Real-Time Software Programmer Manual

SOAR Adaptive Module (SAM)

Revision 3.11, May 2013

Change Record

Version	Date	Description	Owner Name
3.8	January 25, 2013	Changes introduced by RTSOFT v4.1.0	RCR
3.9	March 14, 2013	Modulate command documented	RCR
3.10	March 25, 2013	REC, SETREF documentation	RCR
3.11	May 16, 2013	Reference to SDN-8410. How to build and install the device drivers. How to build and install the Labview modules shared libraries.	RCR

Índice de contenido

Chapter 1: Overview

1.1 Introduction

The SAM software suite of applications comprises a set of programs that differentiate themselves by the mission they serve [1]. The suite includes the Instrument Control Software, Imager Software, AO and LGS Modules Software and Real-Time Software.

This manual covers the implementation details of the Real-Time Software (RTSOFT). The manual is not focused on operational aspects of the software. For an operational focus please read the Real-Time Software User Manual instead.

1.2 Software Architecture

The main task of the Real-Time Computer Software (RTSOFT) is to measure atmospheric distortions and compensate them in real time, understanding real-time as the ability to respond to external events in a minimum and deterministic amount of time.

The requirements imposed [2] called for a hybrid approach. The hard real-time part of the software, represented by data acquisition and loop control, was implemented in kernel modules under an RTAI Linux environment [5]. This set of kernel modules is called the real-time core (RTCORE). The soft and non real-time part of the software, represented by data storage and data presentation, user interfaces and remote connectivity, was implemented in LabVIEW [3].

Figure 1.1 presents the architecture of the RTSOFT. Two primary layers can be identified: hardware and software. Hardware is represented by the several interface boards connecting to actual devices like DM, M3, WFS, etc. Software on the other hand can again be seen in two secondary layers. User space and Kernel space.

The RTCORE belongs to the kernel space and the LabVIEW application belongs to the user space. The two spaces normally don't see each other except through special files or shared memory. The RTCORE uses both to communicate with the LabVIEW user space application.

The block diagram shows how the RTCORE closes the DM control loop, the M3 control loop and the LLT control loop. The WFS CCD interface board writes the acquired images to memory an signals with an interrupts each time it finishes the transfer. The RTCORE is triggered by the interrupt to execute the control algorithm and actuate the DM.

Depending on the operation mode, LGS or NGS, the tilt error signal is derived from the WFS or the TT probes to actuate the M3 mirror and the telescope mount. When in LGS mode the residual error measured by the WFS is used to actuate the LLT.

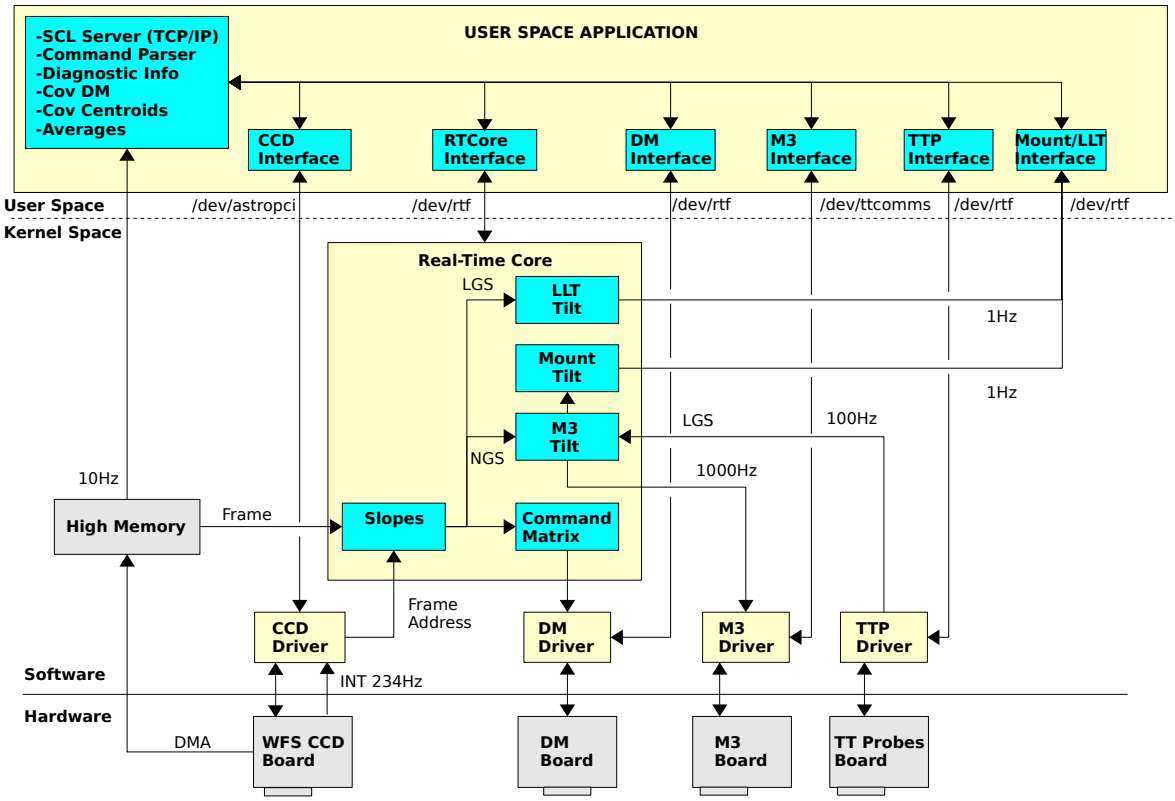

Figure 1.1: Block diagram of the Real-Time Computer Software

1.3 Source Code

The RTSOFT software can be found in the Real-Time Computer machine, installed in the ao user home directory under root directory *RTsoft*. The RTCORE related code lives under subdirectory *KModules* while the LabVIEW application code lives under subdirectory *LV2010Modules* with its main VI living alone in the root directory *RTsoft*.

Executables and shared libraries produced by the building of the LabVIEW application can be found under directory *Bin* and *Lib*. Configuration files, data files, logs and macros are found in the *ConfigFiles*, *Data*, *Logs* and *Macro* subdirectories.

