EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

VLT Software

IRACE-DCS

User Manual

Doc.No. VLT-MAN-ESO-14100-1878

Issue 1.4

Date 05/12/03

	Name	Date	Signature
Prepared	J.Stegmeier	05/12/03	
Approved	G.Finger		
Released	A.Moorwood		

Change Record

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
1.0	19/07/99	All	First preparation
1.1	19/10/99	6	Updates for new Infrared Acquisition Module
1.2	03/04/00	6.4, 6.8, 6.9	New server command line options
		6.4.4, 8.5	Multiple database instances
		6.4.4.2	Changed definition of exposure states
		6.4.3	WAIT command
		11.1	Updated CDT acoording to inscStdCmds.cdt
		11.2, 11.3, 11.4	Updated man pages
		12.1, 12.2	Updated examples
		5	Removed obsolete examples
1.3	12/02/01	2, 3	SBT-interface is obsolete
		6	New naming scheme, updated database, image post-processing, burst mode, detector modes
		8	Updated installation procedure
1.4	05/12/03	All	Updated for new platforms (Linux, VxWorks) and interfaces (IRACE PCI-Bus interface)
		6	Detector Mosaics
		10	Added database to reference

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE

This document describes in detail how to use the infrared detector control software and the IRACE controller. It also gives an introduction in how to apply application specific extensions and data pre-processing algorithms. Most parts of this document are also available via an on-line manual (‘*iracqStart -man*’ from the UNIX-shell or ‘*Extended Help*’ in the control panel).

1.2 APPLICABLE DOCUMENTS

The following documents, of the exact issue shown, form a part of this document to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this document, the contents of this document shall be considered as a superseding requirement.

[1] VLT-PRO-ESO-10000-0228, 1.0 10/03/93 - VLT Software - Programming Standards

1.3 REFERENCE DOCUMENTS

The following documents are referenced in this document.

[2] VLT-TRE-ESO-14100-1654 - Infrared Array Control Electronics (IRACE) Design Description

[3] SCD User’s Guide, Engineering Design Team

[4] PCI CD User’s Guide, Engineering Design Team

[5] VLT-MAN-ESO-14100-2108 - VLT Software - IRACE-DCS Real-Time Display Application, User Manual

[6] VLT-MAN-ESO-14650-3181 - VLT Software - IRACE PCI-BUS Interface Driver, User Manual

[7] VLT-MAN_ESO-14100-2457 - VLT Software - IRACE VME-BUS Interface Driver, User Manual

1.4 ABBREVIATIONS AND ACRONYMS

The following abbreviations and acronyms are used in this document:

ADC	Analog Digital Converter
BPP	Bidirectional Parallel Port (SPARC-SBUS)
CCS	Central Control Software
CI	Command Interpreter
CLDC	Clock converter and DC-voltage generator
DAC	Digital Analog Converter
DFE	Detector Front Electronics
DMA	Direct Memory Access
DCS	Detector Control Software
DTT	Data Transfer Task
ECCS	Extended CCS
HW	Hardware
ICMD	IRACE Command Interface Module

IEEE	Institute of Electrical and Electronics Engineers
IF	Interface
IFP	Interface Protocol
I/O	Input/Output
IPC	IRACE PCI-Bus Interface
IRACE	Infrared Array Control Electronics
IRACQ	Infrared Acquisition (software module)
ISEQ	IRACE Sequencer
ISO	International Standardisation Organisation
LAN	Local Area Network
LCU	Local Control Unit
N/A	Not Applicable
NC	Number Cruncher
PCI	Peripheral Component Interconnect
PP	Parallel Port
PPS	Pre-Processing System
RTC	Real Time Computer
SCL	Sequencer- and CLDC-module
SCSI	Small Computer System Interface
SW	Software
TBC	To Be Confirmed
TBD	To Be Defined
TCOM	Transputer Communication Module
VLT	Very Large Telescope
WS	Workstation

1.5 GLOSSARY

Clock pattern
The consecution of logical states necessary to clock the read-out of the detector;

OS-link
Oversampled link (INMOS T2/T4/T8 series);

Sequencer
Irace Sequencer, which generates the clock patterns for detector read-out;

Subpattern
Block of logical clock states of the sequencer module; The clock pattern is built from these units.

1.6 STYLISTIC CONVENTIONS

The following styles are used:

bold
: in the text, for commands, filenames, pre/suffixes as they have to be typed.

italic
: in the text, for parts that have to be substituted with the real content before typing.

`teletype`
: for examples.

`<name>`
: in the examples, for parts that have to be substituted with the real content before typing.

bold and *italic* are also used to highlight words.

2 OVERVIEW

2.1 IRACE SYSTEM ARCHITECTURE

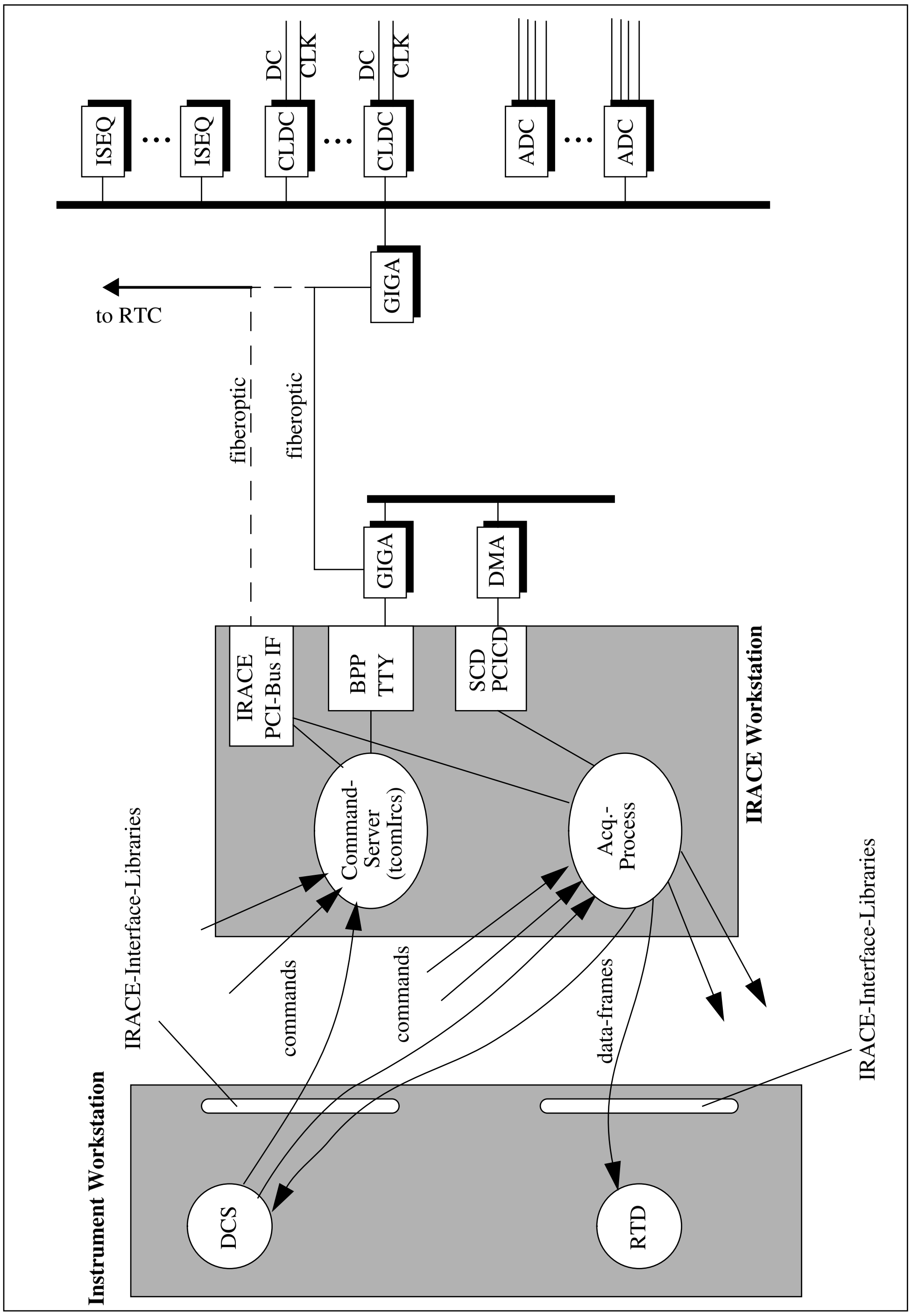


Figure 1: IRACE System-Architecture

2.2 IRACE SOFTWARE MODULE

The ***irace*** software module consists of three parts - the IRACE front-end controller software, the data acquisition process and the interface libraries.

The front-end controller software runs on the micro-processor network of the IRACE controller. This software is booted and controlled via an interface process (IRACE command server process - ***tcomIrcs***), which runs on the IRACE Workstation. The bootable files are delivered together with the ***irace*** software module. They are automatically installed in the ***$INTROOT/config/irace/*** directory by the Makefile of the module.

The tasks on the IRACE Workstation are started as remote-processes from the Instrument Workstation. The remote exec- (and remote kill-) functions are part of the interface libraries supplied together with the ***irace*** module.

For applications which require hard real-time performance (such as control loops for adaptive optics) the IRACE controller has to interface to a dedicated Real-Time Computer (RTC). The current Real-Time OS standard at ESO is VxWorks running on a PowerPC architecture. Other Real-Time-Computer (RTC) standards are under investigation (to be considered as a black box). A VME-Bus data interface is available for the IRACE controller. The appropriate driver for the VxWorks operating system and an appropriate capture task-handling is described in [7].

2.2.1 IRACE COMMAND SERVER PROCESS

This process does the protocol conversion to forward commands to the IRACE controller and provides the system services:

- open link
- close link
- reset
- boot
- setting of link timeout

Currently the IRACE command server process can handle three physical link interfaces:

BPP - Sun SBus-Bidirectional Parallel Port (this is the standard parallel port of the SBus based SUN Ultra-Sparc).
TTY - Sun PCI-Bus via serial port.
IPC - IRACE PCI-Bus Interface.

2.2.2 ACQUISITION PROCESS

The data acquisition process is a multi-threaded process, which runs on the IRACE Workstation. It provides the following parallel tasks:

- Command Interpreter Task (highest priority) for start/stop/parameter-setup and status commands

- Capture-Task (interface to DMA-driver; also does buffer-overrun checks))

- Calculation-Task (parallel computation algorithm)

- Output-Task(s) (transfer data-frames in parallel to any requesting client on any host)

The calculation task is designed as a main-process. The other tasks are started by the main process via functions, which are part of the ***sdma***-library. The ***sdma***-library contains all input-, output-, parameter- and synchronization functions.

This part of the IRACE software is supported for Linux, Sun-Solaris and for HP-UX (simulation mode only, no real-time performance).

2.2.3 IRACE INTERFACE LIBRARIES

The IRACE interface libraries provide all functions to control both the IRACE controller and the acquisition process. They also contain the functions for data reception. These interface libraries are supported for both HP-UX, Linux and Sun-Solaris.

2.2.4 SIMULATION MODE

In simulation mode a simulator process (***tcomScls***) is started instead of the IRACE command server process (***tcomIrcs***). This handles all sequencer, cldc - and status-bus functions and may run either on the IRACE workstation or on the Instrument Workstation.

The acquisition process has a built-in simulator. In simulation mode the input-ringbuffer is prefilled with data. The input sequence is simulated via timer functions (resolution: 1 ms per input-frame). For simulation on HP-UX the ***sdma***-submodule contains a thread-wrapper (***sdmaThr***-library) using posix-threads (no real-time performance).

2.3 THE INFRARED DATA ACQUISITION MODULE

The infrared data acquisition software module (***iracq***) is built on top of the IRACE interface libraries and coordinates the IRACE controller and the acquisition process. This is the interface to the VLT-software. The control server process and the data transfer task are executed on the Instrument Workstation and the RTD either on the Instrument Workstation or on the IRACE workstation.

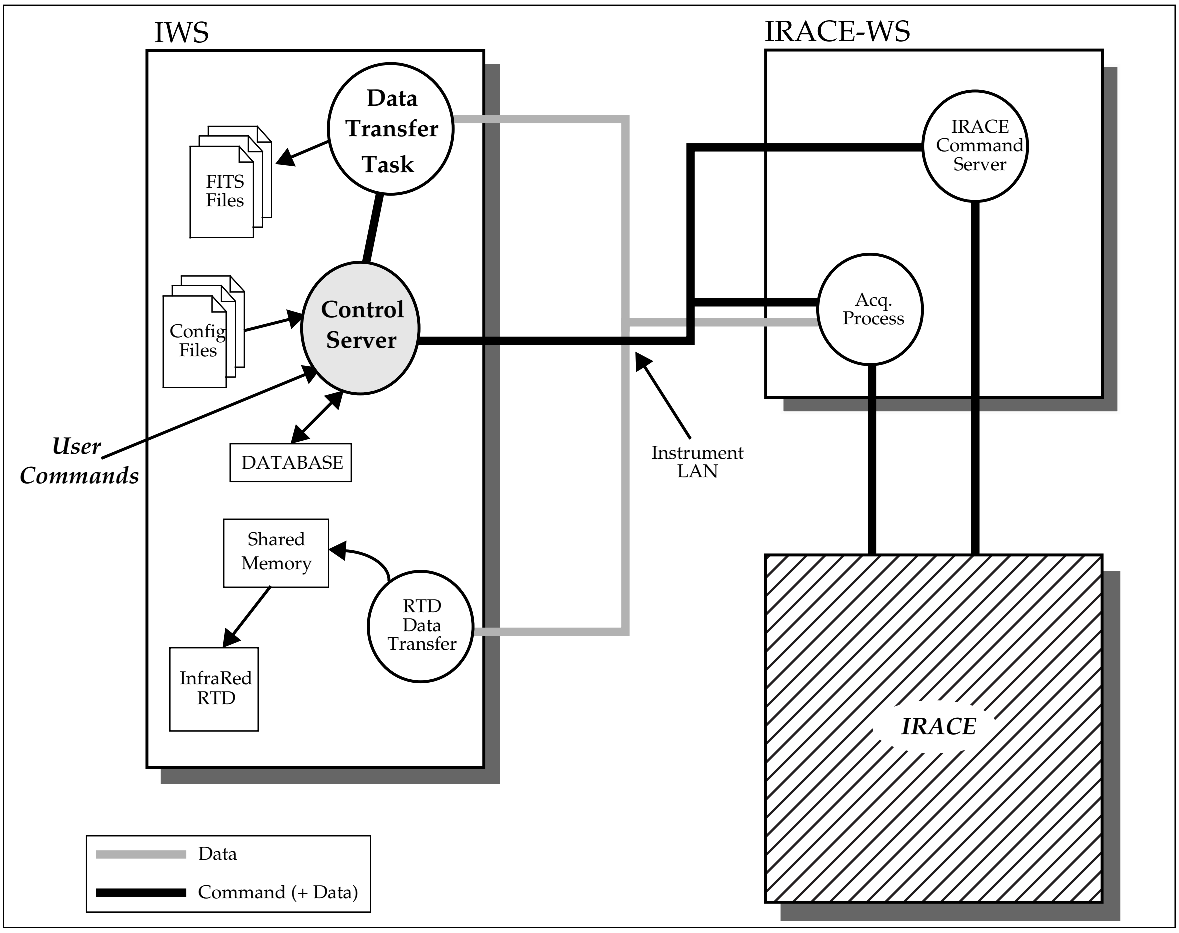


Figure 2: The DCS SW-Architecture

The Science Data is sent to the IWS and stored on the disk as FITS files. The Video Data (RTD data) is sent to the IWS (or to another WS) and displayed in the RTD.

All commands sent by the user must go through the Control Server on the IWS. Under normal circumstances no commands should be sent directly to other processes.

Not shown in Figure 2 is the RTD Server, which must be running in order to signal to the RTD application to display the data.

Infrared acquisition control processes:

Control Server	**iracqServer**	The control server is the interface process to the IRACE-DCS for the external world. Through this server all commands must pass. The server checks the validity of various commands and parameters according to the current state of DCS. The server is running in the IWS environment.
Data Transfer Task	**iracqDtt**	The Data Transfer Task requests and receives the science data from the IRACE-WS and stores the data in FITS Files. This process is started and stopped by the acquisition control server.

2.4 HOW TO BUILD YOUR OWN APPLICATION

An infrared instrument application consists of a set of configuration files for the IRACE front-end controller (see section 7) and a set of acquisition processes that implement detector read-out specific pre-processing algorithms. Section 4.4 describes in detail how to develop such an acquisition process. Selection and startup of the acquisition process is done by the infrared acquisition control server (***iracqServer***). The control server also maintains a dynamic parameter interface to the acquisition process.

To introduce instrument specific functions it is also possible to extend the standard server processes and to add application specific control/status widgets to the control panel. This is described in detail in chapter 6.

3 THE IRACE CONTROLLER

3.1 PHYSICAL INTERFACES

The IRACE controller can be driven via several physical interfaces (Figure 3).

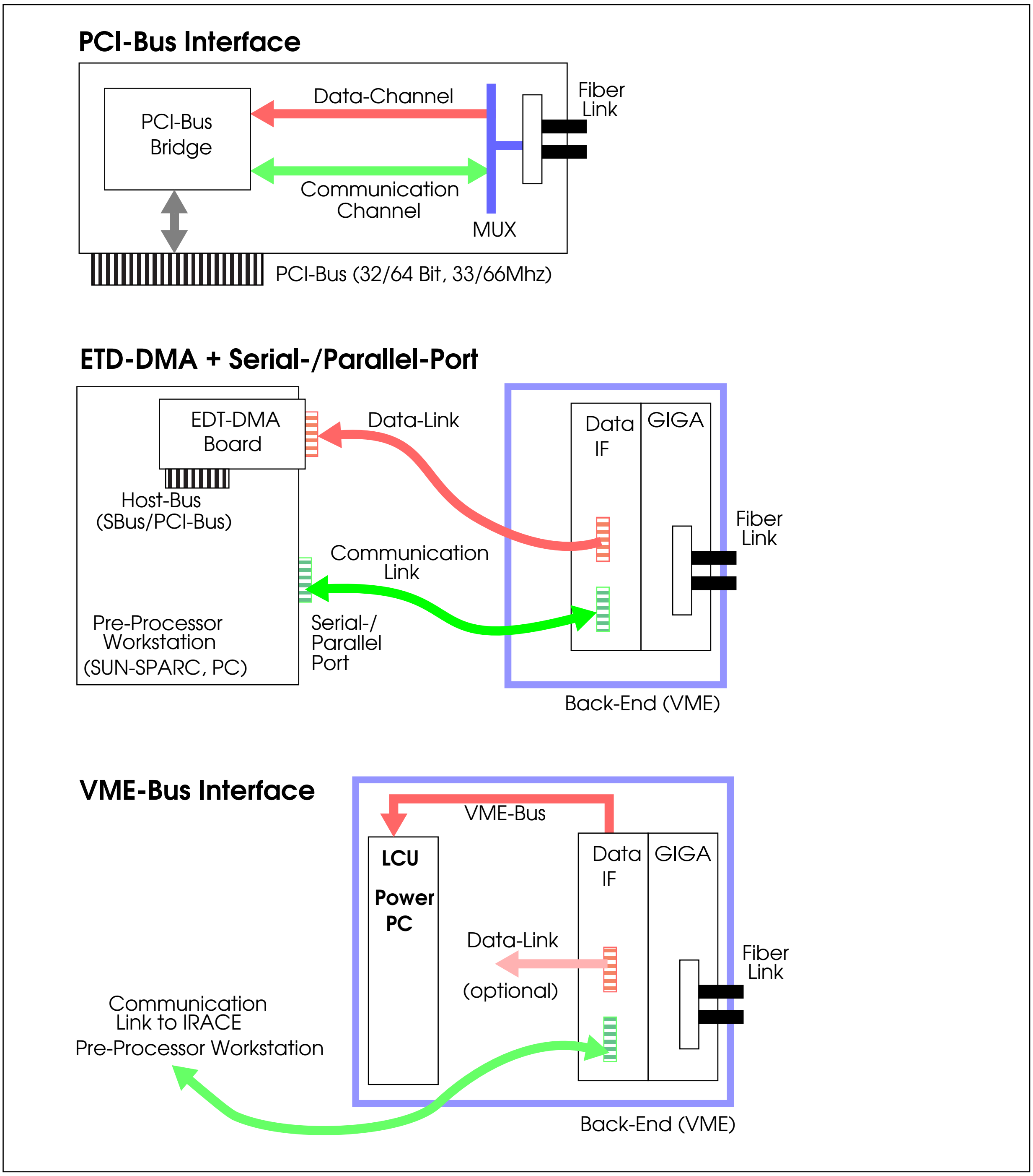


Figure 3: Physical Interfaces

3.1.1 BIDIRECTIONAL PARALLEL PORT INTERFACE

This is the current standard interface for the SBus based Ultra-Sparc.

System configuration file entry (see section 7.1):

```
DET.IRACE.SCLPROTTYPE  "bpp"
DET.IRACE.SCLDEV       "/dev/bpp0"
DET.IRACE.SCLBOOTFILE  "sclpp.btl" or "scl2pp.btl" for 2 CLDC boards
```

3.1.2 SERIAL PORT INTERFACE

This is the current standard interface for the PCI-Bus based Ultra-Sparc
The IRACE front end can also be connected to the SPARC via the serial port (ttya, ttyb).

System configuration file entry (see section 7.1):

```
DET.IRACE.SCLPROTTYPE  "ecpp"
DET.IRACE.SCLDEV       "/dev/ttya" or "/dev/ttyb"
```

3.1.3 EDT DMA INTERFACE

The EDT DMA interface board is available for both SBus (SCD60) and PCI-Bus (PCICD60). Both boards support ring-buffered DMA up to 60 MBytes/s.

System configuration file entry (see section 7.1):

```
DET.IRACE.ACQi.DEV     "/dev/scd0" or "/dev/pcd0"
```

3.1.4 IRACE PCI-BUS INTERFACE

ESO development for a direct interface between PCI-Bus and IRACE detector front-end. This will make the above three interfaces obsolete. The driver for this module is described in [6].

System configuration file entry (see section 7.1):

```
DET.IRACE.SCLPROTTYPE  "ipc"
DET.IRACE.SCLDEV       "/dev/ipc0_com"

DET.IRACE.ACQi.DEV     "/dev/ipc0_dma"
```

3.1.5 IRACE VME-BUS INTERFACE

For applications which require hard real-time performance (such as control loops fro adaptive optics) the IRACE controller has to interface to a dedicated Real-Time Computer (RTC). The current Real-Time OS standard at ESO is VxWorks running on a PowerPC architecture. Other Real-Time-Computer (RTC) standards are under investigation (to be considered as a black box). A VME-Bus data interface is available for the IRACE controller. The appropriate driver for the VxWorks operating system and an appropriate capture task-handling is described in [7].

3.2 MICRO-PROCESSOR NETWORK

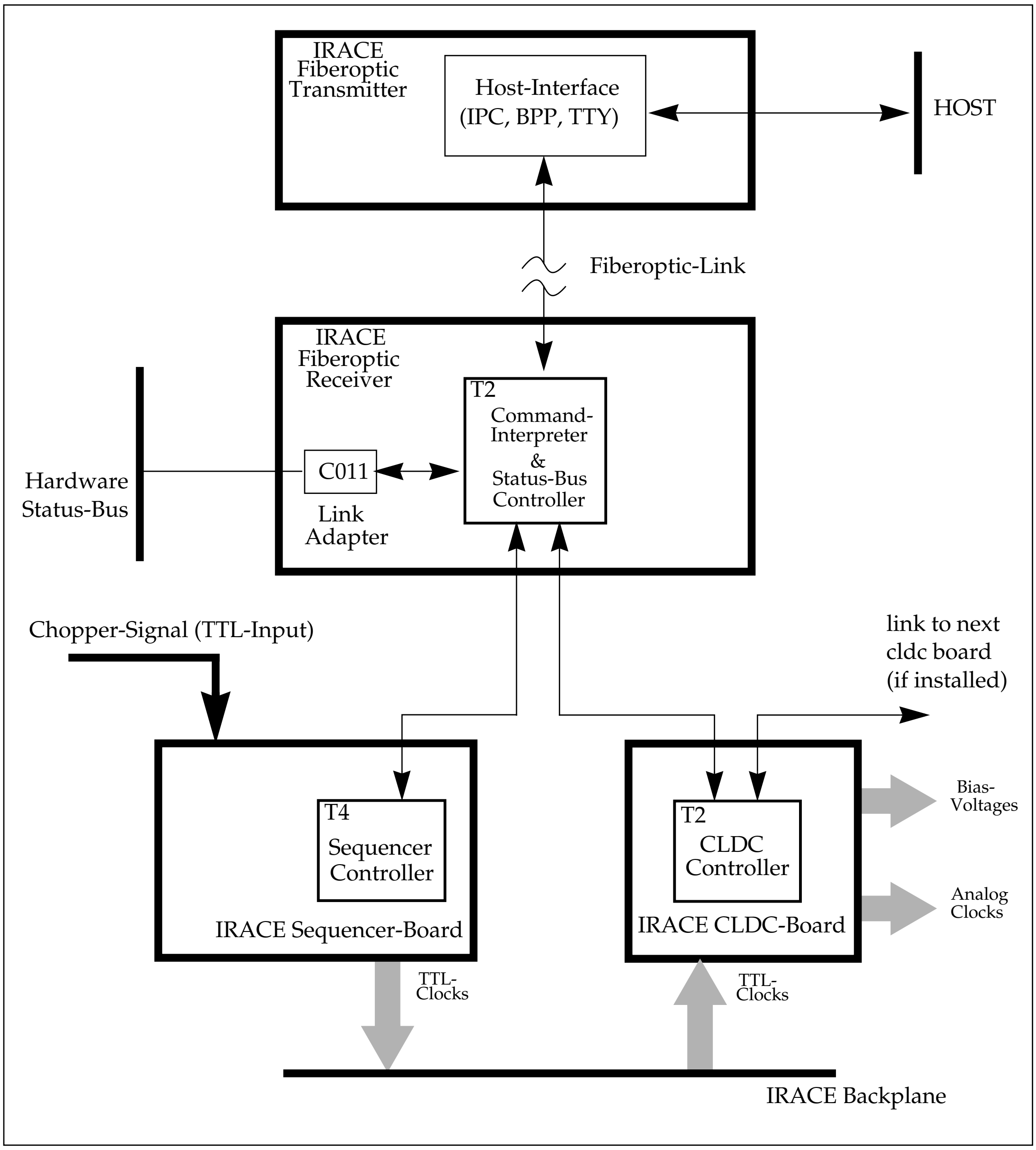


Figure 4: Processor Network

3.3 IRACE STATUS BUS

Control I/O to all connected HW-modules can be done via the status bus. The status bus is connected to the processor on the TIF-module via a link adapter. Each HW-module on the status bus is addressed by a unique module number (4 bit). The status-id (4 bit) selects an 8-bit status register on the module. An error-bit is used to detect error states like cable-break, overflow or loss of synchronization. If the error-bit is set on the status bus, a broadcast has to be done to all modules to find the one, which has generated the error. It is also possible to do some hardware configuration via the status bus, like setting filters on the ADC-board or switching LED's on and off.

