.

# 12. RT configurations

RealTime configurations are specified in the `RealTime/pythonModules/RTconfig.py` python language source file.

GP transactions must be conducted with respect to *named* algorithms. As software is developed, however, it is sometimes necessary to change the names of algorithms allocated to a given processor and/or function. This is inevitable where two or more former algorithms are being merged, to implement load-balancing, for example. It is also desirable where functional development or differing supported hardware configuration have led to genuinely different algorithms supporting differing sets of parameter block transactions. At the same time, it is desirable, of course, to be able to write workstation support libraries which will perform certain generic functions with any running algorithm that supports that function and not to have a different library for every set of algorithm names. Clearly however such a generic function then needs to be able to identify the name of which algorithm is executing and whether it supports the desired function, and if so, what procedure is required to invoke it. The RTconfig system provides the capability to do this in a centralised fashion.

RTconfig introduces the idea of named *software configurations*. Tables in `RTconfig.py` hold the names of the executable programme files that must be loaded on each CPU in order to run a given configuration. It also holds the basic WFS and DM  configurations required to initialise each configuration. A named configuration can be booted simply by passing its name to the `c40Run` utility which then passes on the name to `Run` functions within `WFSlib` and `SGlib`. They in turn use the configuration name to look up the required c40 programme filenames and configuration tables using the services provided by RTconfig.

Software configurations therefore provide at once a means of being able to retrieve older c40 programmes without having to alter and recompile libraries, and also a means of loading different programmes to support different attached hardware. RTconfig provides some additional support for this latter application by introducing a hardware-software compatibility checking system. It works like this: at boot time the `Run` function can check compatibility using the `HWRTcompatible(conf)` function in RTconfig where `conf` is the name of the software configuration that is being booted. RTconfig retrieves basic information on the attached hardware that is recorded in the file `/software/Electra/save/RealTime/<host>.HWconfig` where `<host>` is replaced by the name of the c40 host computer, e.g., `aocontrol1`. It uses this, together with internal compatibility tables, to assess whether or not the proposed hardware/software combination is compatible.

After a software configuration has been successfully booted its name is recorded ad may be retrieved. RTconfig then uses the concept of *functional names* for algorithms to allow workstation processes to find out the real names of the algorithms that have been booted for the recorded software configuration name. For example, the functional name `“generic”` translates to `“ALG_NAOMI_INTERLEAVE”` under the `“Naomi”` software configuration but to `“ALG_GENERIC”` under the `“Electra”` configuration. A workstation support process can therefore deal transparently with either algorithm using the `“generic`” functional name.

There are various ways in which the implementation of RTconfig might be enhanced: (a) it currently stores the software configuration name in a file whereas it might be better to have it stored in the EPM database; (b) it implements a form of “inheritance” between configurations that might be better achieved using the object oriented programming features of the python language.

# 13. Workstation support libraries

## 13.1 GP and DTM libraries

Python and C language version of the GP and DTM libraries are available for workstation programmes to communicate with the c40 processors and each other. The most common way of sending normal GP commands is the python `GP.rpc` remote procedure call. This takes three arguments: the CPU (GP) number, the command identifying string, and a list of parameters to the command. For transactions from python procedures the `GPtransaction.Transact` call provides a convenient mechanism. It takes a single argument which is a list of transaction elements. Each element of this list is itself a list with the following fields: a (GP) CPU number, the name of the algorithm (see RT configurations above), the name of the parameter block section and a list of parameters. For retrieving parameter block sections the python function `GPtransaction.GetParam` is useful. It has only two arguments: the (GP) CPU number and the name of the parameter block section