Kernel modules produced by the source code in the *Kmodules* subdirectory, are installed outside the *RTsoft* directory tree in the standard */lib/modules/<kernel version>* system directory. The details can be found in the *Makefile* for each module.

Copies of the Real-Time Software are kept in the SOAR public file server. Access is possible through local accounts on machine ctioll. The path to the copies is */home/public/SOAR/SAM*.

The following is a tree view of the directory structure of the software:

```
RTsoft
|--Bin
|--Boot
|--ConfigFiles
|--Data
|--Doc
|--KModules
| |--astropciV1.7
| |--djbfft0.76
| |--ni660x
| |--pci9054
| |--powerdaq
| `--rtcore
|--Lib
|--Logs
|--LV2010Modules
|   |-- ASTROLIB
|   |-- CAMERALIB
|   |-- COMMLIB
|   |-- DISPLIB
|   |-- DMLIB
|   |-- DTLIB
|   |-- DYNVILIB
|   |-- FAULTLIB
|   |-- FITS
|   |-- HISTORYLIB
|   |-- INIFLIB
|   |-- LMLIB
|   |-- LV2010-RTAI3.8
|   |-- MEMLIB
|   |-- MULTIFLIB
|   |-- NI660X
|   |-- PARSELIB
|   |-- RTCORELIB
|   |-- RTSOFTLIB
|   |-- SCLN
|   |-- SDSU-III
|   |-- STFLIB
|   |-- TCSLIB
|   |-- TTCOMMSLIB
|   `-- UNSCRLIB
|--Macro
`--Profiles
    |-- B1x1
    `-- B2x2
```

1.4 Device Drivers

The device drivers modules for each board are packed under the *Kmodules* subdirectory. In general these are non standard versions of the device drivers tailored to work under the RTAI environment.

1.4.1 powerdaq

Use the 3.6.23 version of the powerdaq driver:

```
# cd powerdaq/3.6.23
# make
# make install
```

Edit *rc.local* and add the following lines to load the driver at boot time:

```
# Load the Power-DAQ driver: SAM DM Voltages
/sbin/modprobe pwrdaq rqstirq=0
```

1.4.2 pci9054

Use the 2.6-adeos version of the pci9054 driver:

```
# cd pci9054/2.6-adeos
# make
# make install
```

Edit *rc.local* and add the following lines to load the driver at boot time:

```
# Load the PCI9054 driver: SOAR M3 Mirror
/sbin/modprobe ttcomms
```

1.4.3 ni660x

Use the 2.6-adeos version of the ni660x driver:

```
# cd ni660x/2.6-adeos
# make
# make install
```

Edit *rc.local* and add the following lines to load the driver at boot time. Place it after loading the RTAI modules.

```
# Load the NI660x driver: SAM Pockels Cell & APD Counters
/home/ao/RTsoft/KModules/ni660x/2.6-adeos/ni660x_load
```

1.4.4 astropciv1.7

Use the 2.6-adeos version of the astropciv1.7 driver:

```
# cd astropciv1.7/2.6-adeos
# make
# make install
```

Edit *rc.local* and add the following lines to load the driver at boot time. Place it after loading the RTAI modules.

```
# Load the SDSU-III driver: SAM WFS
/home/ao/RTsoft/KModules/astropciV1.7/2.6-adeos/astropci_load
```

Chapter 2: The Real-Time Core

2.1 Introduction

The real-time core (RTCORE) is implemented as a single kernel module with upward and downward dependencies with other device driver modules (Figure 1.1). The RTCORE and the device drivers source code, are all kept together under directory *Kmodules* (see section 1.3). Requirements and design notes on the RTCORE software can be found in document [4].

The module implements basically two tasks: a bottom-half interrupt handler, driven by the interrupt handler of the camera controller driver, and a periodic task hooked to the computer timer. The bottom-half interrupt handler handles the AO loop, while the periodic task handles the TT loop. To serve the run-time configuration of the module, several data paths suitable of being accessed by a user space application, are readily available.

2.2 Source Code and Compilation

The source code of the RTCORE module is located under directory *Kmodules/rtcore/2.6-adeos* (section 1.3). The real-time computer has RTAI installed in it. At the time this document was written the Linux kernel version in use was 2.6.25 patched with RTAI 3.8 under CentOS 5.8[1].

There are conditional compilation directives that can come up handy when doing tests or trouble shooting. These directives are basically part of the *.c* files. Edit the file and uncomment the "*#define DEBUG_SOARAO*" directive to compile with debug messages enabled. Debug messages will be written to the kernel log, and with proper configuration of the kernel log daemon, they can be read from the */var/log/messages* file.

The other handy preprocessor directive removes the dependency of the RTCORE to the driver modules routines. Edit the file and comment out the line "*#define _USE_<driver name>*" to remove all references to routines in the driver module.

To compile the module become superuser and type *make*. Then do a *make install* to install the module

```
# make
# make install
```

This will produce the kernel module *rtcore.ko*. When installed, the module *rtcore.ko* is copied to the */lib/modules/2.6.25/kernel/drivers/misc* directory.

2.3 Loading the Module

The SAM Real-Time Computer (RTC) automatically loads the RTCORE module on boot time. The *rc.local* file was modified to accommodate the lines that load the RTAI base modules, device drivers

1 See document [5] for tips on how to set up an RTAI enabled environment.

and RTCORE in the right order preventing breaks in dependencies. An example of this file follows

```
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

touch /var/lock/subsys/local

# Load the PowerDAQ drivers: SAM DM Voltages
/sbin/modprobe pwrdaq rqstirq=0

# Load the PCI9054 driver: SOAR M3 Mirror
/sbin/modprobe ttcomms

# Loat the RTAI-3.8 base: Real-Time Kernel
/sbin/insmod /usr/realtime/modules/rtai_smi.ko
/sbin/insmod /usr/realtime/modules/rtai_hal.ko
/sbin/insmod /usr/realtime/modules/rtai_ksched.ko
/sbin/insmod /usr/realtime/modules/rtai_sem.ko
/sbin/insmod /usr/realtime/modules/rtai_fifos.ko
/sbin/insmod /usr/realtime/modules/rtai_shm.ko
/sbin/insmod /usr/realtime/modules/rtai_math.ko

# Load the NI660x driver: SAM Pockels Cell & APD Counters
/home/ao/RTsoft/KModules/ni660x/2.6-adeos/ni660x_load

# Load the SDSUIII driver: SAM WFS
/home/ao/RTsoft/KModules/astropciV1.7/2.6-adeos/astropci_load

# Load the RTCORE module
/home/ao/RTsoft/KModules/rtcore/2.6-adeos/rtcore_load
```