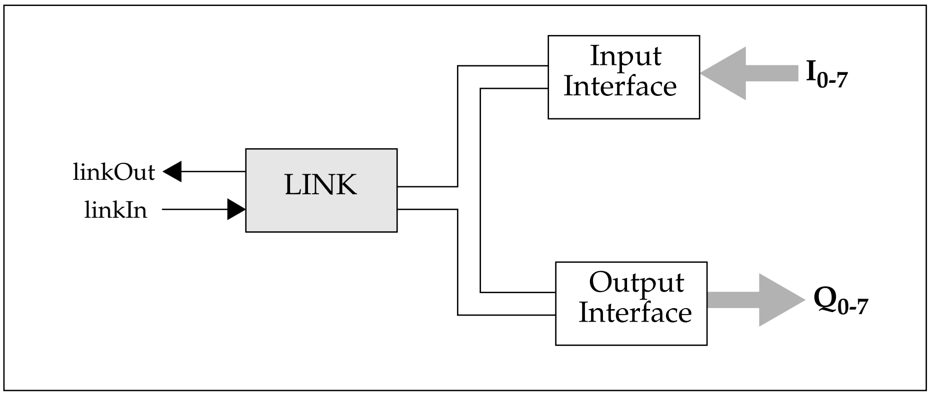


Figure 5: status bus link adapter

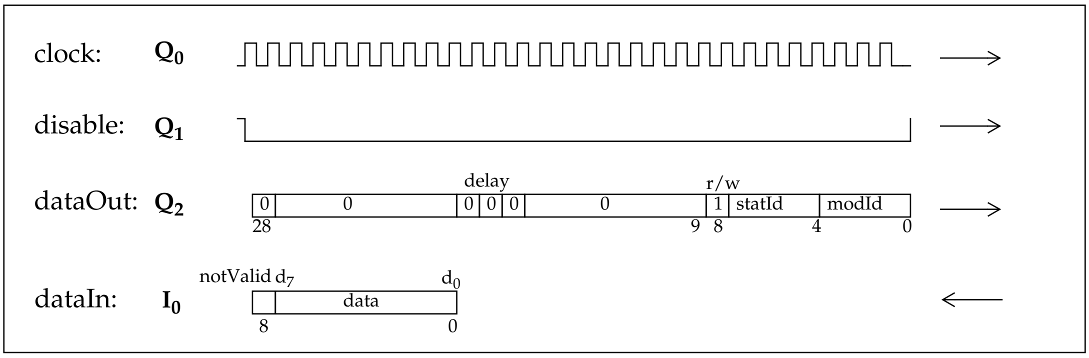


Figure 6: input from status bus

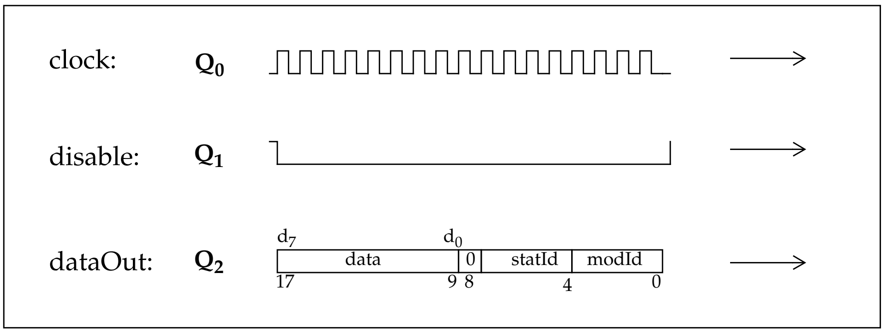


Figure 7: output to status bus

If the module number is not valid on the status bus, the **notValid** bit in **dataIn** will be set to one. Link input (**I_{0-7}**) and link output (**Q_{0-7}**) are done in parallel during the status read phase. While there is no status bus I/O request, the status control process just waits for **linkIn.** If a module goes in error state, it will initiate a **dataIn** phase and set the error bit (**d_7**). The control process will hold the error condition, until it receives an error request command from the command interpreter. Afterwards the host can do a status read from all modules connected to the status bus and can check, if the error bit is set.

3.4 SEQUENCER

The sequencer system processor is a 32-bit transputer (T425) with 8 MBytes DRAM. A detailed description of the sequencer hardware can be found in [2].

3.4.1 CLOCK PATTERNS

The clock patterns produced by the sequencer are composed of a set of sub-patterns. Each sub-pattern consists of a number of logical states for the 48 sequencer clocks. Each state has a 7 bit time-counter which defines a time per state between 50 ns (= sequencer clock rate) and 6.3 us. Each sub-pattern can have up to 128 states. A maximum number of 256 different sub-patterns can be stored in the SRAM-pages of the sequencer processor. The IRACE front-end back-plane routes the first 32 clocks to two possible CLDC-boards. Clocks 33 and 34 are the convert strobes for two possible groups of ADC-boards. Clocks 35,36 are reserved. Clock 37 is used to invert the lowest bit of the header of the ADC-boards which are attached to convert strobe 33 or 34. Clocks 45-48 are available on lemo-connectors on the front-panel of the sequencer board. Clocks 38-48 are not connected to the back-plane.

The clock-patterns are defined in an ASCII file (SHORT-FITS format) which is referred to in the detector configuration. The clock-pattern files are located in

```
"$INS_ROOT/$INS_USER/MISC/IRACE/CLK/*.clk"
```

The parameter ***DET.SUBPAT.NO*** defines the number of sub-patterns, which are configured in the file. ***DET.CLKP.NO*** specifies the number of clocks used by the detector. These clocks have to be mapped onto the 48 physical sequencer clocks:

Example for 16 defined clocks:

```
DET.CLKP.SWHWCLK1           "1,2,3,4,5,";
DET.CLKP.SWHWCLK2           "6,7,8,9,10,";
DET.CLKP.SWHWCLK3           "11,12,13,33,46,";
DET.CLKP.SWHWCLK4           "47";
```

In the following ***DET.CLKP.NO*** sub-pattern blocks are configured. The index (i) is used as reference for the sub-pattern address.

```
DET.SUBPATi.NAME            "<name>";
DET.SUBPATi.STATES          <number of states>;
DET.SUBPATi.RFAC            1; # always 1 for backwards compatibility
DET.SUBPATi.STATEV1         "111111111111";
DET.SUBPATi.STATEV2         "111111000001";
DET.SUBPATi.STATEV3         "000001111111";
DET.SUBPATi.STATEV4         "111111111111";
DET.SUBPATi.STATEV5         "111111111111";
DET.SUBPATi.STATEV6         "000000000000";
DET.SUBPATi.STATEV7         "111111111111";
DET.SUBPATi.STATEV8         "111111111111";
DET.SUBPATi.STATEV9         "111111111111";
DET.SUBPATi.STATEV10        "111111111111";
DET.SUBPATi.STATEV11        "000000000000";
DET.SUBPATi.STATEV12        "000000000000";
DET.SUBPATi.STATEV13        "000000000000";
DET.SUBPATi.STATEV14        "000010000010";
DET.SUBPATi.STATEV15        "000010000010";
DET.SUBPATi.STATEV16        "000000000000";
DET.SUBPATi.RSPEEDV         "5,5,5,5,8,4,5,5,5,5,8,4";
DET.SUBPATi.RSPEEDP         "1,1,1,1,0,0,1,1,1,1,0,0";
```

The ***DET.SUBPATi.RSPEEDV*** vector defines the reference duration for each state. This value can be multiplied later by a global read-speed factor (setup parameter ***DET.RSPEED***). A fine-tuning by adding clock-cycles in multiples of 50 ns can be achieved by setting the ***DET.RSPEEDADD*** parameter, which can also have negative values. The setup parameter ***DET.RSPEED*** cannot be set to zero, because when setting a new factor all state counters have to be divided internally by the previous factor. The ***DET.SUBPATi.RSPEEDP*** vector defines for each state whether its reference duration is tunable via the global read-speed factor/add or not.

3.4.2 SEQUENCER SUBPATTERN DISPATCHER

The execution of subpatterns is controlled by a vector (32 Bit) containing the addresses and repetition factors for a sequence of sub-patterns. The vector is transferred to a FIFO (4K x 16 bit) in blocks of 2K x 16 bit. Several vectors of variable length can be stored in the DRAM of the sequencer processor to define sequences for detector read-out, delay-patterns, detector reset or different window read-outs.

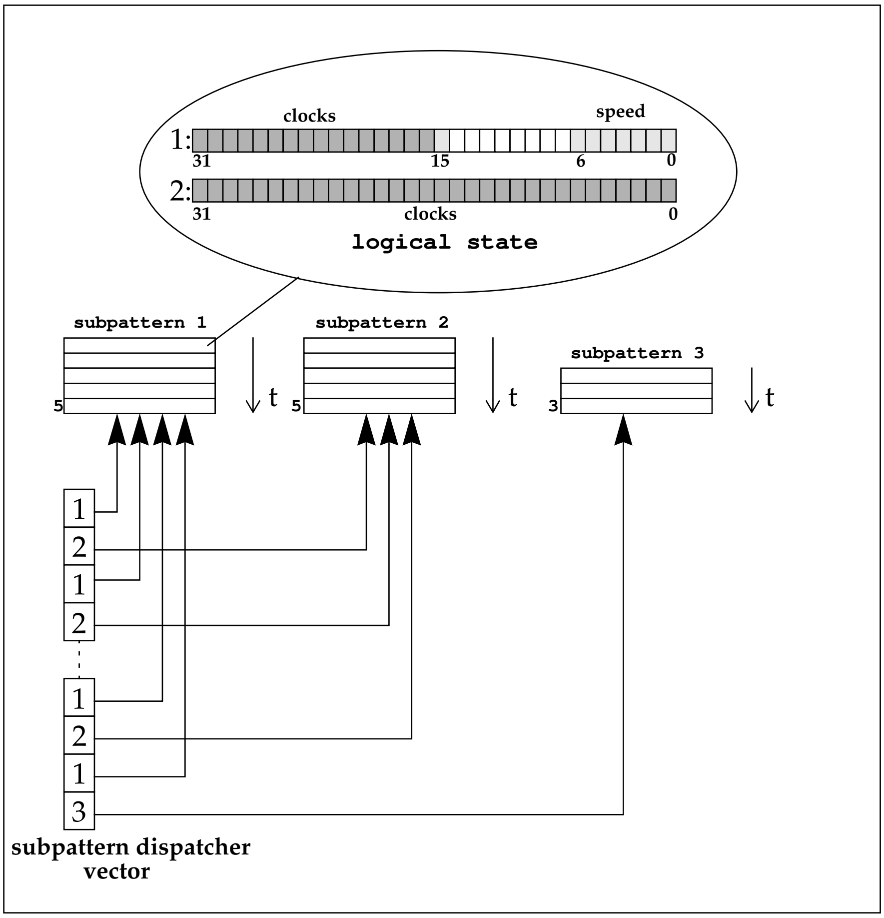


Figure 8: subpattern organization

For each vector the object-code (loop-structure) of a sub-pattern dispatcher program is downloaded to the sequencer. The code is interpreted to generate the vectors before the sequence is started. This avoids the downloading overhead, as the subpattern-dispatcher vector may have a size of several MBytes.

3.4.3 SEQUENCER PROGRAM LOOP

The order of read-outs is defined by the sequencer program, which has the same loop-structure as the subpattern-dispatcher. The sequencer program is executed at run-time. Each readout refers to one subpattern dispatcher vector. Examples for a high-level loop implementation can be found in section 7.6.

3.4.4 SEQUENCER MODES

The sequencer can operate in two different modes. In normal running mode the sequencer starts immediately after receiving the start-command. It terminates, when the sequencer program loop terminates or when a stop-command is received.

In external trigger mode after receiving the start-command the sequencer waits for a pulse on the first TTL-input and then starts executing the sequencer program loop. When the program loop reaches a synchronization point, the execution is suspended, until the next pulse on the other TTL-input is received. Synchronization points are either the beginning or the end of the sequence or explicit synchronization markers (“***SYNC***”) within the sequencer program loop.

The sequencer mode can be changed with the setup command “***SETUP -function DET.IRACE.SEQMODE 0|1***”, where ‘0’ refers to normal running mode and ‘1’ refers to external trigger mode.

The database attribute is ‘<alias>iracq:irace.seqMode’.

3.4.5 SEQUENCER RUN-CONTROL

The sequencer is started/stopped with the “***SEQ -start|stop***” command. The stop command will stop the sequence after the current dispatcher vector has been transferred to the sequencer FIFO (i.e. the sequencer will not interrupt a detector readout). It is also possible to interrupt the running sequence immediately. This is done when the system switches to ***STANDBY*** or ***LOADED*** state. In any case the current FIFO content is transferred and the function will wait until the FIFO is empty.

3.5 CLOCK AND DC-BIAS DRIVER (CLDC)

The Clock and DC-Bias driver (CLDC) provides 16 clock and 16 bias voltage generators with an amplitude range of +/- 10 V. Clock high/low levels and bias voltages are generated by 12-bit DAC’s. At power-up all outputs are disabled and can be enable/disabled with the ‘***CLDC -enable/disable***’ command. Several CLDC-boards can be installed within one IRACE front-end. The board to witch all further CLDC-commands refer is set with the ‘***CLDC -board <num>***’ command. A detailed description of the CLDC-hardware can be found in [2].

The multiple CLDC-boards can be used to control either one or several detector configurations. If only one CLDC board is used, the ***DET.IRACE.CLDC*** parameter can be set in the detector configuration to specify the number of the CLDC board that should be used for the detector (the board numbers start with zero). In this case the ***DET.IRACE.VOLTAGES*** keyword refers to the voltage setup file for the single CLDC-board.

If more than one CLDC-board is used, the ***DET.IRACE.CLDC*** parameter must be omitted and the index (i-1) of ***DET.IRACE.VOLTAGESi*** is used to retrieve the CLDC-board numbers. In that case the ‘***CLDC -enable/disable***’ command refers to all CLDC-boards configured in the detector configuration file. Please note that the indices start with 1, so ***DET.IRACE.VOLTAGES1*** will configure CLDC-board 0.

The voltage setup file is an ASCII-file defining the set-values for all clock- and bias-voltages for one

CLDC-board. The voltage setup files are located in:

```
"$INS_ROOT/$INS_USER/MISC/IRACE/VOLTAGES/*.v"
```

The telemetry can be retrieved with the '***CLDC -readAllVoltages***' command. The format of the result is:

```
<# of clocks><clk_hi_1><clk_lo_1> ... <clk_hi_n><clk_lo_n>
<# of bias><DC_1_tel_1><DC_1_tel_2> ... <DC_n_tel_1><DC_n_tel_2>
```

3.6 ADC CONTROL

The IRACE ADC-boards have several parameters to be configured via the IRACE-DCS. The configuration is done via IRACE front-end "*STATUS BUS*" (see section 3.3).

The system configuration keywords

```
DET.IRACE.ADCi.ADDR<address>;
DET.IRACE.ADCi.NAME<name>;
```

define an address and a name for each ADC-board in the system. The address is unique (slot number in the front-end) while the name can be shared by several boards. All ADC-boards with the same name build a group. If a value is changed from the control panel for one ADC-board, the same value is applied also for all other ADC-boards within the same group.

Default values for all ADC-boards of an individual detector setup have to be defined in the detector configuration.

```
DET.IRACE.ADCi.HEADER   <header>;  # header on IRACE data-Bus
DET.IRACE.ADCi.ENABLE   0|1;       # enable/disable (0/1)
DET.IRACE.ADCi.FILTER   0|1;       # filter1 off/on (0/1)
DET.IRACE.ADCi.FILTER2  0|1;       # filter2 off/on (0/1)
DET.IRACE.ADCi.DELAY    <delay>;   # conversion strobe delay (0-15)
```

The value for the ADC-boards are stored in a table ('*name, address, header, enable, filter1, filter2, delay*') in the database attribute '***<alias>iracq:irace.adcStatus***'. The header is assigned to all data packets created by one ADC-board. It has to be resolved by acquisition interface-devices in the IRACE back-end to route data from the ADC-boards to different targets. The header for the acquisition interface-devices is declared in the system configuration file via the ***DET.IRACE.ACQi.HDR*** keyword. If the ***DET.IRACE.ACQi.HDR*** keyword is not specified, a default header (1) is used.

4 THE PRE-PROCESSOR

4.1 SOFTWARE ARCHITECTURE

The incoming data are read via DMA into a ringbuffer. The capture process provides the synchronization between the DMA-driver and the calculation process. A ringbuffer overflow is also detected by the capture process. When processing the data, the result frame will also be stored in a ringbuffer to guarantee continuous data flow. Any data transfer is fully parallel to the processing loop (i.e. while transferring the result, the next result is computed). There are no memory copies requiered. When the frame has been processed and should be transferred, it is given to the data server. From here the frames can be distributed in parallel to several hosts via parallel data transfer threads. The frames are transferred on request. The request may be FIFO (for science data transfer) or LIFO (for video data transfer). Different frame types can be transferred in parallel to different requesting data clients (i.e. RTD shows raw frames while the averaged frame is transferred and stored to a FITS-file). Using this method the transfer capacity is only lmited by network bandwidth and the computing power of the requesting client process. It is possible to download data sets like flatfields or bad-pixel masks. The data sets can be stored in shared memory and can be shared between acquisition processes.

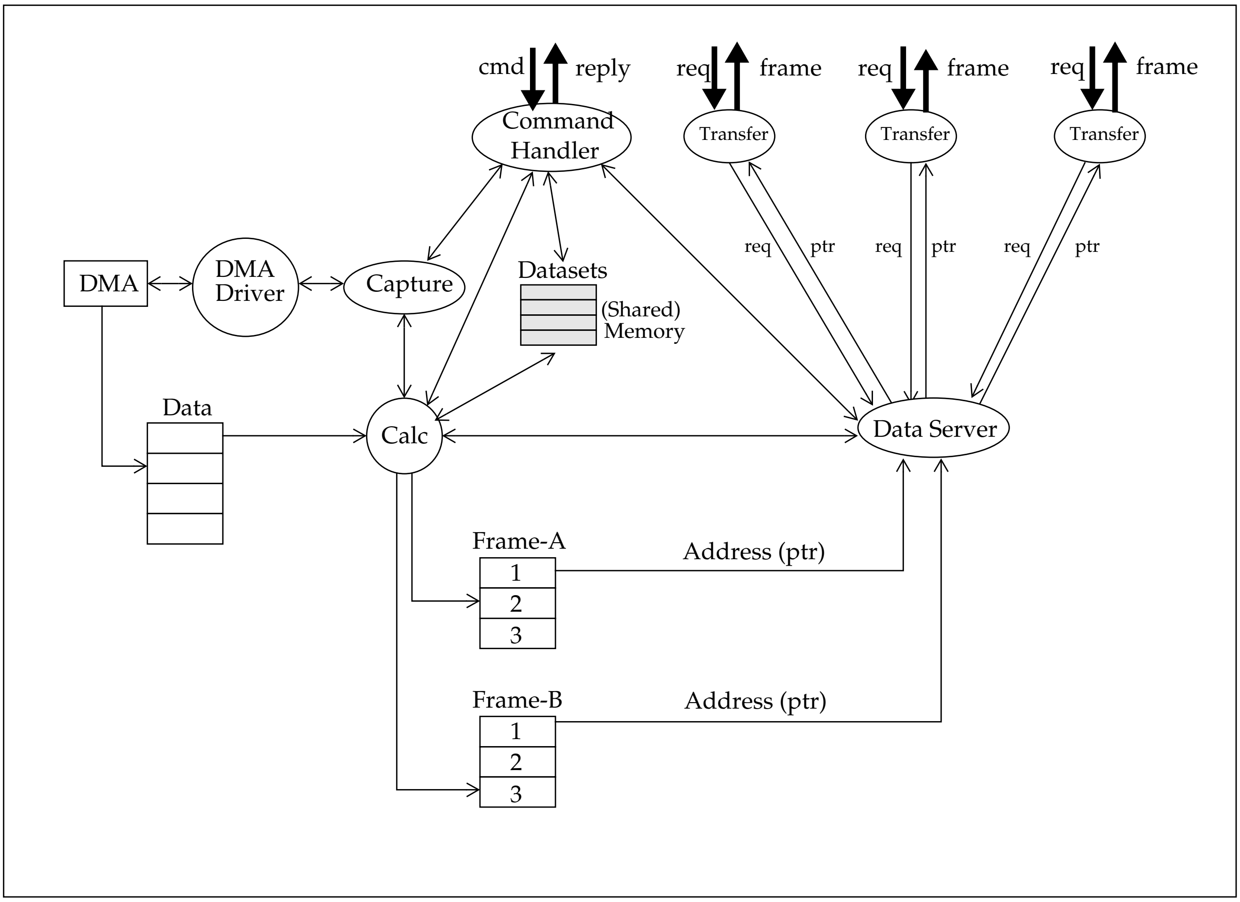


Figure 9: Pre-Processor Software Architecture

4.2 DMA INTERFACE

Several DMA interfaces are supported. The IRACE PCI-Bus interface (as described in [6]) is the current standard for all Linux installations. For SUN Ultra Sparc the commercial products from EDT are used. The EDT DMA interfaces for SBus (EDT-SCD60) and PCI-Bus (EDT-PCICD60) are pin compatible. Both EDT DMA-interfaces support a maximum input data rate of 30 MHz x 16 Bit (SCD/PCICD 60). As the drivers for all machine types are completely different, a driver interface library is included in the ***sdma*** submodule. Depending on the type a different driver library is linked with the acquisition process. The DMA interface type is fully transparent to the acquisition software.

4.3 ACQUISITION PROCESSES

The acquisition processes can be started as individual tasks on any computer running a Linux, Sun-Solaris or HP-UX (simulation only) operating system:

```
usage: sdmaXX [options]
options:
  -i                    enable interactive mode
  -hdr                  device interface header
                        (default: 0)
  -im <input mode>      input mode (thread, sim, mem)
                        (default: thread)
  -cmdport <portNum>    set command server port (default: 8011)
                        (process name for ccs)
  -dataport <portnum>   set data server port (default: 8012)
  -errport <portNum>    set error stack server port (default: 8013)
  -nbuf <num>           size of buffer queue (default: num = 3)
  -burst <n>            number of burst buffers (default: n = 0)
  -nclient <numClient> number of output links (default: numClient = 0)
  -ndet <n>             number of detector instances (default: n = 1)
  -nd <numDev>          number of dma devices (default: numDev = 1)
  -dev <n> <device>     device name for device <n>
                        (default: /dev/scd0 (device 0))
  -st <time>            simulation interval time in ms
                        (default: simulation time = 500 ms)
  -pars                 just print parameter setup
  -v                    switch verbose mode on
  -h                    show this
```

Most readout modes additionally support the '***-nx <pixels>***', '***-ny <pixels>***' option to specify the input frame format. The '***-ndet***' option specifies the number of detector instances. This is needed by the output server for the correct handling of window requests, if the frame buffer contains formatted images of more than one detector. The application can access this parameter via the global variable ***sdmaNumDet***. If the read-out mode supports more than one detector instance (see section 6.4.14), it should allocate buffers for [***NX*** * ***NY*** * ***sdmaNumDet***] sized frames. The frame header elements should be set to:

```
frame.h.nx = NX;
frame.h.ny = NY * sdmaNumDet;
```

If the '*-i*' option is not specified, the acquisition process starts the data capture and processing itself, using the default parameter setting (auto-start). This mode is reserved for test purposes. For using a command interface the acquisition process has to be invoked with the '*-i*' option.