### 13.1.1 Workstation C programs which talk to C40 programs

Here is a minimal C program to talk to the C40s.

```
#include <signal.h>
#include <stdlib.h>
#include "exception.h"
#include "packet.h"
#include "c40Commands.h"

int main(int argc, char **argv)
{
  GPmsgBuffer *buffer;
  ExcStatus *status = ExcStatusNew(1);

  GPsetup(status);
  buffer = GPallocBuffer(status);
  if (!ExcOK(status)) goto end;
  buffer->header.command = DO_PING;
  GPsendMsg(buffer, length, GP_PACK_ADDRESS(4,
PORT_BOOT), status);
  buffer = GPgetMsg(1000, status);
  if (!ExcOK(status)) goto end;

  if(buffer->header.command == DO_PING + GP_ACK
     && buffer->header.arg1 == 0)
        printf("Ping succeeded\n");
  else
        printf("Ping failed\n");

  GPshutdown();
  exit(0);

end:
  ExcToFile(status, stderr);
  GPshutdown();
  exit(1);
}
```

The program goes through the following steps:

It initialises the status variable, which holds the exception state and error messages. The argument of 1 to `ExcStatusNew()` causes the program to exit with an error if the system is out of memory.

It initialises the GP system with `GPsetup()`

It allocates a buffer and puts a command into the header

It sends the command to the appropriate CPU, in this case number 4.

It then waits for a response with a 1000 msec timeout

It checks to see if the response is correct. By convention, the C40 acknowledges all commands by incrementing the initial command number by the value `GP_ACK` (1000). The status is, by convention returned in `header.arg1` and is zero on success.

It shuts down the GP system. This is important to free up ports on the `c40Comms` server.

It deals with any error messages by calling `ExcToFile`, which simply prints out any messages from any of the called subroutines. The reason the error messages are not printed in the subroutines is that the subroutines have no knowledge whether the calling program is connected to a terminal, or whether the program sends the error messages to a higher-level client.

Also, higher-level subroutines in the call chain may decide that an exception is not serious and call `ExcClear()` to clear the error condition without printing any messages.

## *13.2 WFSlib*

`WFSlib.py` provides general access to WFS ring functions to python workstation processes working below the EPM level. At this level only the minimal interlocking provided by the GP transaction system is operational and it is therefore appropriate for engineering level functions. Command line access to many of the functions is available via the `WFS` command which uses the `WFStest.py` library to access the functions in `WFSlib`. These commands are dealt with in the *c40 Users Guide*.

## *13.3 SGLIB*

`SGlib.py` provides general access to WFS ring functions to python workstation processes working below the EPM level. At this level only the minimal interlocking provided by the GP transaction system is operational and it is therefore appropriate for engineering level functions. Command line access to many of the functions is available via the `SG` command which uses the `SGtest.py` library to access the functions in `SGlib`. These commands are dealt with in the *c40 Users Guide*.

### *13.4 DeformableMirror*

`DeformableMirror.py` provides python access to deformable mirror manipulation functions at the sub-EPM (non-interlocked) level.

### *13.5 ReconLIB*

`ReconLib.py` provides python access to reconstructior matrix manipulation functions at the sub-EPM (non-interlocked) level.

### *13.6 Diagnostics libraries*

`GPdiag.py` provides python access to the c40 diagnostics system.

### *13.7 EPM aware libraries*

Python c40 support libraries exist above the EPM level and provide potentially full-interlocked access to the c40s. It is these libraries that are used by top level processes such as `TopGui` and the superscripts.

## 14. Workstation programmes

In addition to `TopGui` there are a number of workstation programmes which access the c40s. These include `FisbaGui`, which deals with initial mirror flattening using the interferometer, `WhiteLightProcedure`, which deals with full flattening using the WFS and `IngridAlign` which deals with the removal of non-common-path errors in the optical path to the science camera.

## 15. Programmes and files

[insert SJG material]

## 16. Work in progress

Implementation of 4x4 subaperture modes
Implementation of synched WFS modes
Improved load-balancing on the SG c40s
Investigation of optimal read/write timing on the SG c40s
[Many other lower priorities]

## 17. Obsolete and little-used software

| | |
|---|---|
| `WFSAlign` | Non-EPM C programme (GUI) for WFS alignment and visualisation |
| `MirrorMimic` | Non-EPM C programme (GUI) for DM control |
| `RTengGui` | python non-EPM Real-time control GUI |
| `DataDiag` | Non-EPM C diagnostic launcher |

[Many other systems to catalogue]