To check that the RTCORE module was properly installed use the "*lsmod*" command. The output should look as follows:

```
% /sbin/lsmod
Module          Size   Used by Tainted: P
input           4928    0 (autoclean)
astropci       11844    0
rtcore      99776    0 [astropci]
ni660x          5664    0 [ao_rtcore]
rtai_shm        6832    0 [ao_rtcore]
rtai_fifos     29868    0 [ao_rtcore]
rtai_up        62940    0 [ao_rtcore rtai_shm rtai_fifos]
rtai_hal       15924    0 [input astropci ao_rtcore ni660x rtai_shm
pwrdaq        940224    0 [ao_rtcore]
lp              7652    0 (autoclean)
parport        30464    0 (autoclean) [lp]
iptable_filter  2404    0 (autoclean) (unused)
ip_tables      13528    1 [iptable_filter]
e100           52504    1
```

```
ext3           63812    3
jbd            44168    3 [ext3]
```

To manually install and remove the module use the following script under super-user mode from a console window. To install the module:

```
# /home/ao/RTsoft/KModules/rtcore/2.6-adeos/rtcore_load
```

To remove the module:

```
# /home/ao/RTsoft/KModules/rtcore/2.6-adeos/rtcore_unload
```

2.3.1 Module Parameters

The module is capable of accepting parameters at load time to configure interface board ID numbers. The source code includes the following parameter definitions (*osLx24.c*):

```
MODULE_PARM(dmb,"2i")
MODULE_PARM(ttb,"i")
```

The *dmb* parameter accepts two integer values specifying the board ID for the two DIO boards that generate the DM voltages. The *ttb* parameter is a single integer value specifying the board ID for the DIO board the generates the M3 set-point commands[2].

For example, the following line will load the module and assign the ID number 0 and 1 to the DM DIO boards, and the ID number 2 to the TT DIO board.

```
# /sbin/insmod ao_rtcore dmb=0,1 ttb=2
```

2.4 Data Flows to User Space

The RTCORE creates several data paths when loaded. These data paths allow the LabVIEW user space application to have access to the acquired data, module status, and to configure the module behavior. Table 2.1 summarizes the available data paths and their characteristics.

Table 2.1: Data Paths to the RTCORE Kernel Module

Name	Type	Description	Size
/dev/rtf1	FIFO	Control FIFO	1024 bytes
/dev/rtf2	FIFO	Response FIFO	1024 bytes
/dev/rtf3	FIFO	TT Loop Data FIFO	5440 bytes
/dev/rtf4	FIFO	AO Loop Data FIFO	694400 bytes
CTRL_I	Shared Memory	32 bit INT Data Structure	8 bytes
CTRL_F	Shared Memory	Double Precision Data Structure	37.8 Kbytes
BANK_I	Shared Memory	16 bit UINT Artificial Pattern	12800 bytes
DARK_I	Shared Memory	16 bit UINT Bias Frame	12800 bytes
MASK_I	Shared Memory	16 bit UINT Bitwise Frame Mask	12800 bytes

2 Only if using the PDAQ DIO board.

Name	Type	Description	Size
GRID_X	Shared Memory	Double Precision Calculation Grid	50 Kbytes
GRID_Y	Shared Memory	Double Precision Calculation Grid	50 Kbytes
WEIGHT	Shared Memory	Double Precision Weights Mask	50 Kbytes
/dev/soartt	Device File	Open/Close/Write/Ioctl interface	N/A

2.4.1 Control and Response FIFO

There is a control FIFO (*/dev/rtf1*) to receive string commands and a response FIFO (*/dev/rtf2*) to send back string responses. The two are created by the *init_module()* routine in file *osLx24.c*.

The RTCORE assigns a handler to the control FIFO so that every time a user space application writes a command to this FIFO, the command gets executed and a response is sent back. The handler is implemented by the routine *sao_command_handler()* in file *osLx24.c*. The format of the message is defined by a data structure of type AO_MESSAGE

```
typedef struct {
  int len;
  char buffer[BUFSIZE];
} _AO_MESSAGE;
```

The header is the length of the string command and the buffer is the string command itself. Most of the module behavior is configured using the command list that follows

AO control loop commands. The commands are implemented by the *ao_service()* routine in the *aoLib.c* file.

aofg open - Open the AO control loop.

aofg close - Close the AO control loop.

aofg irq <*widht*> <*height*> <*binning*> - Runs the AO control loop over an artificial pattern stored in the BANK_I shared memory buffer. User specifies the geometry of the artificial pattern. With no arguments return the number of interrupts so far served by the RTCORE.

aofg reset – Reset the interrupts counter, the out of sync flag and the max latency stored value.

aofg sete <*electrode*> <*board*> <*channel*> - Set the table entry mapping the DM actuator electrode to the specified board and channel. The electrode look-up table corresponds to the element eltrd[] of the CTRL_I data structure (see section 2.4.4)

aofg eltrd <*index*> <*volts*> - Set the DM electrode *index* to the specified voltage value *volts*. The look-up table in t_ctrl_int is used to map the electrode index to a board and channel number.

aofg eltrd_adu <*board*> <*channel*> <*value*> - This is an engineering command. User

specifies the board, the channel and value in volts (10 to +10).

aofg echo - Test the aofg service. The expected response is “Hello from aofg!”.

aofg simdm [0 | 1] - Toggles the flag that enables/disables the writing to the DAC boards. When set to 1 no voltage will be written to the DAC boards.