4.4 HOW TO DEVELOP AN ACQUISITION PROCESS

Now we will develop a new acquisition process from scratch. The final version '***sdmaTemplate.c***' is part of the IRACE module and can be found together with the standard acquisition processes in the test directory of the ***sdma*** sub-module. The standard vltMakefile should be used to build the process. The following flags have to be set:

All platforms:

MAKE_NOSHARED = 'on'

LINUX PC:

USER_CFLAGS = -DLINUX -DIRPCI -DTHR_POSIX -D_XOPEN_SOURCE
OPTIMIZE = 3

USER_INC = -I/opt/EDTscd -I$VLTROOT)/include/rtd

xxxx_LDFLAGS = -lpthread
xxxx_LIBS = sdma sdmaDrv sdmaThr icmdSock icmd m rtdImgEvt

Sparc SBus:

USER_CFLAGS = -DSPARC_SBUS
OPTIMIZE = 3

USER_INC = -I/opt/EDTscd -I$VLTROOT)/include/rtd

xxxx_LDFLAGS = -lthread -ldl
xxxx_LIBS = sdma sdmaDrv sdmaThr icmdSock icmd socket nsl m rtdImgEvt

Sparc PCI-Bus:

USER_CFLAGS = -DSPARC_PCI
OPTIMIZE = 3

USER_INC = -I/opt/EDTpcd -I$VLTROOT)/include/rtd
USER_LIB = -L/opt/EDTpcd

xxxx_LDFLAGS = -ledt -lthread -ldl
xxxx_LIBS = sdma sdmaDrv sdmaThr icmdSock icmd socket nsl m rtdImgEvt

HP_UX:

USER_CFLAGS = -DHP -DTHR_POSIX
OPTIMIZE = 0

USER_INC = -I$VLTROOT)/include/rtd

xxxx_LDFLAGS = -lpthread
xxxx_LIBS = sdma sdmaThr icmdSock icmd m rtdImgEvt

4.4.1 BASIC SYSTEM INITIALIZATION

In a first approach, a main process is generated and the basic system initialization is done. The *sdmaStartup()* function initializes the data acquisition according to the command line arguments. It also sets up the data output server and the command interpreter. The ringbuffer size (number of bytes per element) is passed as parameter to the function. It is up to the application to compute the needed size. Afterwards the dynamic parameter structure (***myPARAM_TYPE***) has to be passed to the system. This is done via the ***sdmaSetupDynParam()*** function. The structure ***myPARAM_TYPE*** and the associated parameter names have to be defined by the application:

```
static char *paramId[] =  {"DET.NC.DIT",
                          "DET.NC.MYFRAME",
                          "DET.NC.MYPARAM1",
                          "DET.NC.MYPARAM2%f",
                          "DET.NDIT",
                          "DET.SETUPID"};
typedef struct
{
  int dit;
  int myFrame;
  int myParam1;
  float myParam2;
  int ndit;
  int setupId;
} myPARAM_TYPE;
```

Then some default parameter-values have to be set. The ***sdmaParamDefault()*** routine informs the system, that the default parameter setup has been done. Now the command interpreter is able to receive a double buffered mirror of all parameters defined in ***myPARAM_TYPE***. The first main process now looks like:

```
#include "sdma.h"
#include "sdmaTemplate.h"

static myPARAM_TYPE *param;          /* pointer to parameter structure     */

/*
 * Main
 */
int main (int argc, char *argv[])
{
  char   erms[256];
  int    nx=256;             /* default chip size              */
  int    ny=256;             /* default chip size              */

  /*
   *  startup whole dma system
   */
  if (sdmaStartup (argc, argv, (nx*ny)*2, erms) == -1)
    {
    printf ("\nsdma: %s \n\n", erms);
    sdmaExit (1);
    }

  /*
   *  setup dynamic parameters
   */
  if ((param = (myPARAM_TYPE *)sdmaSetupDynParam (sdmaNUM_ELE(paramId),
               (char **)paramId, 0, erms)) == NULL)
    {
    printf ("\nsdma: %s \n\n", erms);
    sdmaExit (1);
    }

  /*
```

```
   *  set default parameter values
   */
  param->dit = 1;
  param->setupId = 0;
  param->ndit = 10;
  param->myFrame = 1;
  param->myParam1 = 0;
  param->myParam2 = 1.5;
  sdmaParamDefault();

  /*
   *  signal, that all initialization is done
   */
  sdmaInitClient();

  /*
   *  main loop
   */

  /*
   *  cleanup and exit
   */
  sdmaExit(0);

  exit(0);
}

/*___oOo___*/
```

4.4.2 THE ACQUISITION LOOP

After the initialization phase, the acquisition process has to wait for the start command. After receiving the start command, it can do some further initialization steps and will then signal the capture process, that the main process is now ready to get data. The ***sdmaStartCapture()*** function will enter the DMA-loop. From now on all further processing has to keep up with the incoming data flow. So the acquisition process should enter the acquisition loop immediately. It waits for the next data buffer, processes it and calls the acknowledge-function ***sdmaAckData()***, when no more processing is done with the current data buffer. The main loop would look like:

```
  /*
   *  main loop
   */
  while(1)
    {
    active = 1;

    /*
     *  wait for start
     */
    if (sdmaWaitStart() == -1)
      {
      continue;
      }

    /*
     *  signal capture process, that process is ready to get data
     */
    if (sdmaStartCapture()<0)
      {
```

```
      continue;
      }

    /*
     *  acquisition loop
     */
    while (active)
      {
      /*
       *  get next data buffer
       */
      sdmaWaitData(dataIn);
      data = dataIn[0];

      /*
       *  check, if stop signal has been received
       */
      if (!(active = sdmaCheckStop()))
        {
        continue;
        }

      /*
       *  get pointer to dynamic parameter structure
       */
      param = (myPARAM_TYPE *)sdmaParamPtr();

      /*
       *  now process the data somehow
       */

      /*
       *  transfer computed data frames
       */

      /*
       *  let capture process know, that data has been processed
       */
      sdmaAckData();
      } /* end of acquisition loop */

    /*
     *  terminate processing
     */
    sdmaTermProcessing();
    sdmaFlushOutput();
    } /* end of main loop */
```

4.4.3 DATA FRAMES

A frame consists of a frame header and an output ringbuffer. With each frame a frame type (DIT, INT, ...) and a data type (int, float, ...) are associated. There are default frame-types defined within the ***sdma***-submodule:

sdmaFRAME_SNAPSHOT	snapshot frame
sdmaFRAME_DIT	DIT-frame
sdmaFRAME_INT	INT-frame
sdmaFRAME_INTERMEDIATE_DIT	intermediate DIT-frame
sdmaFRAME_INTERMEDIATE_INT	intermediate INT-frame

sdmaFRAME_SDV	standard-deviation frame
sdmaFRAME_SAMPLE	sample frame
sdmaFRAME_HCYCLE1	first half-cycle frame
sdmaFRAME_HCYCLE2	second half-cycle frame
sdmaFRAME_TRACK	vector for offset corrections

It is also possible to add new frame types to the system. The frame-type has to be a single bit value. The least significant bits are used for the pre-defined default frames. The first unused bit can be retrieved by defining ***myFRAME_TYPE*** in the following way:

```
#define myFRAME_TYPE (sdmaFRAME_USER << 1)
```

To introduce a new frame type to the system, one has also to specify a frame name and a parameter name (both ASCII-strings). If the parameter string is not empty, the specified parameter should be used to switch the generation of the frame on/off.

```
/*
 *  introduce a new frame type
 */
if (sdmaAddFrameType(myFRAME_TYPE,
                    "MYFRAME",
                    "DET.NC.MYFRAME",
                    erms) != 0)
  {
  printf ("\nsdma: cannot add frame type - %s \n\n", erms);
  sdmaExit (1);
  }
```

Each frame has to be at least double-buffered. Science-frames, which have to be stored in any case should have three ringbuffer elements (this depends on the expected computation rate of the frame and the transfer time including storage on disk). The output ringbuffer has to be allocated by the acquisition process. Then the frame has to be passed to the system:

```
  static int          *resMyFrame[2];  /* data-ring buffer of my-frame       */
  static sdmaFRAME     myFrame;        /* frame structure for my-frame       */
  int numPix = nx*ny;

  /*
   *  allocate frame buffers
   */
  for (i=0;i<2;i++)
    {
    resMyFrame[i]=(int *)malloc (numPix * sizeof(int));
    if (resMyFrame[i] == NULL)
      {
      printf ("\nsdma: allocation error \n\n");
      sdmaExit(1);
      }
    }

  /*
   *  initialize frame structures
   */
  myFrame.h.start_x=0;
  myFrame.h.start_y=0;
  myFrame.h.nx=nx;
  myFrame.h.ny=ny;
  myFrame.h.dtype=sdmaDTYPE_INT32;
  myFrame.h.ftype=myFRAME_TYPE;
  sdmaInitFrame (&myFrame, (char **)resMyFrame, 2);
```

The ringbuffer element, that has to be used for processing the frame, is assigned by the system and is always stored in the structure element ***myFrame.dptr***. So all processing has to be done on ***resMyFrame[myFrame.dptr]***. If the frame should be transferred, the output request function has to be called:

```
  /*
   *  transfer computed data frames
   */
  if (param->myFrame)
    {
    /*
     *  update some header elements
     */
    myFrame.h.setupId = param->setupId;

    /*
     *  request for parallel output of my-frame
     */
    sdmaReqOut (sdmaQUEUE_SKIP, &myFrame, 1, -1, -1);
    }
```

The setupId header element is used by the receiving process to identify frames belonging to a certain parameter setup. This is needed, as parameters may change, while the system is running.

The first parameter of the ***sdmaReqOut()*** function is used to specify the behavior, when the output ringbuffer of the frame is full. It has to be one of the following values:

sdmaQUEUE_SKIP	- skip frame, if queue is full
sdmaQUEUE_BLOCK	- block until queue is free
sdmaQUEUE_SETERR	- set error in frame header and skip

The last three parameters specify the transfer target. The first flag determines, whether the frame should be given to the data-output server for transfer via the network. The next is used to specify a camera descriptor for output on a local rtd (this is obsolete and should be set to -1). The last is used to specify a file-descriptor for output on the local disk. The file descriptor has to be retrieved by the ***sdmaInitFileServer()*** function. If the file-descriptor is set to -1, no output on the local disk is done.

Our acquisition process now looks like:

```
#include "sdma.h"
#include "myAcq.h"

static short        *data;           /* pointer to data buffer             */
static myPARAM_TYPE *param;          /* pointer to parameter structure     */

static int          *resMyFrame[2];  /* data-ring buffer of my-frame       */
static sdmaFRAME     myFrame;        /* frame structure for my-frame       */

/*
 * Main
 */
int main (int argc, char *argv[])
{
  char   erms[256];
  int    nx=256;             /* default chip size              */
  int    ny=256;             /* default chip size              */
  short *dataIn[1];          /* pointer to data buffer         */
  int    active = 0;
  int    numPix;
  int    i;
```

```
/*
 *  startup whole dma system
 */
if (sdmaStartup (argc, argv, (nx*ny)*2, erms) == -1)
  {
  printf ("\nsdma: %s \n\n", erms);
  sdmaExit (1);
  }

/*
 *  setup dynamic parameters
 */
if ((param = (myPARAM_TYPE *)sdmaSetupDynParam (sdmaNUM_ELE(paramId),
             (char **)paramId, 0, erms)) == NULL)
  {
  printf ("\nsdma: %s \n\n", erms);
  sdmaExit (1);
  }

/*
 *  introduce a new frame type
 */
if (sdmaAddFrameType(myFRAME_TYPE,
     "MYFRAME",
     "DET.NC.MYFRAME",
     erms) != 0)
  {
  printf ("\nsdma: cannot add frame type - %s \n\n", erms);
  sdmaExit (1);
  }

numPix = nx*ny;

/*
 *  allocate frame buffers
 */
for (i=0;i<2;i++)
  {
  resMyFrame[i]=(int *)malloc (numPix * sizeof(int));
  if (resMyFrame[i] == NULL)
    {
    printf ("\nsdma: allocation error \n\n");
    sdmaExit(1);
    }
  }

/*
 *  initialize frame structures
 */
myFrame.h.start_x=0;
myFrame.h.start_y=0;
myFrame.h.nx=nx;
myFrame.h.ny=ny;
myFrame.h.dtype=sdmaDTYPE_INT32;
myFrame.h.ftype=myFRAME_TYPE;
sdmaInitFrame (&myFrame, (char **)resMyFrame, 2);

/*
 *  set default parameter values
 */
param->dit = 1;
param->setupId = 0;
```

```
param->ndit = 10;
param->myFrame = 1;
param->myParam1 = 0;
param->myParam2 = 1.5;
sdmaParamDefault();

/*
 *  signal, that all initialization is done
 */
sdmaInitClient();

/*
 *  main loop
 */
while(1)
  {
  active = 1;

  /*
   *  wait for start
   */
  if (sdmaWaitStart() == -1)
    {
    continue;
    }

  /*
   *  signal capture process, that process is ready to get data
   */
  if (sdmaStartCapture()<0)
    {
    continue;
    }

  /*
   *  acquisition loop
   */
  while (active)
    {
    /*
     *  get next data buffer
     */
    sdmaWaitData(dataIn);
    data = dataIn[0];

    /*
     *  check, if stop signal has been received
     */
    if (!(active = sdmaCheckStop()))
      {
      continue;
      }

    /*
     *  get pointer to dynamic parameter structure
     */
    param = (myPARAM_TYPE *)sdmaParamPtr();

    /*
     *  now process the data somehow
     */
    /* compute resMyFrame[myFrame.dptr]... */
```

```
      /*
       *  transfer computed data frames
       */
      if (param->myFrame)
        {
        /*
         *  update some header elements
         */
        myFrame.h.setupId = param->setupId;

        /*
         *  request for parallel output of my-frame
         */
        sdmaReqOut (sdmaQUEUE_SKIP, &myFrame, 1, -1, -1);
        }

      /*
       *  let capture process know, that data has been processed
       */
      sdmaAckData();
      } /* end of acquisition loop */

    /*
     *  terminate processing
     */
    sdmaTermProcessing();
    sdmaFlushOutput();
    } /* end of main loop */

  /*
   *  cleanup and exit
   */
  sdmaExit(0);

  exit(0);
}

/*___oOo___*/
```

4.4.4 PARALLEL PROCESSING

The processing algorithm is application dependent. All calculation should be done using parallel threads, to use the full computing bandwidth of a multi-processor CPU. The ***sdma***-library contains functions for starting and synchronizing calculation threads:

```
/*
 * Calculation process
 */
static void calcThread(void *arg)
{
  int     id = (int)arg;              /* process id                  */
  int     n = sdmaProcNum(id);        /* process number              */
  int     size = numPix/numProc;      /* number of pixels per process  */
  short  *dataLoc;
  int    *pMyFrame;
  int     i;

  /*
   *  wait for start
```

```
   */
  sdmaProcWaitStart(id);
  while(1)
    {
    sdmaProcWaitTrigger(id);

    /*
     *  calculate frame(s)
     */
    dataLoc = data+(n*size);
    pMyFrame = resMyFrame[myFrame.dptr]+(n*size);

    /*
     *  just fill buffer(s) with some data
     */
    for (i=0;i<size;i++)
      {
      pMyFrame[i] = i;
      }

     sdmaProcAck(id);
    }
}

int main (int argc, char *argv[])
{
  int numProc = 4; /* should be number of CPUs */

  /*
   *  initialize
   */

  /*
   *  create calculation threads
   */
  if ((setId = sdmaProcCreate ((void *)&calcThread, numProc, erms)) < 0)
    {
    printf ("\nsdma: error: %s\n\n",erms);
    sdmaExit (1);
    }

  /*
   *  data acquisition
   */
  while ...
    {

    /*
     *  now process the data in parallel
     */
    sdmaProcTrigger(setId);
    sdmaProcWaitAck(setId);

    }
}
```

4.4.5 USER DEFINED COMMANDS

It is also possible to add user defined command handling to the acquisition process. A command and a reply buffer are passed unchecked via ***sdmaExecUsrCmd()*** to an external command handling routine. This routine is set by the acquisition process and can vary between the readout modes. To implement such a command handler one has to set the global function ***sdmaUsrCmdHandler*** to a command handling routine during the initialization phase:

```
static void myCommandHandler (char *command, char *reply)
{
  /*
   *  parse command string (format is unspecified)
   */

  if (strcmp(command, "MYCMD") == 0)
    {
    /*
     *  all synchronization with the main thread
     *  has to be done here
     */
    sprintf(reply, "my reply");
    }
  else if (strcmp(command, "NOP") == 0)
    {
    sprintf(reply, "NOP done");
    }
  else
    {
    sprintf(reply, "unknown command");
    }
  /* check command string */

  /* handle command */

  /* set reply string */
}

int main (int argc, char *argv[])
{

  /*
   *  initialize
   */

  sdmaUsrCmdHandler = myCommandHandler;

  /*
   *  data acquisition
   */
}
```

The ***sdmaUsrCmdHandler*** function defaults to NULL (no external handler). The ***sdmaExecUsrCmd()*** function can pass any command/reply string to the external handler. The string length is limited to **sdmaMAX_USRCMD_LEN** characters.

4.4.6 ARRAY SORTING

The structure of the detector arrays differs a lot. Ordering can be in stripes, quadrants, quadrants and stripes, from outside to inside or vice versa. The sorting and processing of the arrays might make the parallel computation algorithms quite complex. For very large arrays, which do no more fit into the cache, the sorting might additionally slow down the performance of the real-time loop. If applicable the frames can be computed as raw unsorted buffers, which are refromatted just before they are transferred. For this purpose a pixel map (***sdmaPixMap***) has to be defined. The ***sdmaPixMap*** is declared as ***NULL*** pointer by default. It is only applied if it points to a valid map. The application process should allocate an integer buffer with appropriate size and fill it with a sorting table. Then the ***sdmaPixMap*** must be set to the defined map:

```
sdmaPixMap = myMap;
```

The map will apply the following sorting:

```
out[i] = data[myMap[i]];
```

The output task also takes care of window requests. If a special frame should be transferred unsorted although a pixel map has been defined, then the ***noSort*** flag must be set in the frame structure when passing it to the output queue:

```
...
myFrame.h.setupId = param->setupId;
myFrame.noSort = 1;
sdmaReqOut (sdmaQUEUE_SKIP, &myFrame, 1, -1, -1);
...
```

4.4.7 DOWNLOADING OF DATA-SETS

The processing task has the possibility to reserve memory that can be accessed from remote via the command interpreter. This memory area(s) may contain flatfield(s), badpixel-mask(s) or other data sets used for the pre-processing of image data. The buffer can be set from the host process by passing *dataSetId* and *size* to the command interpreter with ***sdmaDownloadBuf()***. A blocksize for the transfer has also to be specified. If this blocksize exceeds the maximum command-buffersize, the command interpreter would return a socket (server address and port number) in the reply to the host process. Afterwards it waits for ***size*** bytes to be received on this socket. As the reserved memory is double buffered, the readout-mode process would not block while a download is in progress. If all buffers are set, a sync command ***sdmaMsync()*** has to be sent to the command interpreter to perform the memory swap. The memory areas can be placed in shared memory to be shared between different readout modes.

```
#define myDATA_SET_ID 4  /* has to be a single bit value */
char *myDataSet;

sdmaShDataAttch(0, erms);
sdmaShDataGetPtr ((char **)&myDataSet, myDATA_SET_ID, size, erms);
while (active)
  {
  sdmaWaitStart();
  sdmaStartCapture();
  while (started)
    {
    sdmaWaitData (&dataIn);
    /* process data frame: frame = f(dataIn, myDataSet) */
    sdmaAckData();
    }
  sdmaTermProcessing();
  }
```

4.4.8 COUNTER RESET AND EXPOSURE END FLAG

There are two global flags for exposure control. They are set asynchronously via the command interpreter task and can be checked at run-time within the acquisition loop. After checking both have to be reset by the application.

The counter reset flag (***sdmaResetCnt***) is set, when the exposure should (re-)start while the sequencer is left running. Typically at this point the observing conditions have changed (telescope move etc.). This means, that the DMA-input buffer has to be flushed ***sdmaFlushInput()*** and the current integration has to be skipped. This has to be taken into account when estimating the exposure time. In the worst case one integration is lost.

The exposure end flag (***sdmaEndFlg***) is set, when the exposure should end immediately. Typically now an intermediate result should be transferred.

4.4.9 REAL-TIME PERFORMANCE

The acquisition tasks for pure imaging do not require hard real-time performance as it is offered by real-time operating systems (such as VxWorks, RT-Linux, LYNXOS etc.). However - to assign full CPU computing capacity to the pre-processing algorithm, a priority based scheduling mechanism is required. Thes feature is available both on Solaris and Linux operating systems. To get priority based scheduling, super-user privileges must be given to the acquisition process:

> *chown root* <*process name*>
> *chmod a+s* <*process name*>

The priorities for the command interpreter task, for the capture task and for the data-output task are set internally. The processing task has to set its own priority during the initialization phase:

> *sdmaSetPrio (sdmaPRI_PROC, sdmaPRI_RT, sdmaTQ_INF);*
> *sdmaSetPrio (sdmaPRI_PROC, sdmaPRI_TS, sdmaTQ_DEF);*

The sdmaPRI_PROC constant defines the maximum priority, that can be used by the processing task. When sdmaPRI_RT is used, the process runs as real-time class. ***sdmaPRI_TS*** lets the process run as time-sharing class. Either an infinite (***sdmaTQ_INF***) or default (***sdmaTQ_DEF***) time quantum can be used. It is also possible to specify the time quantum in microseconds. The time quantum value is the maximum amount of time a process may run assuming that it does not complete or enter a resource or event wait state (sleep). Note that if another process becomes runnable at a higher priority, the currently running process may be preempted before receiving its full time quantum. Each thread can set its own priority via the ***sdmaSetPrio()*** function.

4.5 COMMAND INTERFACE

The command interpreter task has the highest real-time priority (to be able to send commands at any time). It handles the commands and synchronizes them with the capture task, the data-output task and the main processing task. All parameters defined by the processing task (plug-in) are double-buffered and can be set also during run-time. The parameter values are overtaken synchronously with a *sync*-command.

4.6 DATA INTERFACE

The data frames can be requested in parallel from the output link processes. The data transfer is done via socket (tcp). The ***sdmaOpenDataLine()*** routine distinguishes between video- and science-data transfer. In science mode the oldest available, but not yet transferred frame, which matches the frame-type in the request structure is transferred (FIFO). In video-mode for each frame-type matching the request, the latest not yet transferred frame is selected (LIFO). From this selection the oldest frame is transferred. In both cases *'not yet transferred'* means, that the frame has not yet been transferred to the requesting client. In video-mode additionally the retransmit flag in the request-structure is checked, to allow frames of the specified type to be transmitted again. This can be used, to display different windows of the same frame during long integrations.

Format of the request structure:

```
typedef struct
{
  int      type;           /* requested frame type              */
  int      blocking;       /* block until frame is ready        */
  int      retransmit;     /* transfer last buffer again        */
  int      start_x;        /* window start-x                    */
  int      start_y;        /* window start-y                    */
  int      nx;             /* window nx                         */
  int      ny;             /* window ny                         */
} sdmaREQ;
```

Format of the returned frame structure:

```
typedef struct
{
  int   dtype;             /* data type              */
  int   ftype;             /* frame type             */
  int   start_x;           /* window start-x         */
  int   start_y;           /* window start-y         */
  int   nx;                /* window nx              */
  int   ny;                /* window ny              */
  int   scal;              /* scale factor           */
  int   cnt;               /* frame counter          */
  int   setupId;           /* setup-id               */
  int   err;               /* error-id               */
  int   overrun;           /* overrun flag           */
  int   frames;            /* available frame types  */
  int   tx;                /* track point x          */
  int   ty;                /* track point y          */
} sdmaFRAME_H;

typedef struct
{
  sdmaFRAME_H  h;          /* data header            */
  char *       fbuf;       /* data buffer            */
} sdmaFRAME_T;
```

The sequence for the function calls should be the following:

- ***sdmaPingDataLine()*** (optional);
- ***sdmaOpenDataLine(serverName, &socket, ...)***;
- ***sdmaSendDataRequest(socket, request, ...)***;
- ***sdmaWaitDataReply(socket, &frameHeader...)***;
- check frame header;
- if frame type is ***sdmaFRAME_NO_FRAME***, then send a new data request. Otherwise do either transfer the frame (window parameters in request can be updated according to the frame type) with sdmaAcceptFrame(socket, request, &frame, ...) or skip it with ***sdmaSkipFrame(socket, ...)***;
- the frame name can be resolved with the ***sdmaGetFrameNameByType()*** function (optional);
- the frame can be scaled using the general ***sdmaScale(&frame)*** function (optional);
- continue with ***sdmaSendDataRequest(socket, request)***;

The socket can be used for a select() call before sdmaWaitDataReply. To cancel a request one has to call ***sdmaCancelReq(socket, ...)*** before ***sdmaWaitDataReply()*** and then send a new request with ***sdmaSendDataRequest()***.

If any of these functions fails, one should call ***sdmaCloseDataLine()*** and then try to reopen with ***sdmaOpenDataLine()*** to recover the data-channel from failure.

The calling program should not exit with an open request. When the process is waiting for the data reply, ***sdmaAbortReq(serverName, pid, ...)*** or ***sdmaCancelReq(spcket, ...)*** have to be called before exiting. It is recommended to add ***sdmaAbortReq()*** or ***sdmaCancelReq()*** to a signal handler.

The ***sdmaAcceptFrameN(socket, request, &frame, partNo, numDiv, ...)*** function can be used instead of ***sdmaAcceptData()*** to request a frame partition. The additional parameters **<*partNo*>** and **<**numDiv**>** have to be supplied to indicate that the partition number **<*partNo*>** out of **<*numDiv*>** sub-divisions of the frame has to be taken as base for the requested window.

The data reception task ***sdmaDart*** can be used to test the data transfer. It uses the above routines and can be taken as a template for other customized data reception tasks:

```
Usage: sdmaDart [options]

options:
  -v                 - verbose mode
  -disp              - switch display on
  -noblock           - non-blocking mode
  -size <squarePix>  - set max. squarepix. size for display
                       (default: 1024)
  -camera <name>     - set camera name (this enables
                       the [-disp] option also)
                       (default: cam)
  -host <ip-addr>    - set ip-address of data server
                       (default: ins8)
  -port <portNum>    - set port number
                       (default: 8012)
  -type <type>       - set frame type (number)
                       (default: -1)
  -video             - switch to video transfer mode
                       (default: science transfer)
  -win <sx sy nx ny> - set window
                       (default: 1 1 0 0)
  -h                 - show this
```

5 IRACE INTERFACE LIBRARIES

The irace module contains the interface libraries for both the IRACE front-end controller (***tcom*** - submodule) and the IRACE pre-processor (***sdma*** - submodule).

5.1 TCOM INTERFACE LIBRARY

The following functions are provided for the client as part of the ***tcomSclInt*** interface library:

5.1.1 SEQUENCER FUNCTIONS

tcomSclBinToInt()	converts from binary "01011..." (32 bit) format to integer format.
tcomSclBinToState()	converts from binary "01011..." (64 bit) format to subpattern state.
tcomSclClkpLoop()	downloads a subpattern dispatcher loop command line to the sequencer.
tcomSclClkpProg()	downloads a binary clock pattern generating process to the sequencer
tcomSclClrPat()	clears all subpatterns in sequencer.
tcomSclClrClkp()	clears all clock patterns in sequencer.
tcomSclGetClkp()	reads a clock pattern from the sequencer.
tcomSclGetPat()	reads a subpattern from the sequencer.
tcomSclGetSpeed()	gets the sequencer readspeed.
tcomSclIntToBin()	converts the bitfields of an integer value to binary "01011..." (32 bit) representation.
tcomSclSeqProg()	downloads a sequencer readout loop command line to the sequencer.
tcomSclSeqStart()	starts the sequencer.
tcomSclSeqStop()	stops the sequencer.
tcomSclSetClkp()	downloads a clock pattern to the sequencer.
tcomSclSetMode()	set sequencer mode (normal/chopping).
tcomSclSetPat()	sends a subpattern to the sequencer.
tcomSclSetSpeed()	sets the sequencer readspeed.
tcomSclStatSeq()	reads the sequencer status.
tcomSclSubpatSetup()	handles sequencer subpattern setup file.
tcomSclTime()	gets sequencer readout times.

5.1.2 CLDC FUNCTIONS

tcomSclDacDisable()	disables the DAC output.
tcomSclDacEnable()	enables the DAC output.
tcomSclNumCldc()	returns number of cldc-boards.
tcomSclSetVolt()	sets the voltage on a single dac channel.
tcomSclSetVoltAll()	sets the voltage on all dac channels.
tcomSclStatCldc()	reads the cldc status.
tcomSclTelChan()	gets the voltage from a single telemetry channel.
tcomSclTelemetry()	reads all telemetry channels.

5.1.3 STATUS BUS FUNCTIONS

tcomSclReadHwStat()	reads from status bus.
tcomSclWriteHwStat()	writes to status bus.

tcomSclErrStat() gets hardware error status from status bus.
tcomSclSyncStat() synchronizes with status bus modules to have special bits set (like GIGALINK-SYNC etc.).

5.1.4 GENERAL STATUS FUNCTIONS

tcomSclErr() maps errorId to error message.
tcomSclGetConf() gets the board configuration.
tcomSclNumCldc() gets the number of installed cldc -boards.
tcomSclPstat() reads the parameter status.

5.1.5 SYSTEM SERVICES FUNCTIONS

tcomSclAbort() aborts all (dac disable, stop, close).
tcomSclBoot() boot the scl network.
tcomSclCloseComLine() closes a connection to the command interpreter.
tcomSclConnect() connects to the command interpreter.
tcomSclExec() executes server or simulator process from remote
tcomSclInitCom() initializes the command structures.
tcomSclKill() kills server or simulator process started with tcomSclExec.
tcomSclOpenComLine() opens a connection to the command interpreter.
tcomSclPing() checks, if the command interpreter is alive.
tcomSclReset() resets the scl network.
tcomSclSrvAbor() abort protocol converter server.
tcomSclSrvStat() read protocol converter server status.
tcomSclTimeout() sets the link timeout.
tcomSclWaitInit() waits until the clients are initialized.

5.1.6 INTERACTIVE MODE FUNCTIONS

tcomSclFrameSetup() handles sequencer frame sequence setup-file.
tcomSclHelp() print on-line help text for interactive mode.
tcomSclInteractive() interactive mode main loop
tcomSclPrintFrameSetup() prints sequencer frame sequence.
tcomSclPrintSubpatSetup() prints sequencer subpattern setup.
tcomSclPrintVoltSetup() prints CLDC voltage setup.
tcomSclSaveVoltSetup() saves CLDC voltage setup.
tcomSclVoltSetup() handled CLDC voltage setup file.

5.2 SDMA INTERFACE LIBRARY

The ***sdma*** submodule contains two libraries. The ***sdma***-library contains all functions needed by the acquisition process. The sdmaInt-library contains all functions for the command- and data-interface.