aofg apos <*sub-aperture*> <*lx*> <*ly*> <*ux*> <*uy*> - Set the sub-aperture box position. These are the row-column coordinates in non-binned units. With no arguments other than the sub aperture id, returns the box position and size.

aofg rshift <sub-aperture> <xshit> <yshift> - Get/set the reference positions. This is the vector between the sub-aperture box center and the spots produced with the reference beam in non-binned units.

aofg di < *k* | *kl* > <*value*> - Digital integrator parameters. With no arguments this command returns a string containing three values: 0 or 1 to indicate if the control law is in use, K and KL. The difference equation representing this digital controller is $y(k) = K_L y(k-1) + Kx(k)$.

aofg sd <*sub-aperture*> - Returns the current slope measurement for the given sub-aperture, and the current flux measurement after bias and background subtraction.

aofg sp [*k* | *kl* | *ks* | *on* | *off*] - Use this command to configure the Smith digital controller. The difference equation describing the controller is $y(k) = (K_L - K_S)y(k-1) + K_S K_L y(k-2) + Kx(k)$. With no arguments this command returns a string containing four

values: 0 or 1 to indicate if the control law is in use, K, K_L and K_S

on - activates the SP digital controller.

off – deactivate the use of the SP digital controller.

k <*k*> - sets the SP controller gain.

kl <*kl*> - sets the SP leaky parameter.

ks <*ks*> - sets the SP parameter.

aofg abgnd <*sub-aperture*> – Return average background for sub aperture.

aofg etime – Return four numbers indicating the elapsed time during execution of the control loop logic, time between calculations, max time between calculations and out of sync flag. Units of nanoseconds.

aofg ns <*value*> - Set the number of useful sub-apertures in the WFS.

aofg nprobes <*value*> - Set the number of background probes.

aofg tilt [*0* | *1*] - Turn on/off the tilt subtraction flag. If active the component in the slopes corresponding to tilt (z2 and z3) will be removed.

aofg Tfx <*T11*> <*T12*> <*T21*> <*T22*> - Set the transformation matrix between WFS X-Y coordinates and Guide Probes X-Y coordinates.

Centroid library commands. The commands are implemented by the *cc_service()* routine in the *centroidLib.c* file.

cc ref4 <*amplitude*> <*sigma*> – Define a convolution pattern for using with the cross correlation algorithm. The convolution pattern is a Gaussian located at the center of the box ((2,2) for a 4x4 box). The arguments are an amplitude and sigma.

cc ref8 <*amplitude*> <*sigma*> – Define a convolution pattern for using with the cross correlation algorithm. The convolution pattern is a Gaussian located at the center of the box ((4,4) for 8x8 box). The arguments are an amplitude and sigma.

cc dark <*0* | *1*> - Turn bias subtraction on and off. The bias pattern has to be previously written to the shared memory buffer DARK_I.

cc bgnd <*0* | *1* > - Turn the background subtraction on and off.

cc abgnd – Return the average background estimation for each CCD quadrant.

cc algorithm <*WCoG = 1* | *CC = 5*> - Selects between the Weighted Center of Gravity and the Cross Correlation method.

cc m – Minimum allowable signal parameter. The readout noise is multiplied by m and then compared to the max pixel value to decide if the signal is below the acceptable value.

cc Nr – Readout noise in ADU.

cc NT – Size of the spot for the Cross Correlation method template.

cc Nw – Size of the spot for the Weighted Center of Gravity template.

Tilt control commands. The commands are implemented by the *tt_service()* routine in the *ttLib.c* file.

tt rot <*angle*> – With no arguments returns the current rotation angle in units of radians. Otherwise set the rotation correction angle to angle degrees.

tt m3 [*K* | *open* | *close* | *etime* | *enable* | *disable* | *test* | *source*] - M3 loop control commands.

KC1 <*gain*> With no arguments returns the current gain of the SOAR M3 controller integrator. Otherwise set the integrator gain to gain.

LP1 <K_{LP1}> <a_{LP1}> With no arguments return the low pass filter parameters for the LLT M3 offload. Otherwise set the parameters to the given values.

open – open the M3 control loop. The tilt command to the M3 is set to zero.

close – close the M3 control loop.

etime – return the elapsed time during last centroid calculation in units of [ns].

enable – enable integrator. Input to the controller is set to the tilt error signal from the the selected source (WFS or TT probes).

disable – disable integrator. Input to the controller is set to zero. The last tilt command to the M3 is preserved.

HstarZ *<h11> <h12> <h21> <h22>* – With no arguments return the current Zero rotation reconstruction matrix $\mathbf{H_z^*}$. Otherwise set the $\mathbf{H_z^*}$ to the given values.

Hstar – return the current tilt reconstruction matrix. This matrix translate guide probes x-y coordinates in arc-seconds to SOAR M3 el-az coordinates in [ADU] including Nasmyth rotation and mount elevation effects.

test [*gamma* | *beta* | *none*] – enable the module to receive user defined waveforms and add them to the M3 command word. Option gamma will enable elevation. Option beta will enable azimuth, and none will disable the feature.

source [*wfs* | *probes*] – select between WFS and TT probes as source of the tilt error signal.

wobble [*on* | *off* | *radius* | *period*] – handles the wobble tool parameters.

on – start calculating and adding wobble offset to the M3 command.

off – stop calculating and adding wobble offsets to the M3 command.

radius <*radius*> – set the radius of the circle described by M3 when wobbling. Units are TBD at this time.

period <*period*> – set the period when describing a complete circle in units of seconds.

tt llt [*KC3* | *open* | *close* | *enable* | *disable*] - LLT loop control commands.

KC3 <K_{C3}> With no arguments return the current gain of the LLT M3 controller integrator. Otherwise set the gain to gain.

KC4 <K_{C4}> With no arguments return the current proportional gain of the LLT

M1 controller. Otherwise set the parameter to the given value.

LP4 <K_{LP4}> <a_{LP4}> With no arguments return the low pass filter parameters for the LLT M3 offload. Otherwise set the parameters to the given values.

open – open the LLT control loop. The tilt command to the LLT is set to zero.

close – close the LLT control loop. The output of the integrator is stored in test point 4.

enable – enable integrator. Input to the controller is set to the tilt error signal from the WFS.

disable – disable integrator. Input to the controller is set to zero. The last tilt command to the LLT is preserved.

HstarZ *<h11> <h12> <h21> <h22>* – With no arguments return the current Zero rotation reconstruction matrix $\mathbf{H_z^*}$. Otherwise set the $\mathbf{H_z^*}$ to the given values.

Hstar – return the current tilt reconstruction matrix. This matrix translate guide probes x-y coordinates in arc-seconds to LLT M3 az-el coordinates in [V] including Nasmyth rotation and mount elevation effects.

offset <*EL*> <*AZ*> – add a user defined amount of offset to the analog voltages to the LLT M3 piezos. EL corresponds to electrode 60 and AZ to electrode 61. Very useful to diagnose if the RTCORE is capable of controlling the M3 PZT voltages. With no arguments returns the current offset values.

tt KC2 <*gain*> – With no arguments return the current mount controller gain. Otherwise set the current mount gain set the current mount gain to *gain*.

LP2 <K_{LP2}> <a_{LP2}> With no arguments return the low pass filter parameters for the M3 offload to the Mount. Otherwise set the parameters to the given values.

tt tp [*3* | *4*] – Return selected test-point value. See Figure TBD.

3 – Return the mount correction in GP x-y coordinates and units of arc-seconds.

4 – Return the LLT M1 correction in GP x-y coordinates and units of arc-seconds.

tt n <*base*>– With no arguments return the current time base multiplier for the TT probes sampling loop. Otherwise set the sampling time to base times the time base of 10ms. base must be lower than 100.

tt setc <*channel*> <*probe*> <*quad*> With no arguments return the current map relating APD channel channel to tilt probe probe and probe quadrant quad. Otherwise maps APD channel channel to quadrant quad of tilt probe probe.

tt pweight <*wp1*> <*wp2*> With no arguments return the current weight assigned to each probe in the centroid error calculation. Otherwise set the weights to the given values wp1 and wp2.

tt bias <*0 | 1*> - With no arguments return the current bias flag. Otherwise turn bias subtraction on and off.

tt setbias <*channel*> <*bias*> - With no arguments return the current bias for the given channel. Otherwise set the bias to the given value. Parameter *channel* can take value 0 to 7. Parameter *value* is in units of [photon-counts].

tt minf <*mf1*> <*mf2*> - With no arguments return the current min flux to calculate tilt errors for each guide probe. Otherwise set the min flux to the given values. Parameters *mf1* and *mf2* are in units of [photon-counts].

tt lock <*0 | 1*> <*0 | 1*> - Control which guide probe is used for guiding.

2.4.2 AO Loop Data FIFO

There's also a FIFO (*/dev/rtf4*) to offload AO loop variables to the user space. This FIFO gets created by the *ao_init()* routine in file *aoLib.c*. Every time the routine *ao_task()* is executed, a data structure is filled with loop variables and put in the FIFO.

Each element is a data structure of type _AO_DATA (*aoLib.h*). The structure contains 6 elements: a time stamp, the x-y tilt error, the x-y slopes, and the DM & LLT M3 voltages. The size of the data structure is 1768 bytes (NS=77, AV_MAX=64).

```
typedef struct {
      double ns;              /* time stamp in nanoseconds */
      double tx;              /* tilt error in x */
      double ty;              /* tilt error in y */
      double sdx[NS];         /* sub-aperture gradients in x */
      double sdy[NS];         /* sub-aperture gradients in y */
      double vol[AV_MAX];     /* DM & LLTM3 voltages */
} _AO_DATA;
```

The size of the FIFO buffer was selected assuming the AO loop is pushing data into the FIFO at a maximum rate of 1KHz, the user space loop is popping data from the FIFO at a rate faster than 10Hz, and imposing the requirement that the time to fill the buffer was four times the pop period (2.3).

$$(Pop\,Period) < 0.25 \times (Time\ to\ Fill\ Buffer) \tag{2.1}$$

$$(\ Time\ to\ Fill\ Buffer\ )\ =\ (Push\ Period)\ \times\ \frac{(Buffer\ Size)}{(Data\ Structure\ Size)} \tag{2.2}$$

$$(\ Buffer\ Size\ )\ >\ \frac{4\ \times\ (\ Pop\ Period\ )\ \times\ (\ Data\ Structure\ Size\ )}{(Push\ Period)} \tag{2.3}$$

Since the size of the data structure is 1736 bytes, at 1KHz the AO loop pushes data into the FIFO every 1ms, and at 10Hz the user space pops data every 100ms, the size of the FIFO buffer must be at least 694,400 bytes.

2.4.3 TT Loop Data FIFO

FIFO */dev/rtf3* is used to offload TT loop variables to the user space. The FIFO gets created by the *tt_init()* routine in file *ttLib.c*. Every time the control routine *tt_100Hz_task()* runs, a data structure is filled with TT loop data variables and put in the FIFO.

The data structure is of type _TT_DATA (*ttLib.c*). The structure contains 8 elements: a time stamp, the weighted x-y tilt error, the photon counts, the M3 position command, and the raw tilt errors. The size of the data structure is 152 bytes (NP=8, NTT=2)

```
typedef struct {
      double ns;        /* time stamp in nanoseconds */
      double ex;        /* weighted tilt error in x */
      double ey;        /* weighted tilt error in y */
      double c[NP];     /* photon counts */
      double gamma;     /* m3 command for elevation */
      double beta;      /* m3 command for azimuth */
      double tx[NTT];   /* tilt errors in x */
      double ty[NTT];   /* tilt errors in y */
      double ox;        /* wobble tool offset in x */
      double oy;        /* wobble tool offset in y */
} _TT_DATA;
```

The size of the FIFO buffer was selected assuming the TT loop is pushing data into the FIFO at a maximum rate of 100Hz, a user space loop is popping data from the FIFO at a rate faster than 10Hz, and imposing the requirement that the time to fill the buffer was four times the pop period.