5.2.1 ACQUISITION

The following functions are provided for the processing task as part of the ***sdma*** library:

sdmaAckData() acknowledges, that the data buffer has been processed.

sdmaCheckStop()	checks, if the stop command has been received since the last call.
sdmaFlushInput()	acknowledges the current input buffer and skips all buffers in the input ringbuffer.
sdmaFlushOutput()	flushes the output queue.
sdmaGetPrio()	get process scheduling and priority.
sdmaInitFrame()	register frame for parallel output.
sdmaParamPtr()	returns a pointer to the current parameter structure.
sdmaParamDefault()	signals, that the default parameter setting has been done.
sdmaReqOut()	requests a queued output of a processed frame to the output link and/or to the on-line display.
sdmaSetPrio()	set process scheduling and priority (requires super user privileges).
sdmaSetupDynParam()	registers a dynamic parameter set.
sdmaShDataAttach()	tries to get access to a (shared) memory data structure.
sdmaShDataGetPtr()	returns a pointer to a (shared) memory data entry identified by an <id>. If the entry for <id> is not yet existing, a new entry is created.
sdmaShDataValid()	mark a dataset as containing valid data.
sdmaShDataInvalid()	mark a dataset as not containing valid data.
sdmaStartCapture()	starts the data capture.
sdmaStartup()	performs start-up of all capture and output tasks.
sdmaTerm()	terminates the asynchronous DMA and clears the on-board fifos.
sdmaTermProcessing()	terminates all processing (resets buffer queues etc.).
sdmaWaitData()	waits for next input data buffer and returns a pointer.
sdmaWaitStart()	waits for start command.

5.2.2 COMMAND INTERFACE

The following functions are provided for the command client as part of the ***sdmaInt*** library:

sdmaAbort()	aborts all.
sdmaCleanup()	cleanup routine to be called after abnormal termination of the acquisition process (POSIX version only).
sdmaCloseComLine()	closes a connection to the command server.
sdmaComExec()	starts a processing task.
sdmaConnect()	reopen connection to command interpreter.
sdmaDownloadBuf()	downloads a buffer as data set to the reserved memory of the processing task.
sdmaEnd()	forces sets a flag for the acquisition process to transfer an intermediate result as measurement frame.
sdmaExecUsrCmd()	execute an external user defined command.
sdmaExitComServer()	exits the command interpreter.
sdmaGetConfig()	receives the current hardware/software configuration.
sdmaGetErrPort()	gets the port number of the error-stack server.
sdmaGetParam()	gets a parameter value by name.
sdmaGetParamBuf()	gets the parameter buffer.
sdmaGetParamSetup()	receives the parameter setup.
sdmaGetStatus()	receives the data capture status.
sdmaGetTime()	receives the calculation and input loop times.
sdmaKill()	kills the processing task.
sdmaLoadBuf()	loads a buffer from file.
sdmaMclose()	remove shared memory data set area.
sdmaMsync()	swap shared memory datasets.
sdmaOpenComLine()	opens a connection to the command server.

sdmaParamSync()	performs parameter buffer swap.
sdmaPing()	checks is the command interpreter is alive.
sdmaResetCnt()	sets a flag for the acquisition process to reset the sum counters.
sdmaSendDynCmd()	sends a dynamic command.
sdmaSetHdr()	sets the DMA-interface device header on the GIGA-bus.
sdmaSetParam()	sets a parameter value by name.
sdmaSetParamBuf()	downloads the parameter buffer.
sdmaSetSimTime()	sets the simulation interval (in ms) of the built-in simulator.
sdmaStart()	starts the data acquisition.
sdmaStop()	stops the data acquisition.
sdmaTimeout()	sets the timeout for the command reply.
sdmaVerbose()	switch verbose mode on/off.

5.2.3 DATA INTERFACE

The following functions are provided for the data client as part of the ***sdmaInt*** library:

sdmaAbortReq()	aborts the current frame request.
sdmaAcceptFrame{}	accepts the frame returned by sdmaWaitNextFrame. The frame is transferred and read into a buffer supplied by <frame>.
sdmaAcceptFrameN{}	accepts the frame returned by sdmaWaitNextFrame. The frame is transferred and read into a buffer supplied by <frame>. The additional parameters <partNo> and <numDiv> have to be supplied to indicate that the partition number <partNo> out of <numDiv> sub-divisions of the frame has to be taken as base for the requested window.
sdmaCancelReq()	aborts the current frame request, waits for the data reply and skips an eventually incoming frame.
sdmaCloseDataLine()	closes a data connection to an output link process.
sdmaGetFrameNameByType()	resolve frame name by specifying a type.
sdmaGetFrameTypeByName()	resolve frame type by specifying a name.
sdmaGetFrameTypes()	get the frame types, which are produced by the current readout mode.
sdmaOpenDataLine()	opens a data connection to an output link process.
sdmaPingDataLine()	checks if the data server is alive.
sdmaScale()	scale frame according to the scale factor specified in the frame header.
sdmaSendDataRequest()	sends a frame request structure to the data output server.
sdmaSkipFrame()	skips the frame offered by sdmaWaitDataReply().
sdmaWaitDataReply()	waits for a reply from the data output server.

6 INFRARED DCS

6.1 INFRARED ACQUISITION LIBRARY

The ***iracq***-library contains a basic set of ANSI-C functions to control both the IRACE front end and the acquisition process. When using the server class libraries, these routines are fully transparent.

6.2 CONTROL SERVER BASE CLASS

On top of the infrared acquisition library a basic C++ server class (***iracqSRV***) has been built to provide control server functions, which are also compliant to VLT-NOCCS installations on the IRACE workstation. This is intended to be used as lab-interface to IRACE. The NOCCS acquisition server (***iracqServerNoCss***), that uses an instance of this class, can be operated under some restrictions (no logging, no database, no templates...) via the same graphical user interface as the higher level ***iracqServer*** (see section 6.9).

6.3 EVENT HANDLER CLASS

The infrared acquisition event handler class (***iracqEVH***) is derived from the control server base class (***iracqSRV***). Using the CCS/ECCS functionalities this class provides the high level interface to the infrared data acquisition. Application specific servers can be built by deriving new classes from ***iracqEVH*** (see section 6.6).

6.4 INFRARED ACQUISITION CONTROL SERVER

The infrared acquisition control server (***iracqServer***) is the interface process to the IRACE-DCS for the external world. Through this server all commands must pass. The server checks the validity of various commands and parameters according to the current state of DCS.

6.4.1 SEVER PROCESS

```
Usage: iracqServer [options]

Options:
   -v <level>          - verbose level
                         (default: level = 0)
   -log <level>        - log level
                         (default: level = 0)
   -xterm              - start processes in xterms
   -sim                - start server in simulation mode
   -test               - just do initialization test
   -dtt <name>         - process name of data transfer task
                         (default: name = iracqDtt)
   -port <port number> - server port number (NOCCS only)
                         (default: port = 8019)
   -cfg <name>         - system configuration file name
                         (default: name = sys.cfg)
   -det <name>         - detector configuration file name
   -inst <instance>    - server instance
                         (default: instance = <>)
   -h                  - show this
```

The ***<-inst>*** option allows to start several control servers within the same environment. This is necessary to interface to more than one IRACE controllers at the same time. The name to register with CCS will be '***iracqServer_<instance>***'. The database point for the iracq-module is then supposed to be '***<alias>iracq_<instance>**:*'. Without the <-inst> option defined the server will register under his actual process name. The database point is then supposed to be '***<alias>iracq:***'.

6.4.2 DATABASE

The file iracqBranch.db contains the class definition for the Infrared Acquisition Module. This file has to be included in the ***DATABASE.db*** file of the environment. The following macros can be defined before each inclusion:

> #define iracqINSTANCE iracq_myInstance
> #define iracqROOT myRoot
> #include "iracqBranch.db"

iracqINSTANCE becomes the alias of the database point for this instance. If not defined it defaults to "***iracq***". ***iracqROOT*** is the absolute path of the database root. If not defined it defaults to ***<iracqINSTANCE>***.

The basic structure of the infrared acquisition database is as follows:

```
--o <alias>iracqINSTANCE --|--o irace      (irace system parameters)
                           |--o exposure   (exposure parameters)
                           |--o detector   (detector specific values)
                           |--o rtd        (RTD interface)
```

6.4.3 SERVER STATES

The server state is called "***OFF***" when no server process is running. Starting the server initializes the state to "***LOADED***". Then it is possible to send commands. If a default detector configuration was specified in the system configuration or the -det option was set, the detector configuration is loaded. The command "***STANDBY***" brings the server to the "***STANDBY***"-state. All sub-systems are up, but not yet configured. The IRACE command server (protocol converter) is running on the IRACE pre-processor workstation. The IRACE front-end is booted. The acquisition process is not yet running. The command "***ONLINE***" configures all sub-systems with the current detector configuration and also starts the acquisition process ("***ONLINE***"-state). It is possible to go directly from "***LOADED***"- to "***ONLINE***"-state and vice-versa (with the command "**OFF**"). When going from "***ONLINE***" to "***STANDBY***" the acquisition process terminates, the sequencer is stopped, the CLDC-boards are all disabled and the control server disconnects from IRACE command server and front-end. However the IRACE front-end remains in booted state unless the -haltIrace flag was set in the "***STANDBY***" command (i.e. "***STANDBY -haltIrace***"). The "***OFF***"-command will terminate all sub-processes and the IRACE front-end is no longer booted.

The data acquisition is started/stopped with the "***SEQ -start/-stop***" command. The command "***MISC -rstcnt***" sets a flag on the acquisition process to reset all loop counters.

If the ***DET.CON.AUTONLIN*** parameter is set to "***T***" in the system configuration, the server will automatically go to "***ONLINE***"-state after being started.

The IRACE front-end can be reset using the ***RESET***-command. This is only valid in "***ONLINE***"- or

"***STANDBY***"-state. Otherwise the command has no effect. If the state was "***ONLINE***" the server goes automatically to "***STANDBY***"-state afterwards.

Server States:

```
iracqSTATE_OFF       (1) - "OFF"
iracqSTATE_LOADED    (3) - "LOADED"
iracqSTATE_STANDBY   (3) - "STANDBY"
iracqSTATE_ONLINE    (4) - "ONLINE"
```

Server Sub-States:

```
iracqSUBSTATE_IDLE   (1) - "IDLE"
iracqSUBSTATE_BUSY   (2) - "BUSY"
iracqSUBSTATE_ERROR  (3) - "ERROR"
```

Operational Modes:

```
iracqMODE_REAL       (1) - "NORMAL"
racqMODE_SIM         (2) - "SIMULATION"
```

Database Attributes:

```
<alias>iracq:irace.state     - current server state
<alias>iracq:irace.stateN    - server state name
<alias>iracq:irace.subState  - current server sub-state
<alias>iracq:irace.subStateN - server sub-state name
<alias>iracq:irace.opMode    - operational mode
<alias>iracq:irace.opModeN   - name of operational mode
```

6.4.4 INTEGRATION TIME

The integration time is set via the setup parameter ***DET.DIT***. There is a minimum value (***DET.INDIT***) for the integration time, which depends on the currently used read-out technique. The actual value of ***DET.MINDIT*** is defined in the Sequencer-Program file. If a the value of less than ***DET.MINDIT*** is specified, ***DET.DIT*** is automatically corrected.

Database Attributes:

```
- DET.DIT          <alias>iracq:exposure.DIT
- DET.MINDIT       <alias>iracq:exposure.MINDIT
```

Access:

```
- DET.DIT          SETUP  "-function DET.DIT <value in seconds>"
                   STATUS "-function DET.DIT"
- DET.MINDIT       STATUS "-function DET.MINDIT"
```

6.4.5 SETUP COMMAND

The syntax of the setup command is:

"***SETUP -function <param1Name> <param1Value> <param2Name> <param2Value> ...***"

All setup keywords specified in the ESO-VLT-DIC.IRD dictionary can be set using this command.

The dynamic parameters of the acquisition process can also be set via the setup command. The database attribute '***<alias>iracq:irace.dynPars***' contains a table [*name (rtBYTES32), value (rtBYTES32)*] of all currently defined dynamic parameters.

The parameter setup contains all dynamic parameters for a read-out mode. These can be parameters exported by the acquisition process or parameters used in the sequencer program or subpattern dispatcher files. All parameters which are not exported from the acquisition process have to be defined in the DSUP-file of the read-out mode. The acquisition process defines valid default values for all its exported parameters (these parameters can optionally be set in the DSUP-file)

6.4.6 READ-OUT MODES

The read-out modes for a detector are defined in the detector configuration file. Each read-out mode is assigned a unique name, which is used for selection via the control panel or the setup command "***SETUP -function DET.NCORRS.NAME <name>***". The setup parameter ***DET.NCORRS*** refers to the unique id (RM-index) of the mode and can be used alternatively.

Example:

```
DET.IRACE.RM2.NAME      "Double"; # read-mode name
DET.IRACE.RM2.ACQ       "sdmaAl2"; # acquisition process
DET.IRACE.RM2.SEQ       "AladdinDblCor_seq"; # readout sequence
DET.IRACE.RM2.DSUP      "AladdinDblCor.dsup"; # default parameters
```

The available read-out modes for the current detector setup can be retrieved with the command "***STATUS -function DET.NCORRS.AVAIL***". The format of the returned list is:

<NCORRS>:<read-out mode name> | ...

The currently defined read-out modes are also stored as table (fields: *name*, *ncorr*) in the database attribute '***<alias>iracq:irace.rmDef***'.

6.4.7 FRAME WINDOW HANDLING

The setup parameters ***DET.WIN.STARTX, DET.WIN.STARTY, DET.WIN.NX, DET.WIN.NY*** define the format and position of the data-frame within the chip. STARTX/Y refers to the lower left corner. If software windowing is used (default, "***SETUP -function DET.WIN.TYPE 0***"), the whole detector is read out and only the data transfer task is informed to request a window from the pre-processor. This is mainly used to save transfer/storage overheads.

If hardware windowing is selected ("***SETUP -function DET.WIN.TYPE 1***") the sequencer program is reloaded with the new window parameters and the acquisition process is restarted with the '-nx, -ny' command line options set accordingly. The ***DET.WIN.HW.SUP*** setup parameter can be used to disable hardware windowing. Generally this should be set in the default parameter setup file for read-out modes which do not support windowed read-out of the detector. If hardware windowing is switched on, the window setup parameters are automatically adjusted following several detector specific rules which are defined in the detector configuration:

The ***DET.CHIP.ADJUST*** parameter specifies the HW-window adjustment mode for the detector.

Valid values are:

```
CENTER   - window is automatically centered
PART     - window is adjusted to the next partition
           specified by the DET.CHIP.PARTNX,PARTNY parameters
FREE     - window is only adjusted to multiples of
           the DET.CHIP.STEPX,STEPY parameters
```

The ***DET.CHIP.STEPX/Y*** parameters specify the adjustment step in x/y-direction for windowed readout. Adjustments are done in multiples of this value.

The ***DET.CHIP.PARTX/Y*** parameters specify the partition size in x/y-direction for windowed readout. Only relevant, if the ***DET.CHIP.ADJUST*** value is set to "*PART*".

6.4.8 EXPOSURE CONTROL

Normally, when an exposure is started both sequencer and acquisition are restarted. It is also possible to let the sequencer run continuously, when a ***START***-command is issued. This is controlled via the ***SETUP*** keyword "***DET.IRACE.SEQCONT T|F***". In continuous mode just the acquisition process resets its buffers and starts building a new sum. This is useful to avoid that corrupted data is used for building the sum, like if for instance the telescope was moved and frames where taken during the movement. It is however possible to specify that the acquisition process should not reset the buffers/counters by using the flag "***-noNcReset***" together with the ***START***-command. In this case the acquisition process continues to build the sum and the frame may be ready earlier than the actual exposure time. This feature can be used when it is important to minimize the overheads, but requires that there are no changes in the fields of view.

The exposure can be aborted using the ***ABORT***-command. In this case no data file is generated unless the frame was already received on the WS as the command was issued.

The ***END***-command makes the acquisition process terminate the exposure as soon as possible. In this case the data file generated may be just an intermediate result.

The ***WAIT***-command can be used to wait for an exposure to complete. It returns the actual exposure state with the last reply.

The FITS-header is generated as specified in the ***ESO-VLT-DIC.IRACE*** dictionary. With the setup command "***SETUP -function DET.FITSHDR.EXT.ST T|F***" the extended FITS-header can be enabled or disabled. When the extended FITS-header is enabled also maintenance keywords like clock- and bias-voltages will be stored.

Exposure States:

```
iracqEXP_UNDEFINED           (0)          UNDEFINED
iracqEXP_INACTIVE            (1)          IDLE
iracqEXP_PENDING             (2)          'not implemented'
iracqEXP_INTEGRATING         (4)          INTEGRATING
iracqEXP_PAUSED              (8)          'not implemented'
iracqEXP_READING_OUT        (16)          'not implemented'
iracqEXP_PROCESSING         (32)          'not implemented'
iracqEXP_TRANSGFERRING      (64)          TRANSFERRING
iracqEXP_COMPL_SUCCESS     (128)          COMPLETED
iracqEXP_COMPL_FAILURE     (256)          FAILURE
iracqEXP_COMPL_ABORTED     (512)          ABORTED
```

Database attributes:

```
<alias>iracq:exposure.expStatus      -exposure status
<alias>iracq:exposure.expStatusN     -exposure status name
<alias>iracq:exposure.expTime        -exposure time
<alias>iracq:exposure.expCountDown   -exposure countdown
<alias>iracq:exposure.expNo          -exposure number
```

After starting the exposure state will be "*INTEGRATING*". When the header of the last file has been received by the data transfer task (i.e. the break-conditions for all frames have been reached) the exposure state goes to "*TRANSFERRING*". At this time all detector-data for the current exposure are taken. When the last file has been stored on disk the exposure state goes to "*COMPLETED*". If an error occurred during the exposure the state goes to "*FAILURE*". If the exposure was aborted the state goes to "*ABORTED*".

6.4.9 DETECTOR MODES

Setup macros for operating the detector in several pre-defined modes can be declared in the detector configuration file:

```
DET.MODEi.NAME       "name of the mode";
DET.MODEi.SETUP      "<keyword> <value>, <keyword> <value>, ...";
     OR
DET.MODEi.SETUP      "setup-file";
```

The mode can be selected with the command:

"***SETUP -function DET.MODE.NAME <name>***"

This will set all keywords given in the ***DET.MODE.SETUP*** of that mode. If the current setup matches all keywords of a mode, the ***DET.MODE.NAME*** keyword reflects the according mode-name and is also set in the FITS-header. If several modes match, the one with the most keywords is taken. The maximum number of modes for one detector configuration is 16. Each mode can contain up to 64 keywords. If the setup line length exceeds 256 characters, a file-name can be specified instead. The setup file must contain the keywords for the mode in SHORT-FITS format. Unless an absolute pathname is given, the file is searched in

```
$INS_ROOT/$INS_USER/MISC/IRACE/MSUP/
```

If no extension is given in the file-name, the extension "***.msup***" is assumed.

6.4.10 FRAME SELECTION

Usually an exposure is finished, when the INT-frame has been received on the WS. As it is required by some readout modes to store also other frames during one exposure, a more general exposure break condition has to be applied. Each frame generated by the acquisition process and selected to be stored can have a counter, that indicates the number of frames of that type, that must be stored during the exposure. The exposure is finished, when all these frames have reached their break condition. All that can be controlled via the command:

"***FRAME -name <frame name> [-gen T|F] [-store T|F] [-break <counter>]***"

A list of all available frames can be retrieved with the command:

*"**STATUS -function DET.NCORRS.FRAMES**"*

The format of the returned list is:

```
<frame name>: <gen (0|1)> <store (0|1)> <break counter>|...
```

The available frames are also stored as a table (fields: *name*, *generate*, *store*, *disp*, *break*) in the data base attribute '***<alias>iracq:irace.frames***'.

6.4.11 NAMING OF DATA FILES

There are three different naming-schemes for files produced during an exposure. Unless an absolute path name is specified as base-name for one of the below naming-schemes all files will be stored by default in the data-path ***$INS_ROOT/$INS_USER/DETDATA***. In addition the data-path can be defined via the ***DET.IRACE.DTT.PATH*** parameter which can also be set in the system-configuration file.

The naming scheme can be adjusted:

- via the system configuration keyword "***DET.CON.NAMING.TPYE <type>***"
- via the command "***SETUP -function DET.EXP.NAMING.TYPE <type>***"

The ***<type>*** can take the values "***Request-Naming***" or "***Sequence-Naming***" or "***Auto-Naming***".
Two different naming schemes are supported:

1. ***Request Data File Naming*****:** The Name must be given in before each exposure is started (***START*** command given). The name is given in with the SETUP command (parameter "***DET.EXP.NAME <name>***"). The FITS-file will be named in the following way:

   ```
   <requested-name>_<frame-no>_<frame-name>.fits
   ```

   If only one INT frame is taken during the exposure the file name for this frame will be just <requested-name>.fits

2. ***Sequence Data File Naming*****:** An index is added to a base name. The index is incremented after each exposure. Setting the base name is done with the SETUP command (parameter "***DET.EXP.SEQ.NAME <name>***"). The index can be set with the "***DET.EXP.SEQ.NO <no>***" parameter. The FITS-file will be named in the following way:

   ```
   <seq-base-name><seq-no>_<frame-no>_<frame-name>.fits
   ```

   If only one INT frame is taken during the exposure the file name for this frame will be just <seq-base-name><seq-no>.fits

3. ***Auto Data File Naming:*** An index is added to a base name. When a new base name is set (or the naming scheme changes) a start index is determined automatically by searching the data target directory for files starting with the base-name. Initially (i.e. when ***DET.EXP.SEQ.NO*** is set to zero) the returned index is the highest existing index plus one. If ***DET.EXP.SEQ.NO*** is larger than zero the returned index is the first not existing index which is larger than ***DET.EXP.SEQ.NO***. Once the index is determined it is incremented by one (without further check) after each exposure until either the base-name or

the naming scheme changes or a new (minimum-)sequence number is explicitly set via a setup command. This makes it necessary that if ***DET.EXP.SEQ.NO*** is set to a value larger than zero, then no file with the current base-name and an index larger than ***DET.EXP.SEQ.NO*** must exist in the data target directory.

The name of the last file saved to disk is stored in the database attribute '***<alias>iracq:exposure.newDataFileName***'.

6.4.12 FITS HEADER

The FITS-header is generated as specified in the ***ESO-VLT-DIC.IRACE*** dictionary. With the setup command "***SETUP -function DET.FITSHDR.EXT.ST <T|F>***" the extended FITS header can be enabled or disabled. When the extended FITS-header is enabled also maintenance keywords like clock- and bias-voltages are stored.

6.4.13 DATA CUBES

To save the FITS-header and file generation overhead it is also possible to store the frames in data cubes. If the data cubes are enabled via the ***SETUP*** command "***DET.FILE.CUBE.ST T|F***", one data cube file is generated for each frame type that has been selected for storage. The number of images in the file is given in the ***NAXIS3*** FITS-header keyword. If the maximum size for one data-cube has been reached, the data transfer task will automatically create a new one for that frame type. The file naming is done in the same way as for normal FITS files. It is also possible to create data cubes with different frame types. Cubes can be defined in the detector configuration file:

```
DET.CUBEi.NAME "any-name"
DET.CUBEi.TYPES "HCYCLE1|HCYCLE2|INT|..."
```

The file name for ***cube<i>*** would be:

```
<BASENAME>_<CUBE_COUNTER>_<DET.CUBEi.NAME>
```

If ***DET.CUBEi.NAME*** is an empty string and the cube contains the *INT* frame it will be just:

```
<BASENAME>_<CUBE_COUNTER>
```

It is the users responsibility to take care, that the data types (integer, floating point, etc.) and the dimensions of all frames in the cube are the same.

6.4.14 DETECTOR MOSAICS

To increase transfer and processing bandwidth the acquisition processes may be distributed among several computers. The control server can handle several acquisition processes on different pre-processor workstations. They are all started/stopped synchronously. The acquisition processes are defined for each read-out mode in the detector configuration file via the ***DET.IRACE.RM<i>.ACQ<n>*** keyword. One might define as many acquisition processes as acquisition devices are pre-defined in the system configuration file (assigning host-name, DMA-device name and port numbers):

```
DET.IRACE.ACQ1.HOST      "<host1>";         # host name
DET.IRACE.ACQ1.CMDPORT    8001;             # command-server port
DET.IRACE.ACQ1.DATAPORT  8002;              # data-server port
DET.IRACE.ACQ1.ERRPORT    0;                # errorstack-server port
DET.IRACE.ACQ1.DEV       "/dev/ipc0_dma";   # dma-device name

DET.IRACE.ACQ2.HOST      "<host2>";         # host name
DET.IRACE.ACQ2.CMDPORT    8001;             # command-server port
DET.IRACE.ACQ2.DATAPORT  8002;              # data-server port
DET.IRACE.ACQ2.ERRPORT    0;                # errorstack-server port
DET.IRACE.ACQ2.DEV       "/dev/ipc0_dma";   # dma-device name
```

Additionally each acquisition process may handle more than one detector. The total number of detectors is given in the ***DET.NUMDET*** keyword in the detector configuration file. It is assumed, that each detector has the same size as given in ***DET.CHIP.NX, DET.CHIP.NY***. If ***DET.NUMDET*** is larger than 1, it is divided by the number of processes defined for the read-out mode. This value can be accessed in the acquisition process through the global variable ***sdmaNumDet***. Each acquisition process is then supposed to generate frames with the dimension in y-axis multiplied by ***sdmaNumDet*** (see section 4.3).

The data transfer task (***iracqDtt***) then receives frames from all acquisition processes and generates a separate file for each detector. The file names get the additional extension "***_DET<n>***", where ***<n>*** is the number of the detector. It is assumed that the order of the detectors is linear across the acquisition processes (i.e. first acquisition process handles the first ***sdmaNumDet*** detectors and so on). If the images are stored to data-cubes, the ***NAXIS3*** parameter increases proportional to ***DET.NUMDET*** (i.e. the data transfer task just adds the frames for all detectors to the cube). Apart from this the same rules for data cubes as described in section 6.4.13 are applied.

6.5 DATA TRANSFER BASE CLASS

The data transfer task (iracqDtt) is started, stopped and controlled via the control server. Generally the data transfer task runs on the same host as the control server. If the data transfer task should run on a different WS, the host name for the data transfer task can be specified in the system configuration file via the keyword "***DET.IRACE.DATAHOST <hostname>***". A base class (***iracqDTT***) is provided to allow application specific extensions. If the extension also wants to add callbacks or database access, the ***iracqDTT_EVH*** class has to be used as parent class instead of ***iracqDTT***.

6.6 SERVER EXTENSIONS

To customize your own server, a new server class has to be derived from the public ***iracqEVH*** class. There it is possible to add new command callbacks. The standard vltMakefile should be used to build the server process. The following flags have to be set:

```
myServer_LIBS= iracqEvh iracqSrv iracqExt iracq \
               irl tcomSclInt tcom sdmaInt \
               icmdSock icmd \
               evh eccs CCS fnd C++ c
```

A detailed example that is also contained in the test directory of the ***iracq***-module can be found in section 11.1.