Since the size of the data structure is 152 bytes, at 100Hz the TT loop pushes data into the FIFO every 10ms, and at 10Hz the user space applications pops data every 100ms, the size of the FIFO buffer must be at least 6080 bytes (2.3).

2.4.4 CTRL_I Shared Memory Buffer

The 32 bit signed integer data buffer (CTRL_I) holds the t_ctrl_int data structure (aoLib.c). It contains the status of the AO control loop and the time spent in calculation on each loop cycle.

```
typedef struct {
      int track;              /* open/closed loop */
```

```
      int etime;
} t_ctrl_int;
```

2.4.5 CTRL_F Shared Memory Buffer

The double precision float data buffer (CTRL_F) holds the *t_ctrl_float* data structure (*aoLib.c*). It contains the arrays to store x-y slopes, flux and background for each sub-aperture, the x-y tilt error, and the DM voltages.

```
typedef struct {
      double sdx[NS];         /* x sub-aperture shift */
      double sdy[NS];         /* y sub-aperture shift */
      double flux[NS];        /* sub-aperture flux */
      double bgnd[NS];        /* sub-aperture bgnd */
      double tx;              /* x tilt measurement */
      double ty;              /* y tilt measurement */
      double nvol[AV_DM];     /* nominal DM voltages */
      double vol[AV_DM];      /* current DM voltages */
} t_ctrl_float;
```

2.4.6 RMAT_X and RMAT_Y

Floating point buffers of max size AV_DM by NS to hold the reconstructor matrices for X and Y slopes. Implemented in *aoLib.c*.

2.4.7 BANK_I Shared Memory Buffer

16 bit signed integer buffer to pass an artificial image of max size 80x80. Data is to be stored in this buffer in quad readout interlaced mode to optimize performance. Implemented in *aoLib.c*.

2.4.8 DARK_I Shared Memory Buffer

16 bit signed integer buffer to pass a bias calibration frame of max size 80x80. Data is to be stored in this buffer in quad readout interlaced mode to optimize performance. The name was kept for historical reasons only. Implemented in *centroidLib.c*.

2.4.9 MASK_I Shared Memory Buffer

16 bit signed integer buffer to characterize pixels in the frame. Each element in the buffer is a bitwise mask that defines the sub-aperture owner, its amplifier in the CCD, if it is a background pixel, and if it is a bad pixel. Data is to be stored in this buffer in quad readout interlaced mode to optimize performance. Implemented in *centroidLib.c*.

Bits 15-8 : Sub-aperture ID. Pixels not owned by any sub-aperture are marked 0xFF.

Bit 7-6 : Amplifier number

Bit 4 : Background pixels. 1 background pixel. 0 regular pixel.

Bit 0 : Bad pixels. 0 bad. 1 good.

2.4.10 GRID_X and GRID_Y Shared Memory Buffers

Double precision buffer to pass x and y centroid weights. Implemented in *aoLib.c*.

2.4.11 WEIGHT Shared Memory Buffer

Double precision buffer. Eventually this buffer can be used to pass flat field corrections also. Implemented in *aoLib.c*.

2.4.12 Device File /dev/soartt

The device file interface allows the user space application to inject user defined waveforms to the M3 mirror. The interface supports the *open*, *close*, *write* and *ioctl* file operations.

Typically the user space application will use the write operation to pass waveform data in units of ADU (+/- 32768). The implementation of the write operation *sao_write()*, will take the data and will write to one of two buffers until its full. At that point the *write* operation will block until all the samples in the other buffer have been generated and sent to the M3 mirror by the 100Hz task *tt_100Hz_task()*. When the 100Hz task has finished with the current buffer, it switches to the other and unblocks the *write* operation so it can start filling the one that is now free. In this way the 100Hz task keeps cycling between the two buffers while the user space application writes big chunks of waveform data.

The synchronization between the 100Hz task and the *write* operation is obtained by using a “system request”. The 100Hz task calls *rt_pend_linux_srq()* at each end of block to trigger the non real-time system request handler. This technique allows for a real-time task to indirectly use the non real-time kernel mechanisms for wait-queues. In this case when *sao_write()* finds that the buffer is full, the process is put to sleep (*interruptible_sleep_on()*) until it is awakened by the system request handler (*wake_up_interruptible()*).

2.5 DM Control Loop

The CCD controller interface board generates an interrupt every time an image transfer to memory is completed (a progressive read might be explored if necessary). That interrupt triggers the interrupt handler *astropci_intr()* in its software driver module *astropci.o*, which in turns triggers the execution of a bottom-half interrupt handler implemented by the *ao_task()* routine in *aoLib.c*.

A pseudo code version of the DM control algorithm follows. The argument to subroutine *ao_task()* is the current frame memory address. NS is the number of sub-apertures and AV the number of actuator voltages.

```
void ao_task (unsigned short *frame)
{
  while (1) {
    rt_task_suspend();                  /* sleep waiting for interrupt */
    for (i=0; i<NS; i++) {              /* x-y slopes */
      cc_slopes(frame,&(subap[i]),&(sdx[i]),&(sdy[i]));
      sdx[i] -= px[i];                  /* references */
```

```
        sdy[i] -= py[i];
      }
      for (i=0; i<AV; i++) {
        for (j=0, evol=0.0; j<NS; j++) {   /* slopes to voltages */
          eK += recmatx[i][j] * sdx[j] + recmaty[i][j] * sdy[j];
        }
                                           /* digital controller */
        volK[i] = K1 * volK-1[i] + K2 * volK-2[i] + K3 * eK;
                                           /* voltage generation */
        dm_save_to_buffer(board, j, value[j]);
      }
      dm_flush_buffer(board, j, value[j]);
    }
  }
```

The algorithm calculates the x-y slopes for each sub-aperture, subtracts the reference, and then multiplies the resulting vector by the reconstruction matrix to obtain a vector of voltages. This vector is then filtered (digital controller) and the resulting command vector is applied to the mirror. The digital controller parameters K_1, K_2, and K_3 are selected depending on the selected type of filter (SP or Simple Integrator) according to the following recipe (see section 2.5.2 below)

```
Smith Predictor:  K1 =  KL – KS;          Integrator: K1 =  KL;
                  K2 =  KL x KS;                      K2 =  0;
                  K3 = -K;                            K3 = -K;
```

To accommodate for the LGS and TSLGS modes of operation, the real code for the DM control algorithm is capable of preventing the DM from correcting tilt. The tilt component is estimated by averaging the slopes and then subtracted and the reconstructor is replaced with a version with its average slopes subtracted (see sections 3.3.7.2 and 3.3.7.3 below).

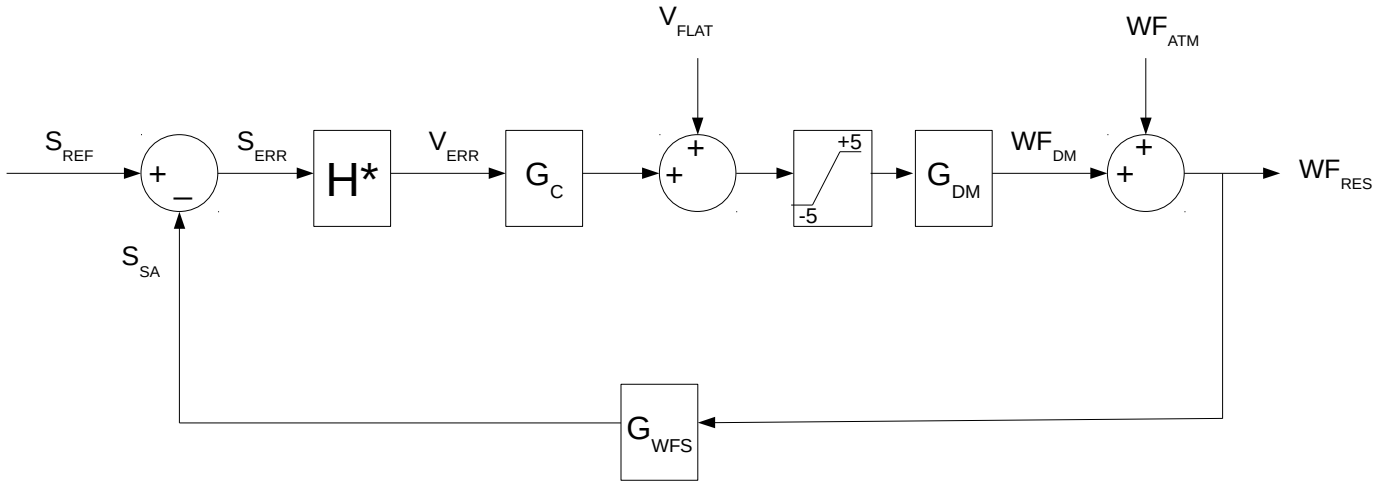

Figure 2.1: The WFS measures the residual wavefront slopes. The residual wavefront is projected to the DM actuators voltage space to be filtered by the controller. The resulting voltages are added to the nominal operation point voltages and then applied to the DM.

2.5.1 Centroid Algorithms

The RTCORE supports two centroid algorithms: *Cross Correlation (CC)* and *Weighted Center of Gravity (WcoG)*. The algorithms are implemented in the file *centroidLib.c*. The entry point for both algorithms is the routine *cc_error ()*. Which algorithm is to be used depends on the current value of the local variable *cc_algorithm*.

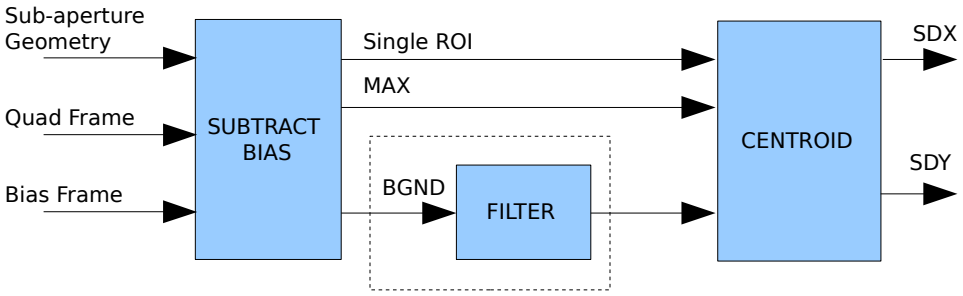

Figure 2.2: The Bias Frame is subtracted to estimate the background. The background is also subtracted to estimate the x-y spot position.

Cross Correlation. The method is implemented by the routine *cc_do_cross_correlationc4()*. Each sub-aperture box is convoluted in the Fourier domain, with a predefined Gaussian spot template. The size of the spot template is defined by the local variable *cc_Nw* (**cc Nw** command). The result is analyzed to find the max value C_0 and then do a parabola fit through three points to estimate the spot position (2.4). The position returned by the algorithm, is referred to the center of the box. For example, for a 8x8 sub-aperture box, the pixel in first row/first column has coordinates (-3.5,-3.5).

$$\hat{x} = x_{Max} + 0.5 \times \frac{C_{-1} - C_1}{C_{-1} + C_1 - 2C_0} \tag{2.4}$$

Weighted Center of Gravity. The method is implemented by the routine *cc_cmass_quad()*. Each sub-aperture box is multiplied by the calculation grid mask (GRID_X and GRID_Y memory buffers, section 2.4.10) and the weights mask (memory buffer WEIGHT, section 2.4.11) as defined in equation (2.5). The coefficient **FL**is needed to ensure unit response [TBD]. N_T is the FHWM of the spot in pixels (local variable *cc_NT*), and N_w is the FWHM of the weighting function (local variable *cc_Nw*). Both parameters can be defined using the **cc NT** and **cc Nw** RTCORE commands.

$$\hat{x} = \gamma \frac{\sum x I_{x,y} (F_w)_{x,y}}{\sum I_{x,y} (F_w)_{x,y}} \tag{2.5}$$

$$\gamma = \frac{N_T^2 + N_w^2}{N_w^2} \tag{2.6}$$

Both implementations include support for discarding bad pixels, bias subtraction, and background subtraction. After bias and background subtraction, negative values are discarded and only positive values are used to estimate the spot position.