To customize your own data transfer task, a new class has to be derived from the public ***iracqDTT_EVH*** class. There it is possible to add new command callbacks. The standard vltMakefile should be used to build the server process. The following flags have to be set:

```
myDtt_LIBS = iracqData iracqDataEvh iracqExt iracq irl \
             tcomSclInt tcom sdmaInt icmdSock icmd \
             evh eccs CCS fnd C++ c
```

The customized data transfer task is selected via the *"-dtt <data transfer task name>"* command line argument of the acquisition server. A detailed example that is also contained in the test directory of the ***iracq***-module can be found in section 11.2.

A special feature is the

virtual int ProcessFrame(sdmaFRAME_T *frame, char *erms)

method, which is defined in the ***iracqDTT*** base class. This method can be overloaded by the application to add own frame post-processing. It is a callback function, which is called just after a data frame has been received from the acquisition process. A pointer to the received frame is passed via the ***sdmaFRAME_T*** structure. If the function returns ***iracqFAILURE***, the string <erms> is sent as data error event to the control server and the exposure is discarded. If the '***FrameHandled()***' method is called within the call-back, no further action (scaling, post-processing, display, storage) is done with the current frame.

The ***sdmaFRAME_T*** structure is defined as follows:

```
/*
 *  frame header type
 */
typedef struct
{
  int   dtype;              /* data type                    */
  int   ftype;              /* frame type                   */
  int   start_x;            /* window start-x               */
  int   start_y;            /* window start-y               */
  int   nx;                 /* window nx                    */
  int   ny;                 /* window ny                    */
  int   scal;               /* scale factor                 */
  int   cnt;                /* frame counter                */
  int   setupId;            /* setup-id                     */
  int   err;                /* error-id                     */
  int   overrun;            /* overrun flag                 */
  int   frames;             /* available frame types        */
  int   tx;                 /* track point x                */
  int   ty;                 /* track point y                */
} sdmaFRAME_H;

/*
 *  frame type (external)
 */
typedef struct
{
  sdmaFRAME_H   h;           /* header structure */
  char         *fbuf;        /* frame buffer     */
} sdmaFRAME_T;
```

6.7 IMAGE POST-PROCESSING

The data transfer task provides a general mechanism to perform operations on the images received from the acquisition process before storing them to disk. The post-operations are selected via the

IMGOP command. The "***IMGOP -enable|disable***" command enables/disables the currently specified operations. The "***IMGOP -clear***" command clears all operations (-> no post-processing is done before storing the image on disk). The operations on the image are performed in the same order as the ***IMGOP*** command is issued. The following features are supported (each operation has to be set with a separate command):

```
"IMGOP -rotate <angle>":The image is rotated by <angle>.
                        Valid values are 0, 90, 180, 270.

"IMGOP -fx":            Flip image in x-direction.

"IMGOP -fy":            Flip image in y-direction.
```

Note that all operations are memorized until an "**IMGOP -clear**" is sent. So the command sequence "***IMGOP -rotate 90; IMGOP -rotate 90;***" will have the same effect (and also same performance) as "***IMGOP -rotate 180***".

6.8 BURST MODE

In some cases it might be necessary to store larger amounts of raw data or to sample at a very high frame-rate. If the frame rate is too high (> 200 Hz) the DMA-interrupt latency becomes dominating and no CPU-power is left for pre-processing. Two kinds of "burst modes" have been introduced to cover these two cases.

6.8.1 RAW DATA MODE

The raw data mode is activated by sending the setup command

"***SETUP -function DET.BURST.NUM <num>***"

If ***<num>*** is greater than zero it indicates the number of [***DET.WIN.NX, DET.WIN.NY***]-dimension sample-frames to be stored in the burst buffer. If an exposure is started both sequencer and acquisition are restarted (the ***DET.IRACE.SEQCONT*** flag is ignored in that case) and the buffer is filled until <num> sample-frames (16 bit short integer) are stored. At the end of the exposure an INT-frame (containing only dummy data at the moment) is transferred. The transfer of the sample-frames starts immediately and runs in parallel to the data recording. A 3 X 2 MBytes input-ring buffer is used to ensure interrupt stability. This means that at least 2 MBytes of data have to be created to satisfy the DMA. The dynamic parameter ***DET.BURST.SKIP*** indicates the number of frames to be skipped before starting to transfer.

This mode can be activated regardless of the currently selected read-out mode. Setting ***DET.BURST.NUM*** to zero deactivates the burst-mode and restores the standard acquisition process for the current read-out mode.

6.8.2 INTERNAL BURST MODE

The internal burst is activated by sending the setup command

"***SETUP -function DET.BURST.NUM <-num>***"

The negative value indicates that an internal burst-buffer should be applied. In this case the DMA is enlarged by a factor of ***<num>***. The data processing is not affected in this case as soft-interrupts are created for each sub-division. The calculation thread will wait for ***<num>*** buffers and will then process them within one step. This helps to workaround the 200Hz interrupt limitation, but depending on the actual processing it may slow down the acquisition performance as a whole (if for example least-square fit or standard deviation have to be solved within the ***<num>*** buffers).

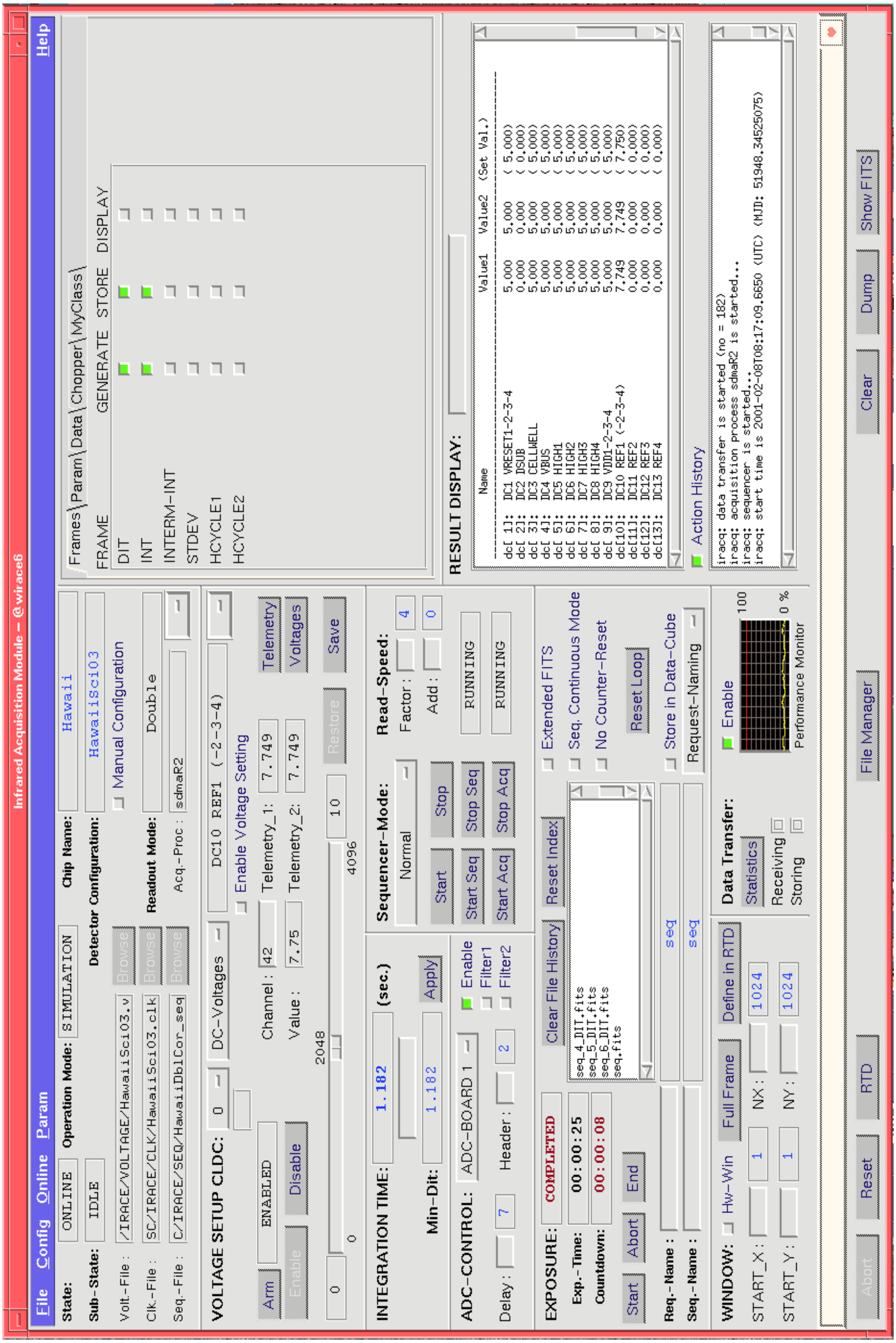
Also in this case a value of zero for DET.BURST.NUM deactivates the burst-mode.

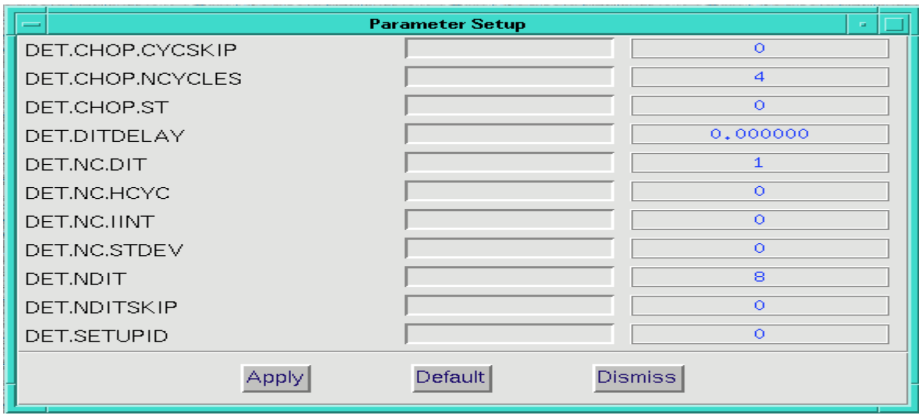
6.9 GRAPHICAL USER INTERFACE

```
Usage: iracqCtrl [options]

    -host <host name>   - host name of the server
                          (default: ins6)
    -load <module>      - load plug-in's in module
                          (searches lib<module>*.tcl)
    -noccs              - do not use CCS
                          (default: use CCS)
    -port <port number> - server port number (NOCCS only)
                          (default: 0)
    -server <name>      - server name
                          (default: iracqServer)
    -dtt <instance>     - process name of data transfer task
                          (default: iracqDtt)
    -verbose <level>    - set verbose level
                          (default: 0)
    -log <level>        - set log level
                          (default: 0)
    -inst <instance>    - server instance
                          (default: <>)
```

In the control panel there is a notebook area, where an application can insert widgets to control functions, that do not belong to the general DCS. These widgets should be designed as standard user interface classes (uif-classes) with the panel editor. The application has to provide its own Tcl-library to communicate with its server extension callbacks. If the '***-load <module-name>***' argument is set in the command line of iracqStart, the Tcl-library '***lib<module-name>.tcl***' is dynamically loaded and all uif-classes, that are contained in that library, are added to the notebook area of the standard DCS control panel. If the Tcl-library contains a '***<module-name>Init***' procedure, this is called during initialization. The instance of the uif-classes is the module name. So all global variables used in the Tcl-library are accessed via '***gvar(<module-name>_variableName)***'. Each uif-class appears on a separate notebook-page. The title of the page is derived from the filename. The uif-class with the name '***<module-name>MyClass_uifClass.tcl***' will appear as page with the title '***MyClass***'.


Infrared Acquisition Module -- @wirace6
File Config Online Param Help
State: ONLINE
Operation Mode: SIMULATION
Chip Name: Hawaii
Sub-State: IDLE
Detector Configuration: HawaiiSci03
Volt.-File : /IRACE/VOLTAGE/HawaiiSci03.v
Clk.-File : SC/IRACE/CLK/HawaiiSci03.clk
Seq.-File : C/IRACE/SEQ/HawaiiDblCor_seq
Manual Configuration
Readout Mode: Double
Acq.-Proc : sdmaR2
VOLTAGE SETUP CLDC: 0 DC-Voltages DC10 REF1 (-2-3-4)
Arm ENABLED Enable Disable
Enable Voltage Setting
Channel : 42 Telemetry_1: 7.749 Telemetry
Value : 7.75 Telemetry_2: 7.749 Voltages
0 2048 4096 10 Restore Save
INTEGRATION TIME: 1.182 (sec.)
Min-Dit: 1.182 Apply
Sequencer-Mode: Normal
Read-Speed: Factor : 4 Add : 0
Start Stop
ADC-CONTROL: ADC-BOARD 1 Enable Filter1 Filter2
Delay : 7 Header : 2
Start Seq Stop Seq RUNNING
Start Acq Stop Acq RUNNING
EXPOSURE: COMPLETED
Exp.-Time: 00:00:25
Countdown: 00:00:08
Clear File History Reset Index
Extended FITS
Seq. Continuous Mode
No Counter-Reset
Start Abort End Reset Loop
Req.-Name : seq
Seq.-Name : seq
Store in Data-Cube
Request-Naming
WINDOW: Hw-Win Full Frame Define in RTD
START_X : 1 NX : 1024
START_Y : 1 NY : 1024
Data Transfer: Statistics Enable
Receiving Storing
Performance Monitor
Frames Param Data Chopper MyClass
FRAME GENERATE STORE DISPLAY
DIT
INT
INTERM-INT
STDEV
HCYCLE1
HCYCLE2
RESULT DISPLAY:
Action History
Abort Reset RTD File Manager Clear Dump Show FITS



6.10 START-UP SCRIPT

The startup/shutdown of the Infrared Data Acquisition is done via the ***iracqStart/Stop*** script:

```
Usage: iracqStart [options]

     -man                      just show on-line manual
     -verbose <level>          set verbose level
                               (default: 0)
     -log <level>              set log level
                               (default: 0)
     -inst <instance>          set server instance
                               (default: <>)
     -xterm                    start processes in xterms
     -cfg <name>               system configuration file name
                               (default: sys.cfg)
     -det <name>               detector configuration file name
     -sim                      start server in simulation mode
     -gui                      starts also the control panel
     -noccs                    do not use CCS
     -port <server port>       specify the command server port
                               (only for maintenance and NOCCS)
     -server <serverName>      uses <serverName> as control server
                             (default: iracqServer/iracqServerNoCcs)
     -dtt <name>               process name of data transfer task
                               (default: iracqDtt)
    -load <mymod>            load uif-classes of <mymod> into panel
                               (this might be repeated)
```

```
Usage: iracqStop [options]

     -verbose <level>      set verbose level
                           (default: 0)
     -log <level>          set log level
                           (default: 0)
     -inst <instance>      set server instance
                           (default: <>)
     -gui                  stops also the control panel
     -noccs                do not use CCS
     -port <server port>   specify the command server port
                           (only for maintenance and NOCCS)
     -server <serverName>  uses <serverName> as control server
                         (default: iracqServer/iracqServerNoCcs)
```

6.11 ON-LINE MANUAL

The infrared acquisition module (*iracq*) is delivered together with an on-line manual covering the main issues of this user manual. The manual can be called either from the UNIX-shell via the start-up-script ('*iracqStart -man*') or via the '*Extended Help*' button in the control panel.

7 CONFIGURATION FILES

Directory Structure:

```
$INS_ROOT/SYSTEM/MISC/IRACE/SYS      - System Configuration File(s)
                                       and IRACE bootables.
                           /DET      - Detector Configuration File(s)
                           /DSUP     - Default Setup Files
                           /VOLTAGE  - Voltage Setup Files
                           /CLK      - Clock Pattern Setup Files
                           /SSD      - Subpattern Dispatcher Files
                           /SEQ      - Sequencer Program Files
```

The following file name extensions are used:

Bootables:	*.btl*
System Configuration Files:	*.cfg*
Detector Configuration Files:	*.dcf*
Voltage Setup Files:	*.v*
Clock Pattern Setup Files:	*.clk*
Sequencer Subpattern Dispatcher Files:	*.ssd* (trivial loop format) *_ssd* (loop executable - tcl format)
Sequencer Program Files:	*.seq* (trivial sequence files) *_seq* (loop executable - tcl format)

7.1 SYSTEM CONFIGURATION FILE

```
############################################################################
# E.S.O. - VLT project
#
# "@(#) $Id: sys.cfg,v 1.1+ 1999/09/29 14:47:45 vltsccm Exp $"
#
# who       when      what
# --------  --------  ----------------------------------------------
# jstegmei  13/10/98  Created
#
############################################################################
#
# DESCRIPTION
# Example ASCII FILE for system configuration.
#
############################################################################

# RTD Configuration
DET.CON.RTD.HOST        "$HOST";         # RTD Host Name
DET.CON.RTD.ENV         "$RTAPENV";      # RTD Environment Name
DET.CON.RTD.DISP        "$DISPLAY";      # Display name for the RTD
DET.CON.RTD.INT         "F";             # use internal RTD-channel

# system path
DET.IRACE.SYSPATH      "$INS_ROOT/SYSTEM/MISC/IRACE";

# data path
DET.IRACE.DTT.PATH     "$INS_ROOT/SYSTEM/DETDATA";

# data transfer task host
DET.IRACE.DTT.HOST     "$HOST";

# software configuration
DET.CON.OPMODE         "SIMULATION";      # default operation mode
#DET.CON.OPMODE         "NORMAL";          # default operation mode
DET.CON.NAMING.TYPE    "Request-Naming";  # default naming type to use
DET.CON.CLDC.ST        "F";               # enable CLDC at ONLINE command
DET.CON.IRACE.ST       "F";               # start IRACE at ONLINE command
DET.CON.SIM.ST         "F";               # simulation installation
DET.CON.DETCFG         "HawaiiSci03";     # default detector config.
DET.CON.AUTONLIN       "F";               # go ONLINE after start-up

# FITS header
DET.FITSHDR.SIZE.NO     8;                # FITS header size (no. of blocks)
DET.FITSHDR.EXT.ST     "F";               # extended FITS header

# front-end configuration
DET.IRACE.SCLHOST      "$SDMA_HOST";      # host of the irace front-end
DET.IRACE.SCLSRVPORT    $TCOM_SRV;        # scl-server port
DET.IRACE.SCLPROTTYPE  "ipc";             # scl-interface-type
DET.IRACE.SCLDEV       "/dev/ipc0_com";   # scl-interface device name
DET.IRACE.SCLBOOTFILE  "sclpp.btl";       # bootable file for irace front-end
DET.IRACE.NUMCLDC       1;                # only for simulation
DET.IRACE.NUMADC        2;                # number of ADC boards in system
DET.IRACE.VDIFFMAX      0.5;              # maximum telemetry difference
DET.IRACE.NCLIENT       3;                # maximum number of data clients

# data acquisition configuration
DET.IRACE.ACQ1.HOST     "$SDMA_HOST";     # host
DET.IRACE.ACQ1.CMDPORT   $SDMA_CMD;       # command-server port
DET.IRACE.ACQ1.DATAPORT  $SDMA_DATA;      # data-server port
DET.IRACE.ACQ1.ERRPORT   0;               # errorstack-server port
DET.IRACE.ACQ1.DEV      "/dev/ipc0_dma";  # dma-device name
DET.IRACE.ACQi.HDR       1;               # header on IRACE data-bus


DET.IRACE.ADC1.ADDR      1;               # address on IRACE-Status-Bus
DET.IRACE.ADC1.NAME      "ADC-G1";        # ADC group name

DET.IRACE.ADC2.ADDR      2;               # address on IRACE-Status-Bus
DET.IRACE.ADC2.NAME      "ADC-G2";        # ADC group name

#
# --- oOo ---
```

7.2 DETECTOR CONFIGURATION FILE

```
############################################################################
# E.S.O. - VLT project
#
# "@(#) $Id: Sb256.dcf,v 1.1+ 1999/09/29 14:47:45 vltsccm Exp $"
#
# who       when      what
# --------  --------  ----------------------------------------------
# jstegmei  13/10/98  Created
#
############################################################################
#
# DESCRIPTION
# ASCII FILE for detector configuration.
#
############################################################################

DET.CHIP.NO              1;               # unique detector number
DET.CHIP.NAME          "Sb256";           # detector name
DET.CHIP.ID            "ESO-Sb256";       # eso detector id
DET.CHIP.TYPE          "IR";              # type
DET.CHIP.NOCHAN          4;               # number of data channels
DET.CHIP.PXSPACE         3.000E-05;       # pixel-pixel spacing
DET.CHIP.NX            256;               # chip-size x-direction
DET.CHIP.NY            256;               # chip-size y-direction
DET.CHIP.ADJUST        "FREE"             # window adjust mode
DET.CHIP.STEPX           8;               # adjust step x-direction
DET.CHIP.STEPY           2;               # adjust step y-direction
DET.CHIP.PARTNX        256;               # partition size x-direction
DET.CHIP.PARTNY        256;               # partition size y-direction

DET.RSPEED               4;               # global read-speed factor
DET.RSPEEDADD            0;               # global read-speed add

DET.IRACE.CLDC           0;               # cldc-board number

DET.IRACE.VOLTAGES    "Sb256.v";          # voltage file

DET.IRACE.CLOCKS      "Sb256.clk";        # clock-pattern file


DET.IRACE.RMDEF          2;               # id of default read-mode

# readout modes
DET.IRACE.RM1.NAME    "uncorr";           # read-mode name
DET.IRACE.RM1.ACQ     "sdmaSb1";          # acquisition process
DET.IRACE.RM1.SEQ     "Sb256Uncor_seq";   # readout sequence
DET.IRACE.RM1.DSUP    "NONE";             # default parameters

DET.IRACE.RM2.NAME    "double";           # read-mode name
DET.IRACE.RM2.ACQ     "sdmaSb2";          # acquisition process
DET.IRACE.RM2.SEQ     "Sb256Dbl_seq";     # readout sequence
DET.IRACE.RM2.DSUP    "Sb256DblCor.dsup"; # default parameters

DET.IRACE.RM4.NAME    "non-destructive";  # read-mode name
DET.IRACE.RM4.ACQ     "sdmaSb4";          # acquisition process
DET.IRACE.RM4.SEQ     "Sb256NonDest_seq"; # readout sequence
DET.IRACE.RM4.DSUP    "Sb256NonDest.dsup";# default parameters

# ADC-board configuration
DET.IRACE.ADC1.HEADER     1;              # header on IRACE data-bus
DET.IRACE.ADC1.ENABLE     1;              # enable/disable (0/1)
DET.IRACE.ADC1.FILTER1    0;              # filter1 off/on (0/1)
DET.IRACE.ADC1.FILTER2    0;              # filter2 off/on (0/1)
DET.IRACE.ADC1.DELAY      0;              # variable delay

DET.IRACE.ADC2.HEADER     2;              # header on IRACE data-bus
DET.IRACE.ADC2.ENABLE     0;              # enable/disable (0/1)
DET.IRACE.ADC2.FILTER1    0;              # filter1 off/on (0/1)
DET.IRACE.ADC2.FILTER2    0;              # filter2 off/on (0/1)
DET.IRACE.ADC2.DELAY      0;              # variable delay

DET.MODE1.NAME   "mode1";
DET.MODE1.SETUP  "DET.VOLT.DC1 -2.8, DET.RSPEED 2, DET.RSPEEDADD 0"
DET.MODE2.NAME   "mode2";
DET.MODE2.SETUP  "DET.VOLT.DC1 -3.5, DET.RSPEED 3, DET.RSPEEDADD 0"

#
# --- oOo ---
```

7.3 DEFAULT SETUP FILE

The default setup file (.dsup) contains default values for the dynamic acquisition process parameters. It is also possible to define any new parameter here, which can then be used in subpattern dispatcher files and the sequencer program files. For each readout mode there exists exactly one default setup file.

```
#############################################################################
# E.S.O. - VLT project
#
# "@(#) $Id: Sb256DblCor.dsup,v 1.1+ 1998/10/08 15:39:14 vltsccm Exp vltsccm $"
#
# who       when      what
# --------  --------  ----------------------------------------------
# jstegmei  13/08/99  Created
#
#############################################################################
#
# DESCRIPTION
# ASCII FILE for default parameter setup.
#
#############################################################################

DET.NC.DIT           1
DET.NC.IINT          0
DET.NC.STDEV         0
DET.NDITSKIP         1
DET.DITDELAY         0.100
DET.MYPARAM          0.3

# --- oOo ---
```

7.4 VOLTAGE SETUP FILE

The voltage setup files contains the definitions of all clock- and DC-bias voltages for one detector:

```
########################################################################
# E.S.O. - VLT project
#
# xxxx.config
#
# who       when      what
# --------  --------  ----------------------------------------------
#
########################################################################
#
# DESCRIPTION
# Voltage file.
#
########################################################################

########################################################################
# Clock Voltages:
########################################################################

DET.VOLT.CLKHINM1  "ck1Hi.SyncFast";  # Name High Clock Voltage 1
DET.VOLT.CLKHI1               -7.35;  # Level High Clock Voltage 1
DET.VOLT.CLKHIRA1    "[-8.0, -6.5]";  # Range High Clock Voltage 1
DET.VOLT.CLKHIDAC1                1;  # DAC Ch High Clock Voltage 1
DET.VOLT.CLKHITEL1                1;  # Tel Ch High Clock Voltage 1

DET.VOLT.CLKLONM1  "ck1Lo.SyncFast";  # Name Low Clock Voltage 1
DET.VOLT.CLKLO1               -3.03;  # Level Low Clock Voltage 1
DET.VOLT.CLKLORA1    "[-3.5, -2.0]";  # Range Low Clock Voltage 1
DET.VOLT.CLKLODAC1                2;  # DAC Ch Low Clock Voltage 1
DET.VOLT.CLKLOTEL1                2;  # Tel Ch Low Clock Voltage 1

### 16 clocks in this syntax

########################################################################
# DC Voltages:
########################################################################

DET.VOLT.DCNM1      "DC1-VDDUC";  # Name DC Voltage 1
DET.VOLT.DC1               -3.8;  # Level DC Voltage 1
DET.VOLT.DCRA1   "[-4.5, -3.5]";  # Range DC Voltage 1
DET.VOLT.DCDAC1              33;  # DAC Ch DC Voltage 1
DET.VOLT.DCTEL1              33;  # Tel Ch DC Voltage 1

### 16 dc voltages in this syntax
```

7.5 CLOCK-PATTERN SETUP FILE

The clock-pattern setup file contains the definitions of all subpatterns needed for the readout modes of one detector:

```
#########################################################################
# E.S.O. - VLT project
#
# xxxx.clk
#
# who       when      what
# --------  --------  ----------------------------------------------
#
#########################################################################
#
# DESCRIPTION
# Clock-pattern file.
#
#########################################################################

# Header parameters
DET.CHIP.NO                1;  # Detector Type
DET.SUBPAT.NO             15;  # No of Sub-Patterns
DET.CLKP.NO               11;  # No of Clocks

# Placement of software clocks onto hardware clocks
# Maximum 5 clocks per line. Can continue over several lines if terminated
# with ",".
DET.CLKP.SWHWCLK1  "1,2,3,4,5,";
DET.CLKP.SWHWCLK2  "6,7,8,9,33,";
DET.CLKP.SWHWCLK3  "11";

DET.SUBPAT1.NAME        "Delay";         # Subpattern name
DET.SUBPAT1.STATES            4;         # No of states
DET.SUBPAT1.RFAC              1;         # repetition factor
DET.SUBPAT1.STATEV1      "0000";         # SYNC Fast
DET.SUBPAT1.STATEV2      "0000";         # CK1 Fast
DET.SUBPAT1.STATEV3      "0000";         # CK2 Fast
DET.SUBPAT1.STATEV4      "0000";         # SYNC Slow
DET.SUBPAT1.STATEV5      "0000";         # CK1 Slow
DET.SUBPAT1.STATEV6      "0000";         # CK2 Slow
DET.SUBPAT1.STATEV7      "0000";         # CK RESET
DET.SUBPAT1.STATEV8      "0000";         # VBout CK
DET.SUBPAT1.STATEV9      "0000";         # Trigger
DET.SUBPAT1.STATEV10     "0000";         # Convert
DET.SUBPAT1.STATEV11     "0000";         # Spare
DET.SUBPAT1.RSPEEDV      "10,10,10,10";  # readspeed
DET.SUBPAT1.RSPEEDP      "0,0,0,0";      # permission

### up to 31 subpatterns like this
# DET.SUBPAT<nr>.NAME               "xxxxxx";      #subpattern name
# DET.SUBPAT<nr>.STATES             x;             #no of states
# DET.SUBPAT<nr>.RFAC               x;             #repetition factor
# DET.SUBPAT<nr>.STATEV<statNr>     "xxxx";        #DET.SUBPAT<nr>.STATES states
# DET.SUBPAT<nr>.RSPEEDV            "x,x,x,x";     #read speed for each state
# DET.SUBPAT<nr>.RSPEEDP            "x,x,x,x";     #permission for each state
#
# nr     = [1 ... DET.SUBPAT.NO]
# statNr = [1 ... DET.CLKP.NO]
###
```

7.6 IRACE LOOP STRUCTURES

The clock-patterns (smallest unit) as described in section 3.4 are stored in the sequencer "memory".. The read-outs are described by loops araound these units. The loops are downloaded to the sequencer as (checked) structural code and not in ASCII format. The ASCII format for the loops is interpreted at higher level. Figure 10 shows the three hirarchies of clock pattern loops

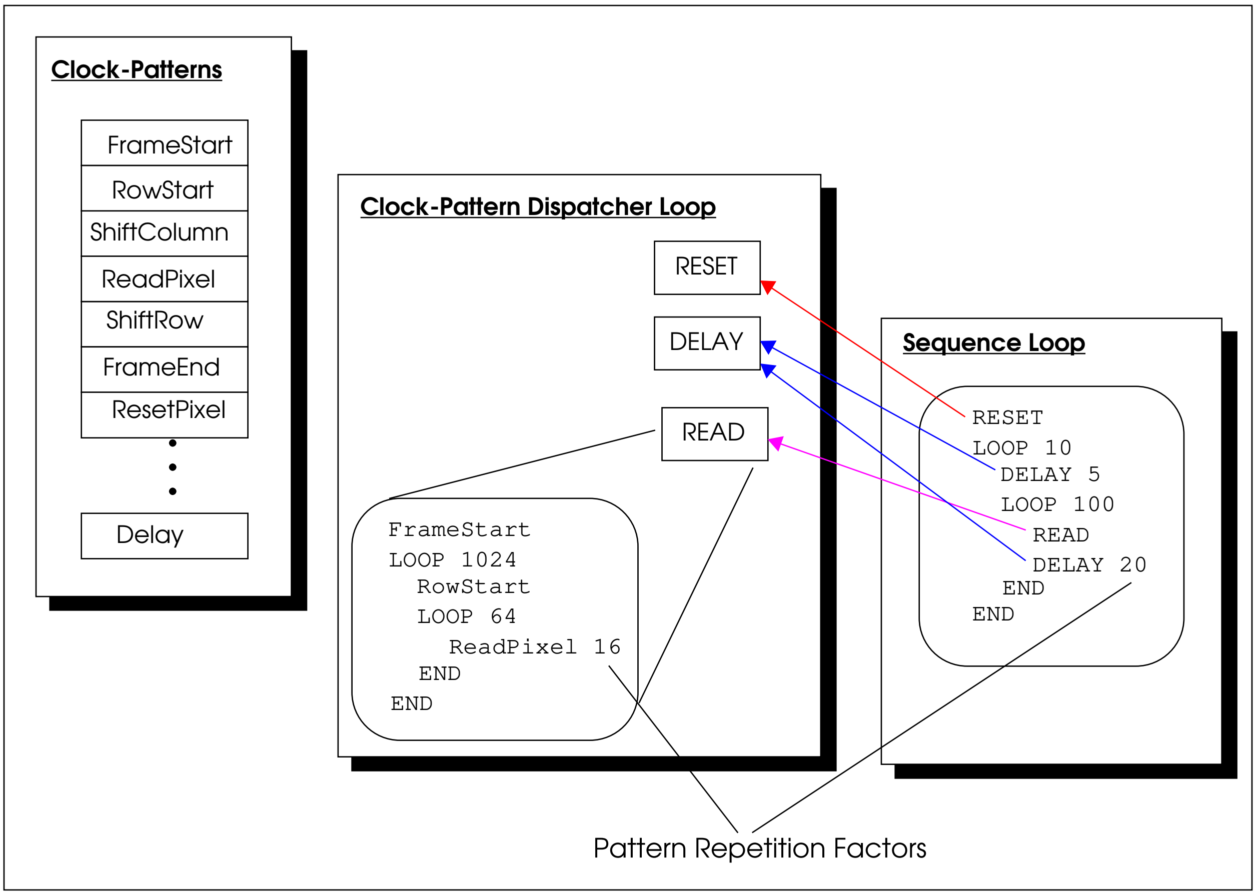


Figure 10

The loop files do not need parsers and expression evaluators. Both subpattern dispatcher files and sequencer program files are executable Tcl-scripts, which are invoked as sub-shells from DCS. The Tcl-language has been chosen, as this is already used for the templates and is supported by all VLT-releases. Within the Tcl-script the developer has full freedom to define local variables to be used in expressions. Often used expressions can be assembled in a Tcl-library by each application. External parameters held by DCS have to be explicitly declared as such within a ***USE*** command. The parameters are referred to afterwards as $irlVar(parameter-name).

References to sub-pattern dispatcher files are done with the

> ***"ASSIGN <name> <file-name>"***

construct in the sequencer program file. The "***$gvar(T_$<name>)***" keyword can be used in the sequencer program file as a substitution for the execution time of a sub-pattern dispatcher program. The special loop-counter "-1" is used to define an infinite loop (only valid in the sequencer program file).

The resulting loop object and possibly new/changed DCS-parameter values are passed back by the

Tcl-shell to the calling function with the “***PROGEND***” command (this will also exit the Tcl-shell). Generally all parameters in the ***$irlVar()*** array are passed back. By defining new array elements it is possible to define and set new DCS-parameters within the Tcl-shell. For later usage within DCS these have to be defined in the dictionary or at least in a default parameter setup file (for NOCCS version) of the current read-out mode.

The Tcl-scripts are built with the standard vltMakefile. The ***<irlParse>*** library has to be added to the Tcl-library list in the Makefile. The TCLSH for the scripts should be set to “tclsh”

Keyword Summary:

```
“ASSIGN <name> <file-name>”  - Assign a name to a sub-pat. dispatcher file

“USE <parameter list>”       - Declare external parameters

“LOOP <loop counter>”        - Loop start token

“END”                        - Loop end token

“EXEC <$name> <n>”           - Execute sub-pat. dispatcher <name> n times
“EXEC <nr> <n>”              - Execute sub-pat. number <nr> n times

 “SYNC”                      - Insert synchronization point
                               only relevant in external trigger mode)

“PROGEND”                    - Pass back results and exit
```

<u>Example for Sub-Pattern Dispatcher File:</u>

```
set DELAY 1
set FRAME_START 2
set READ1 3
set READ2 4
set FRAME_END 5

USE DET.WIN.NX DET.WIN.NY DET.WIN.TYPE

if {$irlVar(DET.WIN.TYPE) == 0} {
    set nx 64
    set ny 128
} else {
    set nx [expr ($irlVar(DET.WIN.NX) / 2)]
    set ny $irlVar(DET.WIN.NY)
}

EXEC $FRAME_START 1
LOOP $ny
    LOOP $nx
        EXEC $READ1 1
        EXEC $READ2 1
    END
END
EXEC $FRAME_END 1

PROGEND
```

Example for Sequencer Program File (-> Current Sequencer Program):

```
ASSIGN DELAY "../SSD/Delay_ssd"
ASSIGN FRAME "../SSD/Read_ssd"
ASSIGN RESET "../SSD/Reset_ssd"


USE DET.DIT DET.NDIT DET.NDITSKIP

if {$irlVar(DET.NDIT) <= 0} {
    set irlVar(DET.NDIT) 1
}

set irlVar(DET.MINDIT) $gvar(T_$FRAME)
set t1 [expr ($irlVar(DET.NDIT) + $irlVar(DET.NDITSKIP))]

set irlVar(DET.EXPTIME) [expr ($t1 * \
    ($irlVar(DET.DIT) + $gvar(T_$FRAME) + $gvar(T_$RESET)))]

set delFac [expr ($irlVar(DET.DIT) - $gvar(T_$FRAME)) / $gvar(T_$DELAY)]

LOOP -1
    LOOP $irlVar(DET.NDIT)
        EXEC $RESET
        EXEC $FRAME 1
        EXEC $DELAY $delFac
        EXEC $FRAME 1
    END
SYNC
END

PROGEND
```

8 ENVIRONMENT SETTING AND INSTALLATION

8.1 HW AND SW REQUIREMENTS

The following components are needed for running a full installation of DCS:

- IRACE Workstation. This can be one of:
  - Ultra-Sparc (2 CPU) running Solaris 2.6 or later.
    Recommended are ULTRA-2 for SBus or ULTRA 60/80 for PCI-Bus.
    Plus DMA Interface(s) EDT SCD 60 (SBus) or EDT PCICD 60 (PCI-Bus).
  - Linux PC (2 CPU) running Red Hat Linux Release 7.3 or later.
    Plus IRACE PCI-Bus Interface Board.
- Instrument Workstation. This can be one of:
  - HP-Workstation running HP-UX 10.20 or HP-UX 11.
  - Sun-Workstation running Solaris 2.6 or later.
  - PC running Red Hat Linux Release 7.3 or later.
- VLT SW Package NOCCS for the IRACE Workstation.
- VLT SW Package CCS (HP-UX 10-20) or CCSLite (HP-UX 11, Solaris, Linux) for the IWS.

For software tests it is possible to run the data-acquisition software also on HP-UX in simulation mode. In that case only the HP-Workstation is needed. As the simulation of the acquisition process consumes a lot of system resources and CPU-time it is recommended not to run the simulation on the IWS at high data-rates (< 2 MHz).

8.2 ENVIRONMENT VARIABLES

The following environment variables have to be set if not marked as optional:

```
$INS_USER        Specifies the current user (defaults to SYSTEM, if not set).
$INS_ROOT        Root directory of the instrument configuration.
$IR_SPARC        SPARC_SBUS(optional) or SPARC_PCI.
$SDMA_HOST       Host name of the IRACE - pre-processor workstation
                 (for simulation, this can also be the HP-workstation).
$SDMA_DEV        Default device name of the DMA-driver (optional).
$SDMA_CMD        Command port of the DMA pre-processor.
$SDMA_DATA       Data port of the DMA pre-processor.
$SDMA_ERR        Error port of the DMA pre-processor (optional).
$TCOM_SRV        Command port of the IRACE front-end command server.
$IRACQ_PORT      Command port for the NOCCS control server (only
                 needed for NOCCS installations on the IRACE Workstation).
$RTD_CAMERA      RTD-Camera name.
$IRTD_PORT       Port number for the RTD-interface.
```

The port numbers must not exist in /etc/services of $SDMA_HOST.

8.3 IRD PACKAGE INSTALLATION

The procedure to create the IRD Software package consists of the following steps:

(1) Retrieve from the archive and install module irdarch:

```
a) mkdir IRDROOT; cd IRDROOT
```

```
b) cmmCopy irdarch
c) cd irdarch/src; make prepare_installation
```

(2) Go to installation directory:

```
cd ../../INSTALL
```

(3) Retrieve all needed modules from the archive:

```
./buildFromArchive
```

(4) Build and install modules:

```
./buildIRD
```

(5) Run automatic test procedure:

```
./buildTestIRD (DISPLAY variable must be set)
```

Steps 3 to 5 can also be done in one step:

```
./buildAll
```

This procedure is only applicable on the instrument workstation. The software on the IRACE workstation must be installed manually as after retrieving the irace module the DMA interface driver has to be installed before compilation. The driver installation requires root access.

8.4 MANUAL SOFTWARE INSTALLATION

The first step of installation is to get the standard IRACE-DCS base modules:

IRACE-DCS:

cmmCopy irace
cmmCopy iracq
cmmCopy dicIRACE

Optional RTD interface:

cmmCopy rtdb
cmmCopy irtd

8.4.1 SOFTWARE INSTALLATION ON LINUX

Driver installation:

- The driver sources have to be compiled on the target operating system:

```
cd irace/SYSTEM/DRV/
gunzip irpci.tar.gz
tar xvf irpci.tar
cd irpci/src
make
su
<enter password>
```

```
make install
```

- Afterwards the driver module can be loaded and unloaded using the ***irpci_load/unload*** scripts:

```
../bin/irpci_load
../bin/irpci_unload
```

The ***irpci_load*** script will also create the appropriate entries in the ***/dev*** directory. To load the driver module at boot time the following line has to be added to ***/etc/rc.local***:

```
/usr/local/bin/irpci_load
```

Software Installation:

- If a version of irace is already installed in $INTROOT:

```
cd $INTROOT/bin; su
<enter password>
./sdmaChMod <user-name> and/or
./chmod a-s <your application processes>
./chown <user-name> <your application processes>
```

- Compilation:

```
cd irace/src; make
cd iracq/src; make man all install
cd rtdb/src; make man all install
cd irtd/src; make man all install
```

- Set root privileges for acquisition processes (for real-time scheduling):

```
cd $INTROOT/bin; su
<enter password>
./sdmaChMod su (to enable real-time facilities) and/or
chown root <your acquisition processes>
chmod a+s <your acquisition processes>
```

8.4.2 SOFTWARE INSTALLATION ON SOLARIS

The interface device drivers for IRACE are found in the ***irace/SYSTEM/DRV*** directory of the irace module. Generally the DMA driver-packages (EDTscd for SBUS-based sparcs, EDTpcd for PCI-based sparcs) have to be installed before compilation on the Ultra-Sparc, as the driver interface defined in sdmaDrv.h will include header files from that package and may also link object files (PCI-version). If the package cannot be installed, one has the possibility to compile also on the Ultra-Sparc for simulation purposes only (by using the MAKE_SIM=T flag when compiling the irace/iracq modules).

Some entries have to be added to /etc/system:

```
For IPC facilities:
set msgsys:msginfo_msgmap=1024
set msgsys:msginfo_msgmax=8192
set msgsys:msginfo_msgmnb=65535
set msgsys:msginfo_msgmni=512
set msgsys:msginfo_msgseg=4096
set msgsys:msginfo_msgssz=64
set msgsys:msginfo_msgtql=1024
set semsys:seminfo_semmap=2048
set semsys:seminfo_semmni=500
```

```
set semsys:seminfo_semmns=8192
set semsys:seminfo_semmsl=500
set semsys:seminfo_semopm=50
set semsys:seminfo_semvmx=32767
set shmsys:shminfo_shmmax=32000000
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=64
set shmsys:ism_off = 1

For faster scheduling:
set hires_tick = 1
```

Driver Installation for SBus:

- To remove a previously installed driver:

```
su
<enter password>
pkgrm EDTscd; rm -rf /opt/EDTscd
```

- To install the driver:

```
cd irace/SYSTEM/DRV;
gunzip SCD64.tar.gz
tar xf SCD64.tar
su
<enter password>
cp SCD64/scd /platform/`uname -m`/kernel/drv/sparcv9
cp EDTscd2.88_64 /usr/spool/pkg/

cd /usr/spool/pkg;
pkgadd EDTscd

cd /opt/EDTscd
./scdrequest
enter <1> (RXT looped...)
```

Now you have to check the permissions of the parallel port device referenced by /dev/bpp0 (type 'ls -lL /dev/bpp0'; this is a link to '/devices/sbus...'). The device must be given read/write permissions for all users.

Driver Installation for PCI-Bus:

- To remove a previously installed driver:

```
su
<enter password>
pkgrm EDTpcd; rm -rf /opt/EDTpcd; rm -f /usr/lib/libedt.so
```

- To install the driver:

```
cd irace/SYSTEM/DRV; su
<enter password>
cp EDTpcdx.xx.tar /usr/spool/pkg/
<x.xx> is the version of the driver (currently EDTpcd2.304.tar)

cd /usr/spool/pkg; tar xvf EDTpcdx.xx.tar; rm EDTpcdx.xx.tar
pkgadd EDTpcd
```

```
cd /opt/EDTpcd
./pcdrequest
enter <1> (RXT looped...)

ln -s /opt/EDTpcd/libedt.so /usr/lib/libedt.so
```

Software Installation (SBus and PCI-Bus):

- If a version of irace is already installed in $INTROOT:

```
cd $INTROOT/bin; su
<enter password>
./sdmaChMod <user-name> and/or
./chmod a-s <your application processes>
./chown <user-name> <your application processes>
```

- Compilation:

```
cd irace/src; make
cd iracq/src; make man all install
cd rtdb/src; make man all install
cd irtd/src; make man all install
```

- Set root privileges for acquisition processes (for real-time scheduling):

```
cd $INTROOT/bin; su
<enter password>
./sdmaChMod su (to enable real-time facilities) and/or
chown root <your applications>
chmod a+s <your applications>
```

8.4.3 SOFTWARE INSTALLATION ON IWS

The following steps have to be done to install the software on the instrument workstation:

```
cd irace/src; make (or 'make MAKE_POSIX=T' to compile also the posix-wrapper for
the Solaris data-acquisition simulation)

cd iracq/src; make man all install
cd dicIRACE/src; make all install
cd rtdb/src; make man all install
cd irtd/src; make man all install

cd $VLTDATA/ENVIRONMENTS/$RTAPENV/dbl

add 'iracqBranch.db' to 'DATABASE.db' (-> Database)

make db
vccEnvStop -e $RTAPENV
vccEnvInit -e $RTAPENV
vccEnvStart -e $RTAPENV
```

9 INTERACTIVE MODE

The interactive mode has been designed for stand-alone tests of the IRACE hardware and for development of acquisition processes.

9.1 PRE-PROCESSOR INTERACTIVE MODE

To start the acquisition process type:

> ***sdmaXXX -v -nclient 2*** (Santa Barbara uncorrelated)

This starts the plug-ins's in default mode producing DIT- and INT-frames. You can start them by typing this line and stop them with ctrl-c.

Remember: first boot the irace front end, then start the plug-in process, then start the sequencer.

To start a plug-in interactive mode (so that one can enter start/stop/stat commands etc.) the -i command line option has to be set and the interactive interface process sdmaImode has to be started (on any host):

Example:

> host-a: ***sdmaXXX -v -nclient 2 -i***
> host-b: ***sdmaImode -host <host-a>***

After starting ***sdmaImode*** you will get help with '?' or '? cmd'.

Important commands:

start	- start acquisition
stop	- stop acquisition
pset <parameter name> <value>	- set parameter
psync <timeout>	- overtake all parameters set with pset
time	- get current calculation time and data input interval

If one starts a new plug-in by hand, while the sdmaImode process is still running, one has to type '***connect***' (in sdmaImode) to reconnect to the plug-in.

To start the RTD type:

> ***irtd&***

9.2 DETECTOR FRONT-END INTERACTIVE MODE

To run the interactive mode for the IRACE controller, the bootables and the example configuration files should be installed in a test directory:

> ***iraceInstallData <iraceTest>*** (iraceTest is the name of the test directory)

Already existing files with the same name as the default files are overwritten, all others remain untouched.

9.2.1 PROCESS STARTUP

<u>On Linux-PC:</u>

```
tcomSclImode -v -type ipc -dev /dev/ipc0_com -b ../SYS/sclpp.btl
(- scl2pp.btl for 2 cldc)
```

<u>On SPARC with SBus:</u>

```
tcomSclImode -v -type bpp -dev /dev/bpp0 -b ../SYS/sclpp.btl
(- scl2pp.btl for 2 cldc)
```

<u>On SPARC with PCI-Bus:</u>

```
tcomSclImode -v -type ecpp -dev /dev/ttya -b ../SYS/sclpp.btl
(- scl2pp.btl for 2 cldc)
```

After starting tcomSclImode you will get help with '?' or '? cmd'

9.2.2 IMPORTANT COMMANDS

csetup	- print clock setup
fsetup	- print frame setup (readout loop)
vsetup	- print voltage setup
pstat	- read parameters and subpatterns
tel	- telemetry
start	- start sequencer
stop	- stop sequencer
pset rspeed <factor> <add>	- set readspeed
pset dc <n> <value>	- set dc-voltage n
pset clkhi <n> <value>	- set clock high n
pset clklo <n> <value>	- set clock lo n
pset dac <n> <value>	- set dac channel n
pget rspeed	- get current readspeed
pget dc <n>	- get dc-voltage n
pget clk <n>	- get clock-voltages (high/low) n
pget tel <n>	- read telemetry channel n
mode 0	- normal mode (= default)
mode 1	- chopping mode (external trigger)
fotest	- continuous fiber optic test (interrupt with space)
seqchk <number of addresses>	- continuous sequencer SRAM test (interrupt with space)

9.2.3 MACRO FILES

The interactive mode allows the usage of macro-files. These contain a sequence of commands to be exectued by typing ***ldmacro*** <filename>.

The macro file should look like:

```
ldv ../VOLTAGE/myVoltages.v
ldclk ../CLK/myClocks.clk
ldseq ../SEQ/mySeq.seq
```

Predefined macros (in the ***<iraceTest>/TEST/*** directory):

sb1.cmd	-	Santa Barbara uncorrelated
sb2.cmd	-	Santa Barbara double correlated
r1.cmd	-	Rockwell uncorrelated
r2.cmd	-	Rockwell double correlated

test.cmd - Test sequence as template, which produces 256x256 convert pulses (assuming a 4-channel system). The number of convert pulses can be set:

>pset var DET.NX <value>
>pset var DET.NY <value>
>ldseq ../SEQ/test.seq (or just <psync>)

9.2.4 SEQUENCE FILE

The sequence file has a similar structure as the sequencer program file but uses a reduced syntax:

```
ASSIGN myFrame "../SSD/myFrame.ssd"
ASSIGN myDelay "../SSD/myDelay.ssd"
ASSIGN myReset "../SSD/myReset.ssd"

LOOP <loop-counter>
myReset 1
myDelay <repetition factor>
myFrame 1
END

PROG.END
```

9.2.5 DIRECTORY STRUCTURE

```
<iraceTest>/SYS/*.btl   - bootables
<iraceTest>/VOLTAGE/    - voltage files
<iraceTest>/CLK/        - clock patterns
<iraceTest>/SSD/        - subpattern dispatcher files
<iraceTest>/SEQ/        - readout sequences
<iraceTest>/TEST/       - test macros (.cmd files)
```

10 REFERENCE

10.1 COMMAND DEFINITION TABLE (CDT)

```
//*************************************************************************
// E.S.O. - VLT project
//
// "@(#) $Id: iracqServer.cdt,v 1.54+ 2000/12/20 16:43:00 vltsccm Exp $"
//
// iracqServer.cdt
//
// Command Definition Table for the IR Acquisition Server.
//
// who       when      what
// --------  --------  --------------------------------------------
// jstegmei  18/08/99  created
//--------------------------------------------------------------------------

PUBLIC_COMMANDS

COMMAND=                     ABORT
FORMAT=                      A
PARAMETERS=
        // not implemented
        PAR_NAME=            expoId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT=
Abort the exposure currently active.
@

COMMAND=                     CLDC
FORMAT=                      A
PARAMETERS=
        PAR_NAME=            setVolt
        PAR_TYPE=            REAL
        PAR_OPTIONAL=        YES
        PAR_REPETITION_FACTOR=2

        PAR_NAME=            enable
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        PAR_NAME=            disable
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        PAR_NAME=            readVolt
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES

        PAR_NAME=            readAllVoltages
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        PAR_NAME=            board
```

```
        PAR_TYPE=           INTEGER
        PAR_OPTIONAL=       YES

        PAR_NAME=           checkAll
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

REPLY_FORMAT=               A
HELP_TEXT=
Interact with the IRACE CLDC.
@

COMMAND=                    COMMENT
FORMAT=                     A
PARAMETERS=
        PAR_NAME=           string
        PAR_TYPE=           STRING
        PAR_MAX_REPETITION= 999

        PAR_NAME=           addToNextExposure
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        PAR_NAME=           addToLastFile
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        PAR_NAME=           addToNextFile
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        PAR_NAME=           addToFile
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES

        PAR_NAME=           clear
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES
REPLY_FORMAT =              A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT =
Add comment to a FITS-header.
@

COMMAND=                    END
FORMAT=                     A
PARAMETERS=
        // not implemented
        PAR_NAME=           expoId
        PAR_TYPE=           INTEGER
        PAR_OPTIONAL=       YES

REPLY_FORMAT=               A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT=
Make DCS terminate the current exposure as quickly as possible.
@
```

```
COMMAND=                    EXIT
FORMAT=                     B
REPLY_FORMAT=               A
HELP_TEXT=
Make the application exit/terminate.
@

COMMAND=                    FRAME
FORMAT=                     A
PARAMETERS=
        // generate
        PAR_NAME=           gen
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES

        // store
        PAR_NAME=           store
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES

        // break condition
        PAR_NAME=           break
        PAR_TYPE=           INTEGER
        PAR_OPTIONAL=       YES

        // display
        PAR_NAME=           display
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        // enable or disable window parameters
        PAR_NAME=           win
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES

        // frame name
        PAR_NAME=           name
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       NO

        // detector index
        PAR_NAME=           det
        PAR_TYPE=           INTEGER
        PAR_OPTIONAL=       YES

        // add new type
        PAR_NAME=           add
        PAR_TYPE=           INTEGER
        PAR_OPTIONAL=       YES

        // parameter to enable/disable frame generation
        PAR_NAME=           param
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES

REPLY_FORMAT =              A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT =
Various services.
```

```
@

COMMAND=                    IMGOP
FORMAT=                     A
PARAMETERS=
        // enable
        PAR_NAME=           enable
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        // disable
        PAR_NAME=           disable
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        // rotate
        PAR_NAME=           rotate
        PAR_TYPE=           INTEGER
        PAR_OPTIONAL=       YES

        // flip-x
        PAR_NAME=           fx
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        // flip-y
        PAR_NAME=           fy
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

        // clear
        PAR_NAME=           clear
        PAR_TYPE=           LOGICAL
        PAR_OPTIONAL=       YES

REPLY_FORMAT =              A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT =
Various services.
@

COMMAND=                    IRACE
FORMAT=                     B
REPLY_FORMAT=               A
HELP_TEXT=
Handle an IRACE specific command. This is a
low-level interface to all IRACE-functions.
@

COMMAND=                    MISC
FORMAT=                     A
PARAMETERS=
        // Add a FITS-keyword (this will automatically appear
        // in all FITS-files when an exposure is started)
        PAR_NAME=           addFitsKey
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES
        PAR_MAX_REPETITION= 3

        // Clear all FITS-keywords added with 'addFitsKey'
```

```
        PAR_NAME=               clearFits
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

        // Dump the FITS-information into a file
        PAR_NAME=               dumpFitsInfo
        PAR_TYPE=               STRING
        PAR_OPTIONAL=           YES

        // start status polling
        PAR_NAME=               pollStart
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

        // stop status polling
        PAR_NAME=               pollStop
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

        // start perfomance polling
        PAR_NAME=               perfStart
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

        // stop perfomance polling
        PAR_NAME=               perfStop
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

        // detector index
        PAR_NAME=               det
        PAR_TYPE=               INTEGER
        PAR_OPTIONAL=           YES

        // add dynamic parameter
        PAR_NAME=               addParam
        PAR_TYPE=               STRING
        PAR_OPTIONAL=           YES
        PAR_MAX_REPETITION=     3

        // clear dynamic parameters
        PAR_NAME=               clearParam
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

        // reset acquisition process loop
        PAR_NAME=               rstcnt
        PAR_TYPE=               LOGICAL
        PAR_OPTIONAL=           YES

REPLY_FORMAT =                  A
HELP_TEXT =
Various services.
@

COMMAND=                        NC
FORMAT=                         A
PARAMETERS=
        PAR_NAME=               usrCmd
        PAR_TYPE=               STRING
        PAR_OPTIONAL=           YES
REPLY_FORMAT =                  A
HELP_TEXT =
```

```
Commands to interact with the IRACE Number Cruncher.
@

COMMAND=                    OFF
FORMAT=                     A
PARAMETERS=
REPLY_FORMAT=               A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT=
Shut-down the system - make all processes terminate.
@

COMMAND=                    ONLINE
FORMAT=                     A
PARAMETERS=
        PAR_NAME=           detConfig
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES

        PAR_NAME=           dcsConfig
        PAR_TYPE=           STRING
        PAR_OPTIONAL=       YES
REPLY_FORMAT=               A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT=
Initialize DCS with the given Detector Configuration File
@

COMMAND=                    PING
FORMAT=                     A
PARAMETERS=
REPLY_FORMAT=               A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT=
Make a check of the functioning of the application and send back an
overall status message.
@

COMMAND=                    RESET
FORMAT=                     A
PARAMETERS=
REPLY_FORMAT=               A
REPLY_PARAMETERS=
        PAR_NAME=           done
        PAR_TYPE=           STRING
        PAR_DEF_VAL=        "OK"
HELP_TEXT=
Reset the IRACE front-end.
@

COMMAND=                    SEQ
FORMAT=                     A
PARAMETERS=
        PAR_NAME=           stop
```

```
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        PAR_NAME=            start
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT=
Interact with the IRACE Sequencer
@

COMMAND=                     SETUP
FORMAT=                      A
PARAMETERS=
        // not implemented
        PAR_NAME=            expoId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES

        // not implemented
        PAR_NAME=            file
        PAR_TYPE=            STRING
        PAR_OPTIONAL=        YES
        PAR_MAX_REPETITION=  999

        PAR_NAME=            function
        PAR_TYPE=            STRING
        PAR_OPTIONAL=        YES
        PAR_MAX_REPETITION=  999

        // not implemented
        PAR_NAME=            noMove
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        // not implemented
        PAR_NAME=            check
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        PAR_NAME=            default
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

REPLY_FORMAT =               A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT =
Setup the functions as listed. The default flag sets
paramter to their default values as specified in the
parameter default setup file.
@

COMMAND=                     SIMULAT
FORMAT=                      A
PARAMETERS=
        // not implemented
```

```
        PAR_NAME=            detId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES

        // not implemented
        PAR_NAME=            function
        PAR_TYPE=            STRING
        PAR_OPTIONAL=        YES
        PAR_MAX_REPETITION=  999
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT=
Change to operation in Simulation Mode.
@

COMMAND=                     STANDBY
FORMAT=                      A
PARAMETERS=
        PAR_NAME=            haltIrace
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT=
Bring the system to Stand-By State. If the haltIrace flag is
set, the IRACE front-end is reset.
@

COMMAND=                     START
FORMAT=                      A
PARAMETERS=
        // not implemented
        PAR_NAME=            expoId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES

        // not implemented
        PAR_NAME=            at
        PAR_TYPE=            STRING
        PAR_OPTIONAL=        YES
        PAR_DEF_VAL=         "now"

        PAR_NAME=            ignoreCldc
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES

        PAR_NAME=            noNcReset
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT=
Start new exposure
@
```

```
COMMAND=                     STATUS
FORMAT=                      A
PARAMETERS=
        // not implemented
        PAR_NAME=            expoId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES

        PAR_NAME=            function
        PAR_TYPE=            STRING
        PAR_OPTIONAL=        YES
        PAR_MAX_REPETITION=  999

        // not implemented
        PAR_NAME=            global
        PAR_TYPE=            LOGICAL
        PAR_OPTIONAL=        YES
REPLY_FORMAT =               A
HELP_TEXT =
Get status for various functions.
@

COMMAND=                     STOPSIM
FORMAT=                      A
PARAMETERS=
        // not implemented
        PAR_NAME=            detId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES

        // not implemented
        PAR_NAME=            function
        PAR_TYPE=            STRING
        PAR_OPTIONAL=        YES
        PAR_MAX_REPETITION=  999
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            done
        PAR_TYPE=            STRING
        PAR_DEF_VAL=         "OK"
HELP_TEXT=
Change to Normal Operation Mode.
@

COMMAND=                     VERSION
FORMAT=                      A
PARAMETERS=
REPLY_FORMAT=                A
HELP_TEXT=
Return the present version of IR DCS.
@

COMMAND=                     WAIT
FORMAT=                      A
PARAMETERS=
        // not implemented
        PAR_NAME=            expoId
        PAR_TYPE=            INTEGER
        PAR_OPTIONAL=        YES
REPLY_FORMAT=                A
REPLY_PARAMETERS=
        PAR_NAME=            expStatus
```

```
        PAR_TYPE=            INTEGER
        PAR_DEF_VAL=         0
HELP_TEXT=
Wait for current exposure to finish. The command immediately
returns an intermediate reply indicating the current exposure
status. The last reply is sent, when the exposure has finished.
@

MAINTENANCE_COMMANDS

TEST_COMMANDS

// --- oOo ---
```

10.2 DATABASE

```
//
// IRACE acquisition class
//
CLASS BASE_CLASS iracqIRACE
BEGIN
    //
    // sequencer
    //

    // sequencer status
    ATTRIBUTE rtBYTES32    seqStatusN         "UNKNOWN"
    ATTRIBUTE rtINT32      seqStatus         -1

    // sequencer mode
    ATTRIBUTE rtINT32      seqMode            0

    // sequencer global readspeed (multiplier and add)
    ATTRIBUTE rtINT32      rspeedFactor       1
    ATTRIBUTE rtINT32      rspeedAdd          0


    //
    // CLDC
    //

    // number of CLDC-boards in system
    ATTRIBUTE rtINT32      numCldc            1

    // current CLDC-board number
    ATTRIBUTE rtINT32      cldcBoard          0

    // status of current CLDC-board
    ATTRIBUTE rtBYTES32    cldcStatusN        "UNKNOWN"
    ATTRIBUTE rtINT32      cldcStatus        -1

    // clock-voltage settings of current CLDC-board (Volt)
    ATTRIBUTE Table        clkVolt(16,
                                   rtBYTES64 nameLow,
                                   rtBYTES64 nameHigh,
                                   rtFLOAT   voltageLow,
                                   rtFLOAT   voltageHigh,
                                   rtINT32   dacChLow,
                                   rtINT32   dacChHigh,
                                   rtINT32   telChLow,
                                   rtINT32   telChHigh,
                                   rtFLOAT   range1Low,
                                   rtFLOAT   range1High,
                                   rtFLOAT   range2Low,
                                   rtFLOAT   range2High)

    // DC-voltage settings of current CLDC-board (Volt)
    ATTRIBUTE Table        dcVolt(16,
                                  rtBYTES64 name,
                                  rtFLOAT   voltage,
                                  rtINT32   dacCh,
                                  rtINT32   telCh,
                                  rtFLOAT   range1,
                                  rtFLOAT   range2)
```

```
//
// ADC
//

// number of ADC-boards in system
ATTRIBUTE rtINT32      numAdc            1

// ADC status
ATTRIBUTE Table        adcStatus(16,
                                 rtBYTES64 name,
                                 rtINT32   address,
                                 rtINT32   header,
                                 rtINT32   enable,
                                 rtINT32   filter1,
                                 rtINT32   filter2,
                                 rtINT32   delay,
                                 rtINT32   samp)


//
// readout mode
//

// acquisition status
ATTRIBUTE rtBYTES32    acqStatusN         "UNKNOWN"
ATTRIBUTE rtINT32      acqStatus         -1
ATTRIBUTE rtFLOAT      perf               0.0

// offest
ATTRIBUTE Vector       offset            (2, rtFLOAT)

// readout mode
ATTRIBUTE rtBYTES256   readoutMode        ""

// NCORRS identifier for readout mode
ATTRIBUTE rtINT32      NCORRS             2

// acquisition process name
ATTRIBUTE rtBYTES256   acqProcName        ""

// number of readout modes
ATTRIBUTE rtINT32      numRm              ""

// table containing name and id of all specified readout modes
ATTRIBUTE Table        rmDef(32,
                             rtBYTES64  name,
                             rtINT32    ncorr)

// table containing names and attributes of all available frames
ATTRIBUTE Table        frames(32,
                              rtBYTES64  name,
                              rtINT32    generate,
                              rtINT32    store,
                              rtINT32    disp,
                              rtINT32    break)

// table containing the dynamic parameters
ATTRIBUTE Table        dynPars(100,
                               rtBYTES32  name,
                               rtBYTES32  value)
```

```
    //
    // system parameters
    //

    // system status
    ATTRIBUTE rtBYTES32    stateN          "OFF"
    ATTRIBUTE rtINT32      state           1
    ATTRIBUTE rtBYTES32    subStateN       "IDLE"
    ATTRIBUTE rtINT32      subState        1

    // version string
    ATTRIBUTE rtBYTES256   version         ""

    // current operation mode
    ATTRIBUTE rtBYTES64    opModeN         "REAL"
    ATTRIBUTE rtINT32      opMode          1

    // current user
    ATTRIBUTE rtBYTES64    insUser         "SYSTEM"

    // current action for action log
    ATTRIBUTE rtBYTES256   currentAction   ""

    // update (containing originator name)
    ATTRIBUTE rtBYTES64    update          ""

    // alarm
    ATTRIBUTE rtBYTES256   alarm           ""


    //
    //  configuration filenames
    //

    // configuration path
    ATTRIBUTE rtBYTES256   configPath      ""

    // detector configuration file
    ATTRIBUTE rtBYTES256   detConfigFile   ""

    // clock pattern file from detector configuration
    ATTRIBUTE rtBYTES256   detClkFile      ""

    // CLDC-board specified in detector configuration
    ATTRIBUTE rtINT32      detCldcBoard    0

    // voltage file currently used in system
    ATTRIBUTE rtBYTES256   sysVoltFile     ""

    // clock pattern file currently used in system
    ATTRIBUTE rtBYTES256   sysClkFile      ""

    // sequencer program file currently used in system
    ATTRIBUTE rtBYTES256   sysSeqFile      ""

END
```

```
//
// RTD interface class
//
CLASS BASE_CLASS iracqRTD
BEGIN
    // internal RTD-channel
    ATTRIBUTE rtINT32      rtdInt          0

    // RTD display name
    ATTRIBUTE rtBYTES64    rtdDisp         ""

    // RTD host name
    ATTRIBUTE rtBYTES64    rtdHost         ""

    // RTD host name
    ATTRIBUTE rtBYTES128   rtdCmd          ""
END


//
// exposure class
//
CLASS BASE_CLASS iracqEXPOSURE
BEGIN

    //
    //  exposure parameters
    //

    // detector integration time (s)
    ATTRIBUTE rtFLOAT      DIT                1.0

    // minimum detector integration time (s)
    ATTRIBUTE rtFLOAT      MINDIT             0.0

    // NDIT * DIT
    ATTRIBUTE rtFLOAT      INT                1.0

    // number of integrations per INT
    ATTRIBUTE rtINT32      NDIT               1

    // number of integrations to skip at the beginning of INT
    ATTRIBUTE rtINT32      NDITSKIP           0

    // number of non-destructive samples per integration
    ATTRIBUTE rtINT32      NDSAMPLES          2

    // number of non-destructive samples to skip at start of integration
    ATTRIBUTE rtINT32      NDSKIP             0

    // window
    ATTRIBUTE rtFLOAT      STARTX             1.0
    ATTRIBUTE rtFLOAT      STARTY             1.0
    ATTRIBUTE rtINT32      NX              1024
    ATTRIBUTE rtINT32      NY              1024
    ATTRIBUTE rtINT32      STEPX              1
    ATTRIBUTE rtINT32      STEPY              1
    ATTRIBUTE rtINT32      PARTNX          1024
    ATTRIBUTE rtINT32      PARTNY          1024
    ATTRIBUTE rtINT32      adjustMode         0
```

```
    // flag to use hardware-window instead of software-window
    ATTRIBUTE rtINT32      hwWindow           0

    // indicates if the readout mode supports HW windowing
    ATTRIBUTE rtINT32      hwWindowAllowed    0

    // exposure time (s)
    ATTRIBUTE rtFLOAT      expTime            0.0

    // exposure status
    ATTRIBUTE rtBYTES32    expStatusN         "INACTIVE"
    ATTRIBUTE rtINT32      expStatus          0
    ATTRIBUTE rtINT32      dtStatus           0

    // exposure number
    ATTRIBUTE rtINT32      expNo              0

    // exposure countdown (seconds)
    ATTRIBUTE rtINT32      expCountDown       0

    // reset number cruncher counters at start command (0 = reset)
    ATTRIBUTE rtINT32      noNcReset          0

    // sequencer continuous mode flag
    ATTRIBUTE rtINT32      seqCont            0

    //
    //  FITS header
    //

    // extended FITS-header
    ATTRIBUTE rtINT32      extendedFitsHeader 1

    // FITS blocks to reserve for the header
    ATTRIBUTE rtINT32      noOfFitsBlocks     8

    //
    //  file naming
    //

    // exposure sequence base name
    ATTRIBUTE rtBYTES256   seqBaseName        ""

    // requested name (without path)
    ATTRIBUTE rtBYTES64    reqFileName        ""

    // naming type used (default request-naming)
    ATTRIBUTE rtINT32      dataFileNamingType 1

    // sequence Naming Index - index to generate names of sequences of files:
    // name generated as: <seqBaseName><seqNamingIndex>.fits
    ATTRIBUTE rtINT32      seqNamingIndex     0

    // indicates if data should be stored in a FITS data-cube (1 = store)
    ATTRIBUTE rtINT32      genDataCube        0

    // new data file
    ATTRIBUTE rtBYTES256   newDataFileName    ""
END
```

```
//
// detector class
//
CLASS BASE_CLASS iracqDETECTOR
BEGIN
    //
    // detector
    //

    // list of CLDC-boards for current detector
    ATTRIBUTE rtBYTES64    cldcList        ""

    // detector name
    ATTRIBUTE rtBYTES64    NAME            ""

    // detector type
    ATTRIBUTE rtBYTES64    TYPE            ""

    // detector number
    ATTRIBUTE rtINT32      NO              0

    // ESO ID of the detector
    ATTRIBUTE rtBYTES64    ID              ""

    // pixel-to-pixel space of the detector
    ATTRIBUTE rtFLOAT      PXSPACE         0.0

    // chip-size
    ATTRIBUTE rtINT32      NX              1024
    ATTRIBUTE rtINT32      NY              1024

    // detector mode
    ATTRIBUTE rtBYTES64    detModeN        ""
    ATTRIBUTE rtINT32      detMode         0
END


//
// chopper class
//
CLASS BASE_CLASS iracqCHOPPER
BEGIN
    //
    // chopper
    //

    // chopper state
    ATTRIBUTE rtINT32      state           0

    // frequency
    ATTRIBUTE rtFLOAT      freq            0.0

    // transition time
    ATTRIBUTE rtFLOAT      transTime       0.0
END
```

10.3 iracqEVH CLASS

NAME

iracqEVH - basic class for infrared data acquisition applications

SYNOPSIS

```
#include <iracqEVH.h>

iracqEVH myServer;
```

PARENT CLASS

iracqEVH: public evhTASK, private iracqSRV

DESCRIPTION

Basic class for infrared data acquisition applications

PUBLIC METHODS

```
iracqEVH();
    Constructor method

virtual ~iracqEVH();
    Destructor method

int InitTest();
    Returns 1, if server just does initialization test. Otherwise
    zero is returned.

const char *Instance();
    Returns the server instance.

void PrintUsage(const char *argv0);
    Prints out all server command line options.

ccsCOMPL_STAT ParseInputPars(int argc, char *argv[]);
    Parses the server command line.

ccsCOMPL_STAT Init();
    Initializes the control server.
    The method is intended to be overwritten by the application.
    It is recommended to call iracqEVH::Init first and to add
    then your intitialization.

ccsCOMPL_STAT AbortServer();
    Aborts the server and cleans up file structures. After a
    successfull Init() this function should be called before calling
    the Init() method again.

void Verbose(const char *format, ...);
    Verbose output handler. This function is called from
    all methods, when the macro iracqVB(...) is used.
    The method is intended to be overwritten by the application.
    It is recommended to call iracqEVH::VbHandler first and to
    add then your own handler.

void StateHandler(int state, int subState);
    Asynchronous state handler. This functions is called
    whenever the server state or sub-state change.
    The method is intended to be overwritten by the application.
    It is recommended to call iracqEVH::StateHandler first and to
    add then your own handler.

void iracqDataHandler(iracqDATA_EVT *evt);
    Method to handle asynchronous events issued by the data
    transfer task. The iracqDATA_EVT structure contains the
    following elements:

      int   errorFlag;      - indicates an error event
      char  fileName[256];  - contains the name of the file
```

```
                              which has been stored by the
                              data transfer task. Only contains
                              valid data, if errorFlag is not set.
      char  erms[256];      - contains an error message when
                              the errorFlag is set

    The method is intended to be overwritten by the application.
    It is recommended to call iracqEVH::DataHandler first and to
    add then your own handler.

void ExitHandler(int sig);
    Method to handle exit signals.
    The method is intended to be overwritten by the application.
    It is recommended to add first your own cleanup functions and
    then to call iracqEVH::ExitHandler.

virtual ccsCOMPL_STAT SeqHandler(const char *seqFile,
                                 vltINT32   *loopCmd,
                                 vltINT32   *loopCmdLen);
    This method can be overloaded to use your own parser/compiler
    for the sequencer program loop. The SeqHandler method is called
    by the server only if the file-extension does not match any
    supported format (.seq, .prg, tcl-shell).

vltLOGICAL ExpActive();
    Returns TRUE, if the exposurse is in an actives states. Otherwise
    FALSE is returned.

int ExpStatus();
    Returns the current exposure status. This can be one of the
    following:

        iracqEXP_UNDEFINED
        iracqEXP_INACTIVE
        iracqEXP_PENDING
        iracqEXP_INTEGRATING
        iracqEXP_PAUSED
        iracqEXP_READING_OUT
        iracqEXP_PROCESSING
        iracqEXP_TRANSFERRING
        iracqEXP_COMPL_SUCCESS
        iracqEXP_COMPL_FAILURE
        iracqEXP_COMPL_ABORTED

ccsCOMPL_STAT Param2Db(const char         *paramName,
                       const dbSYMADDRESS  pointName,
                       const dbATTRIBUTE   attrName);
    Traces the value of dynamic parameter with name <paramName>
    in the data base attribute <attrName>.

ccsCOMPL_STAT Alarm(const char *errString);
    Issues a data-base alarm.

ccsCOMPL_STAT ReplaceEnv(char *string);
    Replaces all environment variables in <string> by their
    actual values.

void EnterCB(msgMESSAGE &msg,
             vltINT32    updateSys = iracqNO_UPDATE);
    Method to be called when entering a callback that uses
    low-level irace commands.

void ExitCB();
    Method to be called when leaving a callback that uses
    low-level irace commands.

void ExitCB(msgMESSAGE &msg);
    Method to be called when leaving a callback that uses
    low-level irace commands. This will also send back a
    reply message.
```

```
ccsCOMPL_STAT IraceCmd(const char *cmd, const char *orgName);
    Method to call a low-level irace command

virtual ccsCOMPL_STAT SetParam(const char *pname,
                               const char *value,
                               vltLOGICAL *handled);
    Parameter setup function to be overloaded by an
    application specific method. If the parameter has been
    handled, the <handled> flag has to be set to TRUE.

ccsCOMPL_STAT GetParam(const char *pname,
                       char       *value,
                       vltLOGICAL *handled);
    Parameter status function to be overloaded by an
    application specific method. If the parameter has been
    handled, the <handled> flag has to be set to TRUE.

ccsCOMPL_STAT GetParam(const char *pname,
                       char       *value,
                       vltINT32   *type,
                       vltLOGICAL *handled);
    Additionally returns the parameter type.
    Type can be one of the following:

       iracqTYPE_INT     - 32 bit signed integer
       iracqTYPE_FLOAT   - 32 bit floating point
       iracqTYPE_DOUBLE  - 64 bit floating point
       iracqTYPE_STRING  - string
       iracqTYPE_LOGICAL - logical value ('T', 'F');

ccsCOMPL_STAT SetSysParam(char      **param,
                          vltINT32    size)
    Does a setup for all keywords in the parameter list pointed
    to by param. Size is the number of string values in the list
    and must be a multiple of 2. <param> should have the following
    format:

    <param1Name> <param1Value> <param2Name> <param2Value>...

ccsCOMPL_STAT SetSysParam(const char *keyWord,
                          const char *keyVal,
                          vltLOGICAL *handled);
    Sets system parameter. If the keyWord was valid, the <handled> flag
    is set to TRUE.

ccsCOMPL_STAT SetSysParam(char *paramList)
    Applies a dynamic parameter list. <paramList> must have the
    following format:

    <param1Name> <param1Value> <param2Name> <param2Value>...

ccsCOMPL_STAT GetSysParam(const char *pname, char *value)
    Gets the value of a system parameter.

ccsCOMPL_STAT GetSysParam(const char *pname,
                          char       *value,
                          vltINT32   *type)
    Additionally returns the parameter type.

ccsCOMPL_STAT ParamDefault();
    Sets all parameters back to their default values defined
    in the .dsup file.

ccsCOMPL_STAT ParamStore();
    Stores the current parameter setup.

ccsCOMPL_STAT ParamRestore();
    Restores the parameter setup stored with ParamStore().
```

```
ccsCOMPL_STAT FrameDefault(const char  *frameName,
                           vltLOGICAL   gen,
                           vltLOGICAL   store,
                           vltLOGICAL   win);
<gen> and <store> are the initial values to indicate whether the
frame should be generated/stored. If win is set to TRUE the
(sw-)window parameters are applied to the frame.

ccsCOMPL_STAT AddFrame(vltINT32     type,
                       const char  *frameName,
                       vltLOGICAL   gen,
                       vltLOGICAL   store,
                       vltLOGICAL   win,
                       char        *paramName);
    Adds a new frame type. <gen> and <store> are the initial values
    to indicate whether the frame should be generated/stored. If win
    is set to TRUE the (sw-)window parameters are applied to the frame.
    <paramName> contains the plug-in parameter associated with
    the generation of the frame type. If this is an empty string (""),
    it is assumed that the frame is always generated.

ccsCOMPL_STAT BreakCnt(const char *frameName, vltINT32 cnt);
    Apply an exposure break-condition for the frame <frameName>
    after <cnt> frames of that type have been stored by the
    data transfer task. Only frames which are stored can have
    a break-condition.

ccsCOMPL_STAT FrameGen(const char *frameName, vltLOGICAL flag);
    Enable/disable generation of a frame specified by name.

ccsCOMPL_STAT FrameStore(const char *frameName, vltLOGICAL flag);
    Enable/disable storage of a frame specified by name.

ccsCOMPL_STAT ClearClkp();
    Clears all clock patterns. This should be called before downloading
    a new set of clock-pattern loops.
    Only relevant, if the application provides its own sequencer
    program parser/compiler.

ccsCOMPL_STAT LoadClkp(vltINT32  id,
                       vltINT32 *loopCmd,
                       vltINT32  loopCmdLen);
    Downloads a clock-pattern loop. Only relevant, if the
    application provides its own sequencer program parser/compiler.

ccsCOMPL_STAT LoadSeqProg(vltINT32 *loopCmd,
                          vltINT32  loopCmdLen);
    Downloads a sequencer program loop. Only relevant, if the
    application provides its own sequencer program parser/compiler.

ccsCOMPL_STAT FrameTime(vltINT32  id,
                        vltFLOAT *frameTime);
    Returns the readout time of the clock-pattern with id <id>. If
    <id> is less than zero, <frameTime> is assumed to point to an
    array of frame-times, which is then updated with the readout
    times of all clock-patterns.

ccsCOMPL_STAT SeqMode(vltINT32 mode);
    Set the sequencer operational mode. Valid modes are:

    tcomISEQ_MOD_CLK       - normal running mode
    tcomISEQ_MOD_TRIG_EXT  - external trigger mode
    tcomISEQ_MOD_TRIG_COM  - trigger via command link
                             (reserved for test purposes)

ccsCOMPL_STAT AddFits(const char *keyword,
                      int        *var,
                      const char *commentStr);
ccsCOMPL_STAT AddFits(const char *keyword,
                      float      *var,
                      const char *commentStr);
ccsCOMPL_STAT AddFits(const char *keyword,
```

```
                      double     *var,
                      char       *commentStr);
ccsCOMPL_STAT AddFits(const char *keyword,
                      char       *var,
                      const char *commentStr);
    Trace the value of a variable in the FITS-header <keyword>.

ccsCOMPL_STAT int AddFits(const char *commentStr);
    Add a comment to the current FITS info structure.

ccsCOMPL_STAT AddFitsDynPar(const char *keyword,
                            const char *commentStr);
    Trace the value of a dynamic parameter in the FITS-header.

ccsCOMPL_STAT DownloadData(vltINT32    devId,
                           vltINT32    dataId,
                           const char *fileName);
    Download a FITS-file to the acquisition process running on
    acquisition device specified by <devId>. dataId defines the
    usage of the file and can be one of the following:

          sdmaFLATFIELD    (1)
          sdmaBADPIXMAX    (2)

    The application may define other ones, as this is passed
    unchecked to the acquisition process. In any case this has to
    be a single bit value. fileName contains the full path name of
    a FITS-file.

ccsCOMPL_STAT DownloadDataRaw(vltINT32    devId,,
                              vltINT32    dataId,
                              char       *buffer,
                              vltINT32    size);
    Download a binary buffer to the acquisition process running on
    acquisition device specified by <devId>. dataId defines the
    usage of the file and can be one of the following:

          sdmaFLATFIELD    (1)
          sdmaBADPIXMAX    (2)

    The application may define other ones, as this is passed
    unchecked to the acquisition process. In any case this has to
    be a single bit value (2^n).

ccsCOMPL_STAT ImgOp(vltINT32 op);
    Perform image post-operation on data transfer task
    before storing. Valid values for op are:

      iracqIMG_ROT90   - rotate 90 degrees clockwise
      iracqIMG_ROT180  - rotate 180 degrees
      iracqIMG_ROT270  - rotate 270 degrees clockwise
      iracqIMG_FLIPX   - mirror on y-axis
      iracqIMG_FLIPY   - mirror on x-axis

    Multiple image operations can be done by subsequent calls
    Additionally several flags can be or'ed with the above
    values to influence the application of the operations:

      iracqIMG_DISABLE - enable image operations
      iracqIMG_ENABLE  - disable image operations
      iracqIMG_CLEAR   - clear all image operations

ccsCOMPL_STAT AcqParam(const char *paramString);
    Sets an additional command line parameter string, which is added
    to the standard command line, when the IRACE acquisition process(es)
    are called.

char *AcqParam();
    Returns a pointer to the additional command line parameter
    string, which is added to the standard command line, when
```

```
    the IRACE acquisition process(es) are called.

virtual ccsCOMPL_STAT Online();
    Extension to ONLINE command. To be overloaded by an application
    specific method.

virtual ccsCOMPL_STAT Standby();
    Extension to STANDBY command. To be overloaded by an application
    specific method.

virtual ccsCOMPL_STAT Abort();
    Extension to ABORT command. To be overloaded by an application
    specific method.

ccsCOMPL_STAT UsrCmd(const char *cmd, char *reply);
    Send a user-defined command to the acquisition process.
```

RETURN VALUES

```
If the function prototype allows, methods return ccsCOMPL_STAT.
```

SEE ALSO

```
iracqSRV(4), iracqDTT(4), iracqDTT_EVH(4), evhTASK(4)
```

10.4 iracqDTT CLASS

NAME

iracqDTT - basic class for data transfer task

SYNOPSIS

```
#include <iracqDTT.h>

iracqDTT myServer;
```

DESCRIPTION

Basic class for data transfer task

PUBLIC METHODS

```
iracqDTT();
    Constructor method.

virtual ~iracqDTT();
    Destructor method.

void PrintUsage(const char *argv0);
    Prints out all server command line options.

int ParseInputPars(int argc, char *argv[]);
    Parses the server command line.

char *Instance();
    Returns the server instance.

int Init();
    Initializes the control server.
    The method is intended to be overwritten by the application.
    It is recommended to call iracqDTT::Init first and to add
    then your own intitialization.

virtual void Verbose(const char *format, ...);
    Verbose output handler. This function is called from
    all methods, when the macro iracqVB(...) is used.

virtual void ExitHandler(int sig);
    Method to handle exit signals.

char *DataPath();
    Returns current path, where data files will be stored.

char *FileName();
    Returns the proposed FITS-file name (without extension)
    depending on the selected naming scheme and the current
    exposure sequence index.

int FitsAppend(const char *keyWord,
               int         var,
               const char *comment,
               char       *erms);
int FitsAppend(const char *keyWord,
               float       var,
               const char *comment,
               char       *erms);
int FitsAppend(const char *keyWord,
               double      var,
               const char *comment,
               char       *erms);
int FitsAppend(const char *keyWord,
               const char *var,
               const char *comment,
               char       *erms);
int FitsAppend(const char *keyWord,
               const char *comment,
               char       *erms);
```

```
    Append a FITS-keyword to the local FITS-header.

int AddFits(const char *keyword,
            int         type,
            void       *addr,
            const char *comment,
            char       *erms)
int AddFits(const char *keyword,
            int        *var,
            const char *comment,
            char       *erms);
int AddFits(const char *keyword,
            float      *var,
            const char *comment,
            char       *erms);
int AddFits(const char *keyword,
            double     *var,
            char       *comment,
            char       *erms);
int AddFits(const char *keyword,
            char       *var,
            const char *comment,
            char       *erms);
    Trace the value of a variable in the FITS-header <keyword>.

int AddFits(const char *comment, char *erms);
    Add a comment to the current FITS info structure.

char *GetFitsBuf();
    Returns the global null terminated FITS-buffer
    filled by the control server. The number of FITS-cards
    in the buffer is the length of the returned string
    divided by 80.

int ExpNo();
    Returns the current exposure number.

void GetFrameName(int ftype, char *fname);
    Get the name associated with the specified frame
    type. If fname returns an empty string no name has
    been assigned to the type by the acquisition process.

char *ExpStartTime();
double *ExpStartTimeMJD();
    Returns the exposure start time.

int ExpStatus();
    Returns the current exposure status. Use the
    iracqEXP_ISACTIVE(ExpStatus()) macro to see whether the
    exposure is active or not.

char *StartTransferTime();
    Returns the time when the header of a frame was received
    from the acquisition process.

int SendFileEvent(const char *fileName, char *erms);
    Send a file-event to the control server;

int DetNum();
    Returns the current detector number/partition, which is
    selected to be stored. If 0 is returned, all detectors
    are selected.

int DetIdx();
    Returns the number/partition of the detector, which is
    actually received.

virtual int ProcessFrame(sdmaFRAME_T *frame, char *erms);
    Callback function, which is called just after a data
    frame has been received from the acquisition process.
    A pointer to the received frame structure is passed
```

```
        via <frame>. If the function returns iracqFAILURE,
        the string <erms> is sent as data error event to
        the control server and the exposure is discarded.

    void FrameHandled();
        Call this method within ProcessFrame to signal that
        all handling with the frame is done. The frame data
        will then not be object to image post-processing,
        scaling, display or storage.

    int ReplaceEnv(char *string, char *erms)
        Replaces all environment variables in string.
```

RETURN VALUES

```
    If the function prototype allows, methods return iracqSUCCESS in
    case of success. In case of failure iracqFAILURE is returned
    and erms contains an error message.
```

SEE ALSO

```
    iracqSRV(4), iracqEVH(4), iracqDTT_EVH(4)
```

10.5 iracqDTT_EVH CLASS

NAME

```
iracqDTT_EVH - basic class for data transfer task
```

SYNOPSIS

```
#include <iracqDTT_EVH.h>

iracqDTT_EVH myServer;
```

PARENT CLASS

```
iracqDTT_EVH: public evhTASK, public iracqDTT
```

DESCRIPTION

```
Basic class for data transfer task
```

PUBLIC METHODS

```
iracqDTT_EVH();
    Constructor method

virtual ~iracqDTT_EVH();
    Destructor method

void PrintUsage(const char *argv0);
    Prints out all server command line options.

ccsCOMPL_STAT ParseInputPars(int argc, char *argv[]);
    Parses the server command line.

ccsCOMPL_STAT Init();
    Initializes the control server.
    The method is intended to be overwritten by the application.
    It is recommended to call iracqDTT_EVH::Init first and to add
    then your intitialization.

void Verbose(const char *format, ...);
    Verbose output handler. This function is called from
    all methods, when the macro iracqVB(...) is used.
    The method is intended to be overwritten by the application.
    It is recommended to call iracqEVH::VbHandler first and to
    add then your own handler.

void ExitHandler(int sig);
    Method to handle exit signals.
    The method is intended to be overwritten by the application.
    It is recommended to add first your own cleanup functions and
    then to call iracqDTT_EVH::ExitHandler.

ccsCOMPL_STAT FitsAppend(const char *keyWord,
                         int         var,
                         const char *comment);
ccsCOMPL_STAT FitsAppend(const char *keyWord,
                         float       var,
                         const char *comment);
ccsCOMPL_STAT FitsAppend(const char *keyWord,
                         double      var,
                         const char *comment);
ccsCOMPL_STAT FitsAppend(const char *keyWord,
                         const char *var,
                         const char *comment);
ccsCOMPL_STAT FitsAppend(const char *keyWord,
                         const char *comment);
    Append a FITS-keyword to the local FITS-header.

ccsCOMPL_STAT AddFits(const char *keyword,
                      int        *var,
                      const char *comment);
ccsCOMPL_STAT AddFits(const char *keyword,
                      float      *var,
```

```
                      const char *comment);
ccsCOMPL_STAT AddFits(const char *keyword,
                      double     *var,
                      char       *comment);
ccsCOMPL_STAT AddFits(const char *keyword,
                      char       *var,
                      const char *comment);
    Trace the value of a variable in the FITS-header <keyword>.

ccsCOMPL_STAT int AddFits(const char *comment);
    Add a comment to the current FITS info structure.

ccsCOMPL_STAT SendFileEvent(const char *fileName);
    Send a file-event to the control server;

ccsCOMPL_STAT ReplaceEnv(char *string)
    Replaces all environment variables in string.
```

RETURN VALUES

If the function prototype allows, methods return ccsCOMPL_STAT.

SEE ALSO

iracqSRV(4), iracqEVH(4), iracqDTT(4), evhTASK(4)

11 EXAMPLES

11.1 CONTROL SERVER EXTENSION

The following example for a derived server is also contained in the test directory of the ***iracq*** module:

mySERVER.h:

```
#ifndef __cplusplus
#error This is a C++ include file and cannot be used from plain C
#endif

#include "iracqEVH.h"

#define myFRAME_TYPE (sdmaFRAME_USER << 1)

class mySRV: public iracqEVH
{
  public:
  mySRV();             // constructor
  ~mySRV();            // destructor

  void                      PrintUsage(const char *);
  ccsCOMPL_STAT             ParseInputPars(int, char **);

  // verbose callback
  void                      Verbose(const char *, ...);

  // server state callback
  void                      StateHandler(int, int);

  // data transfer task event callback
  void                      DataHandler(iracqDATA_EVT *);

  // exit callback
  void                      ExitHandler(int);

  // parameter callbacks
  ccsCOMPL_STAT             SetParam(const char *, const char *, vltLOGICAL *);
  ccsCOMPL_STAT             GetParam(const char *, char *, vltINT32 *,vltLOGICAL *);

  // sequencer program handler
  ccsCOMPL_STAT             SeqHandler(const char *, vltINT32 *, vltINT32 *);

  // initialization routine
  ccsCOMPL_STAT             Init();

  // command callbacks
  evhCB_COMPL_STAT          MyCB(msgMESSAGE &msg, void *udata);

  protected:

  private:
  // callback structure
  evhOBJ_CALLBACK           cb_;
  cmdPARAM_LIST             paramList_;

  // just some dummy variables to be added to FITS-header
  int                       dummyInt_;
```

```
  float                    dummyFloat_;
  double                   dummyDouble_;
  char                     dummyString_[128];
};
```

myServer.C

```
#define _POSIX_SOURCE 1

// Uncomment this if you are using the VLT environment
#include "vltPort.h"

#include <stdlib.h>
#include <stdio.h>

#include "mySERVER.h"

static char *rcsId="@(#) $Id: myServer.C,v 1.54+ 2000/12/20 16:43:03 vltsccm Exp $";
static void *use_rcsId = ((void)&use_rcsId,(void *) &rcsId);

/*
 *  control server
 */
static int serverProcess(int argc, char *argv[])
{
  mySRV  server;
  int    exitStatus = 0;

  /*
   *  parse command line arguments
   */
  if (server.ParseInputPars(argc, argv) == FAILURE)
    {
    server.PrintUsage(argv[0]);
    errResetStack();
    return (0);
    }

  /*
   *  intialize
   */
  if (server.Init() == FAILURE)
    {
    fprintf(stdout, "\nmyServer: initialization failed\n");
    errDisplay(ccsTRUE);
    errPrint();
    return (1);
    }

  if (server.InitTest())
    {
    return (0);
    }

  /*
   *  main loop
   */
  server.Verbose("\nmyServer: entering main loop and waiting for commands!\n");
  evhHandler->UseSelect(TRUE);
  exitStatus = evhHandler->MainLoop();
```

```
  return (exitStatus);
}


int main(int argc, char *argv[])
{
  ccsERROR error;
  int      exitStatus;
  char     procName[64];

  /*
   *  CCS init
   */
  memset(procName, 0, sizeof(procName));
  iracqRtapEnv(argc, argv, procName);
  if (ccsInit(procName, 0, NULL, NULL, &error) == FAILURE)
    {
    errPrint();
    exit(1);
    }

  /*
   *  call server
   */
  exitStatus = serverProcess(argc, argv);
  if (exitStatus == 1)
    {
    errCloseStack();
    }

  /*
   *  CCS exit
   */
  ccsExit();

  exit(exitStatus);
}

/*___oOo___*/
```

mySERVER.C:

```
#define _POSIX_SOURCE 1
#include "vltPort.h"

static char *rcsId="@(#) $Id: mySERVER.C,v 1.54+ 2000/12/20 16:43:04 vltsccm Exp $";
static void *use_rcsId = ((void)&use_rcsId,(void *) &rcsId);


/*
 * System Headers
 */
#include <stdlib.h>
#include <stdio.h>

/*
 * Local Headers
 */
#include "mySERVER.h"
```

```
mySRV *myPtr;

mySRV::mySRV()
{
  myPtr = this;

  /*
   *  initialize dummy value to be added to FITS-header
   */
  dummyInt_ = 55;
  dummyFloat_ = 55.5;
  dummyDouble_ = 55.5;
  strcpy(dummyString_, "dummy");

  /*
   *  initialize parameter list for callback
   */
  memset (&paramList_, '\0', sizeof(cmdPARAM_LIST));
}

mySRV::~mySRV()
{
  iracqVB ("\nmyServer: terminating...");

  cmdParamFree(&paramList_, stdErr);
}

void mySRV::PrintUsage(const char *argv0)
{
  /*
   *  print out standard options
   */
  iracqEVH::PrintUsage(argv0);

  /*
   *  print out additional options
   */
  fprintf(stdout, "\nmyOptions:");
  fprintf(stdout, "\n\n");
}

ccsCOMPL_STAT mySRV::ParseInputPars(int argc, char *argv[])
{
  ccsCOMPL_STAT retValOrig;
  ccsCOMPL_STAT retVal;

  /*
   *  pass arguments first to control server
   */
  retValOrig = iracqEVH::ParseInputPars(argc, argv);

  /*
   *  now parse arguments for additional options
   *  and decide, whether to return the original
   *  return value or an error
   */
  retVal = retValOrig;

  return (retVal);
}

ccsCOMPL_STAT mySRV::Init()
{
```

```
evhMSG_TYPE_KEY  key;
ErrReset();

/*
 *  intitialize control server
 */
if (iracqEVH::Init() != SUCCESS)
  {
  return (FAILURE);
  }

/*
 *  some dummy FITS-header additions
 */
if (AddFits("MY.INT", &dummyInt_, "My int") == FAILURE)
  {
  return (FAILURE);
  }
if (AddFits("MY.FLOAT", &dummyFloat_, "My float") == FAILURE)
  {
  return (FAILURE);
  }
if (AddFits("MY.DOUBLE", &dummyDouble_, "My double") == FAILURE)
  {
  return (FAILURE);
  }
if (AddFits("MY.STRING", dummyString_, "My string") == FAILURE)
  {
  return (FAILURE);
  }
if (AddFits("My comment") == FAILURE)
  {
  return (FAILURE);
  }

/*
 *  trace some dynamic parameters (if existing in read-out mode)
 */
if (AddFitsDynPar("DET.NC.MYPARAM1", "My dynamic parameter1") == FAILURE)
  {
  return (FAILURE);
  }
if (AddFitsDynPar("DET.NC.MYPARAM2", "My dynamic parameter2") == FAILURE)
  {
  return (FAILURE);
  }

/*
 *  introduce a new frame type
 */
if (AddFrame(myFRAME_TYPE, "MY-FRAME",
             FALSE, FALSE, TRUE, "DET.NC.MYFRAME") == FAILURE)
  {
  ErrAdd(iracqMOD_ID, iracqERR_INIT, __FILE_LINE__,
         "cannot add new frame");
  return (FAILURE);
  }

/*
 *  initialize additional database values
 */
iracqVB("\nmyServer: instance is %s...", Instance());
```

```
  /*
   *  add additional callbacks
   */
  iracqVB("\nmyServer: adding my callback...");
  cb_.Object(this);
  cb_.Proc((evhCB_METHOD2) &mySRV::MyCB);
  key.MsgType(msgTYPE_COMMAND);
  key.CommandId(0);
  if (evhHandler->AddCallback(key.Command("MYCMD"), cb_) == FAILURE)
    {
    AbortServer();
    ErrAdd(iracqMOD_ID, iracqERR_INIT, __FILE_LINE__,
           "unable to add MYCMD callback");
    return (FAILURE);
    }

  return (SUCCESS);
}


void mySRV::Verbose(const char *str, ...)
{

  iracqEVH::Verbose(str);

  /*
   * here you can add your own verbose handler
   */
}

void mySRV::StateHandler(int state, int subState)
{

  iracqEVH::StateHandler(state, subState);

  /*
   * here you can add your own status handler
   */

}

void mySRV::DataHandler(iracqDATA_EVT *evt)
{

  iracqEVH::DataHandler(evt);

  /*
   * here you can add your own data event handler
   */

}

void mySRV::ExitHandler(int sig)
{
  /*
   * here you can add your own exit handler
   */
  iracqVB("\nmyServer: exiting...");

  iracqEVH::ExitHandler(sig);
}

ccsCOMPL_STAT mySRV::SeqHandler(const char *seqFile,
```

```
            vltINT32 *loopCmd,
            vltINT32 *loopCmdLen)
{
  USE(loopCmd);
  USE(seqFile);

  *loopCmdLen = 0;

  ErrAdd(iracqMOD_ID, iracqERR_SYSTEM, __FILE_LINE__,
         "format of sequencer program is not supported");

  return (FAILURE);
}

ccsCOMPL_STAT mySRV::SetParam(const char *name,
                              const char *value,
                              vltLOGICAL *handled)
{
  char tmpStr[64];

  *handled = TRUE;

  if (strcmp(name, "DET.MYSETUP") == 0)
    {
    fprintf(stdout, "\nmyServer: doing MYSETUP (value = %s)...",
            value);

    sprintf(tmpStr, "-myArg %s", value);

    /*
     *  do some action
     */
    if (AcqParam(tmpStr) == FAILURE)
      {
      return (FAILURE);
      }

    fprintf(stdout, "\nmyServer: MYSETUP done...");

    /*
     *  leave 'handled' set to FALSE to apply the value also
     *  in sequence file or acq-process (if applicable)
     */
    }
  else if (strcmp(name, "DET.MYVAL") == 0)
    {
    fprintf(stdout, "\nmyServer: MYVAL = %s...",
            value);
    }
  else
    {
    *handled = FALSE;
    }

  return (SUCCESS);
}

ccsCOMPL_STAT mySRV::GetParam(const char *name,
                              char       *value,
                              vltINT32   *type,
                              vltLOGICAL *handled)
{
  USE(name);
```

```
  USE(value);

  *handled = FALSE;

  if (strcmp(name, "DET.MYVAL") == 0)
    {
    sprintf(value, "%d", dummyInt_);
    *type = iracqTYPE_INT;
    *handled = TRUE;
    }

  return (SUCCESS);
}

evhCB_COMPL_STAT mySRV::MyCB(msgMESSAGE &msg, void *)
{
  void           *paramValues;
  char          **stringVal;
  vltINT32        paramNb;
  cmdPARAM_TYPE   paramType;
  int             i;

  ErrReset();

  EnterCB(msg, iracqDO_UPDATE);

  fprintf(stdout, "\nmyServer: handling MYCMD (buffer is <%s>)...",
          msg.Buffer());

  /*
   *  unpack message buffer
   */
  if (cmdParamList("MYCMD", msg.Buffer(), msg.Buflen(), &paramList_,
                   stdErr) == FAILURE)
    {
    ExitCB(msg);
    return (evhCB_NO_DELETE);
    }

  /*
   *  -myParam
   */
  if (cmdParamGetByName(&paramList_, "myParam", &paramNb, &paramType,
                        &paramValues, stdErr) == FAILURE)
    {
    ExitCB(msg);
    return (evhCB_NO_DELETE);
    }
  if (paramNb > 0)
    {
    stringVal = (char **)paramValues;
    fprintf(stdout, "\nmyServer: myParam = ");
    for (i=0;i<paramNb;i++)
      {
      fprintf(stdout, "<%s> ", stringVal[i]);
      }
    }
  fflush(stdout);

  msg.Buffer("My-Reply");
  ExitCB(msg);

  return (evhCB_NO_DELETE);
```

```
}

/*___oOo___*/
```

11.2 DATA TRANSFER TASK EXTENSION

The following example for a derived data transfer task is also contained in the test directory of the iracq module:

myDTT.h:

```
#ifndef __cplusplus
#error This is a C++ include file and cannot be used from plain C
#endif

#include "iracqDTT_EVH.h"

class myDTT: public iracqDTT_EVH
{
  public:

  myDTT();              // constructor
  virtual ~myDTT();     // destructor

  // initialization routine
  ccsCOMPL_STAT          Init();

  // command callbacks
  evhCB_COMPL_STAT       MyCB(msgMESSAGE &msg, void *udata);

 // application specific frame processing
  int                    ProcessFrame(sdmaFRAME_T *, char *);

  protected:

  private:
  // callback structure
  evhOBJ_CALLBACK        cb_;
  cmdPARAM_LIST          paramList_;

  // just some dummy variables to be added to FITS-header
  int                    dummyInt_;
  float                  dummyFloat_;
  double                 dummyDouble_;
  char                   dummyString_[128];
};
```

myDtt.C:

```
#define _POSIX_SOURCE 1

#include "vltPort.h"

static char *rcsId="@(#) $Id: myDtt.C,v 1.12+ 1999/12/14 11:42:19 vltsccm Exp $";
static void *use_rcsId = ((void)&use_rcsId,(void *) &rcsId);

#include <stdio.h>
#include <stdlib.h>

#include "myDTT.h"

/*
 *  control server
 */
```

```
static int server(int argc, char *argv[])
{
  myDTT    server;
  int      exitStatus = 0;

  /*
   *  parse command line arguments
   */
  if (server.ParseInputPars(argc, argv) == FAILURE)
    {
    server.PrintUsage(argv[0]);
    errResetStack();
    return (0);
    }

  /*
   *  intialize
   */
  if (server.Init() == FAILURE)
    {
    fprintf(stderr, "\nmyDtt: initialization failed\n");
    errPrint();
    return (1);
    }

  /*
   *  main loop
   */
  iracqVB("\nmyDtt: entering the main loop and waiting for commands!");
  evhHandler->UseSelect(TRUE);
  exitStatus = evhHandler->MainLoop();

  return (exitStatus);
}

int main(int argc, char *argv[])
{
  ccsERROR error;
  int exitStatus;
  char     procName[64];

  /*
   *  CCS init
   */
  iracqRtapEnv(argc, argv, procName);
  if (ccsInit(procName, 0, NULL, NULL, &error) == FAILURE)
    {
    errPrint(&error);
    exit(1);
    }

  /*
   *  call server
   */
  exitStatus = server(argc, argv);
  if (exitStatus == 1)
    {
    errCloseStack();
    }

  /*
   *  CCS exit
   */
```

```
  ccsExit();

  fprintf(stderr, "\n");
  exit(exitStatus);
}


/*___oOo___*/
```

myDTT.C:

```
#ifdef HP
#define _POSIX_SOURCE 1
#endif

#include "vltPort.h"

static char *rcsId="@(#) $Id: myDTT.C,v 1.54+ 2000/12/20 16:43:04 vltsccm Exp $";
static void *use_rcsId = ((void)&use_rcsId,(void *) &rcsId);

#include <stdio.h>
#include <stdlib.h>

#include "myDTT.h"

myDTT *myPtr;

myDTT::myDTT()
{
  myPtr = this;

  /*
   *  initialize dummy value to be added to FITS-header
   */
  dummyInt_ = 65;
  dummyFloat_ = 65.5;
  dummyDouble_ = 65.5;
  strcpy(dummyString_, "dtt dummy");

  /*
   *  initialize parameter list for callback
   */
  memset (&paramList_, '\0', sizeof(cmdPARAM_LIST));
}

myDTT::~myDTT()
{
  iracqVB ("\nmyDtt: terminating...");

  cmdParamFree(&paramList_, stdErr);
}

ccsCOMPL_STAT myDTT::Init()
{
  evhMSG_TYPE_KEY key;

  if (iracqDTT_EVH::Init() != SUCCESS)
    {
    return (FAILURE);
    }
```

```
  /*
   *  add additional callbacks
   */
  iracqVB("\nmyDtt: adding my callback...");
  cb_.Object(this);
  cb_.Proc((evhCB_METHOD2) &myDTT::MyCB);
  key.MsgType(msgTYPE_COMMAND);
  key.CommandId(0);
  if (evhHandler->AddCallback(key.Command("MYCMD"), cb_) == FAILURE)
    {
    ErrAdd(iracqMOD_ID, iracqERR_INIT, __FILE_LINE__,
           "unable to add MYCMD callback");
    return (FAILURE);
    }

  /*
   *  some dummy FITS-header additions
   */
  if (AddFits("MYDTT.INT", &dummyInt_, "My int") == FAILURE)
    {
    return (FAILURE);
    }
  if (AddFits("MYDTT.FLOAT", &dummyFloat_, "My float") == FAILURE)
    {
    return (FAILURE);
    }
  if (AddFits("MYDTT.DOUBLE", &dummyDouble_, "My double") == FAILURE)
    {
    return (FAILURE);
    }
  if (AddFits("MYDTT.STRING", dummyString_, "My dtt string") == FAILURE)
    {
    return (FAILURE);
    }
  if (AddFits("My dtt comment") == FAILURE)
    {
    return (FAILURE);
    }

  return (SUCCESS);
}

evhCB_COMPL_STAT myDTT::MyCB(msgMESSAGE &msg, void *)
{
  void           *paramValues;
  char          **stringVal;
  vltINT32        paramNb;
  cmdPARAM_TYPE   paramType;
  int             i;

  ErrReset();

  fprintf(stdout, "\nmyDtt: handling MYCMD (buffer is <%s>)...",
          msg.Buffer());

  /*
   *  unpack message buffer
   */
  if (cmdParamList("MYCMD", msg.Buffer(), msg.Buflen(), &paramList_,
                   stdErr) == FAILURE)
    {
    return (evhCB_NO_DELETE);
```

```
    }

  /*
   *  -myParam
   */
  if (cmdParamGetByName(&paramList_, "myParam", &paramNb, &paramType,
                        &paramValues, stdErr) == FAILURE)
    {
    return (evhCB_NO_DELETE);
    }
  if (paramNb > 0)
    {
    stringVal = (char **)paramValues;
    fprintf(stdout, "\nmyDtt: myParam = ");
    for (i=0;i<paramNb;i++)
      {
      fprintf(stdout, "<%s> ", stringVal[i]);
      }
    }
  fflush(stdout);

  msg.LastReply(ccsTRUE);
  msg.Buffer("My-Reply");
  msg.SendReply();
  ErrStackClose();

  return (evhCB_NO_DELETE);
}

int myDTT::ProcessFrame(sdmaFRAME_T *frame, char *erms)
{
  USE(erms);
  USE(frame);

  printf("\nmyDTT: processing frame...");

  dummyInt_++;

  if (frame->h.ftype == sdmaFRAME_DIT)
    {
    FrameHandled();
    }

  return (iracqSUCCESS);
}

/*___oOo___*/
```