2.5.1.1 Bad Pixels

Bit 0 of the bitwise frame mask in buffer MASK_I (see section 2.4.9 above) is used to check if pixels are good or bad. Bad pixels are ignored by the centroid algorithms.

2.5.1.2 Bias Subtraction

A bias frame is subtracted from each acquired image before calculating the centroid on each sub-aperture. The bias frame is stored in buffer DARK_I (section 2.4.8). Is to the user to keep the bias frame up to date, depending on the current readout parameters.

2.5.1.3 Background Subtraction

A leaky average is calculated on each bias-subtracted sub-aperture, at each loop cycle. The sub-aperture pixels to use in the average are defined by bit 4 of the bitwise mask MASK_I (section 2.4.9). The background estimate for each sub-aperture is then subtracted before estimating the centroid.

$$Bgnd_L(k) = \frac{Bgnd(k) + f_L \times Bgnd_L(k-1)}{1 + f_L} \tag{2.7}$$

2.5.1.4 Sub-aperture Flux

For each sub-aperture the RTCORE estimates its flux as the result of adding all the good pixels, as defined by the bitwise frame mask. The sub-aperture pixels are first bias and background subtracted. The estimate is store in the flux element of the *roi* data structure linked to each sub-aperture (*centroidLib.h*), and is accessible from user space through the **sd** command (see section 2.4.1).

2.5.2 Digital Controller

The difference equations for the digital controllers supported by the RTCORE are as follow [TBD]. The general equations are

$$v(k) = (K_L - K_S)v(k-1) + K_S K_L v(k-2) + K\Delta v(k) \qquad \textit{Smith Predictor} \tag{2.8}$$

$$v(k) = K_L v(k-1) + K \Delta v(k) \qquad \textit{Integrator} \qquad (2.9)$$

v(k) is the voltage at time *kT* (k=0,1,... and *T*=loop time), and *ův(k)* is the voltage error at time *kT* obtained from multiplying the slopes by the reconstructor matrix. The two are implemented by the function *ao_task()* in file *aoLib.c* in a generic way as

$$v(k) = K_1 v(k-1) + K_2 v(k-2) + K_3 \Delta v(k) \qquad (2.10)$$

where K_2 is zero when the pure integrator is used. The values for the different parameters can be set using the **aofg sp** and **aofg di** commands.

2.5.3 Voltage Generation

Voltage generation is accomplished by first filling a generation buffer with the output voltages of the digital controller, and then flushing that buffer to update the D/A outputs of the DAQ boards (Waveform Regenerate Mode) . This is the fastest method of generation since single output generation adds too much overhead. The single update mode was discarded; best performance is 66us per channel, which means 4ms for 60 channels in sequential mode.

Under Waveform Regenerate Mode, the board continuously recycles through a data set resident in its DSP buffer without fetching any new data. The transfer mode selected was standard mode. In this mode data are stored in the DSP buffer in a format where the channel number and associate actions are combined with output code. This mode is capable of achieving up to 455K samples/second.

The Analog output board needs a clock to instruct it how quickly to process entries in the channel list. The clock can be external or internal. It was chosen to use the internal clock which has a time base of 11Mhz. That give us a limit of 90.9 [ns] for each channel, so if we consider all 60 channels that leave us with a minimum of 5.454 [us]. To give time to the DSP to empty the FIFO before the next set of voltages is ready, it was decided to use a clock divider of 10.

DIO routines are implemented in the *dmLib.c* file. The routines define a tailored API to the Power DAQ drivers [6].

2.6 SOAR M3 Control Loop

The M3 control loop is implemented by the tt_100Hz_task() and tt_1KHz_task routines (*ttLib.c*). The 100Hz periodic task counts the photons received by the APD modules (8 channels), and then obtains the weighted averaged x-y tilt error vector. The 1KHz periodic task samples the x-y tilt error vector and passes it to the integrator to obtain the next command to the M3. A pseudo code version of the TT control algorithm follows.

```
void tt_100Hz_task ()
{
```

```
  apd_start_counters();             /* start counting photons */
  while (1) {
    rt_task_wait_period();
    apd_centroid (&x, &y);    /* tilt error signal */
  }
}

void tt_1KHz_task ()
{
  while (1) {
    rt_task_wait_period();
    x_k = x_k-1 + K * ux_k-1;
    y_k = y_k-1 + K * uy_k-1;
    [gamma_k beta_k]= 32768 + [Hstar][x_k y_k];
    m3_send_command (gamma_k, beta_k);
  }
}
```

The x-y tilt error plus other parameters are off loaded through the RTFIFO */dev/rtf3* to user space for later processing.

2.6.1 Centroid Algorithm

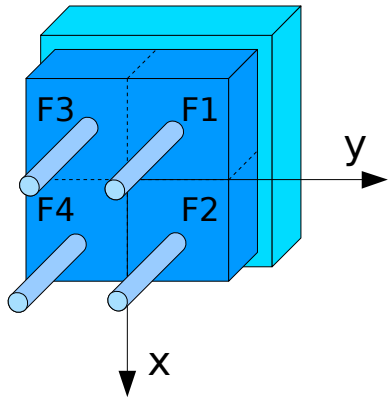

Figure 2.3: TT probe x-y coordinate system. The convention follows the one proposed in document [7].

The photon counting support is implemented in the library *apdLib.c*. This library is a wrapper around ni660x driver module. The ni660x module is the Linux driver for the counter timer PXI board NI6602. The library API defines three functions: *apd_init(), apd_centroid(), apd_cleanup()*. *apd_init()* configures the eight counters in the NI6602 board, and *apd_centroid()* returns the x-y centroid. *apd_cleanup()* stop the counters.

$$ex = \frac{(F2+F4)-(F1+F3)}{F1+F2+F3+F4} \qquad ey = \frac{(F1+F2)-(F3+F4)}{F1+F2+F3+F4} \tag{2.11}$$

The algorithm is a classic quad centroid (2.11). The convention for the x-y coordinate system and

fiber labels, follows the one proposed in document [7]. When both TT probes are used, the error vector for each probe is weighted and then added. Weights are normalized to sum 1. The criterion to defined the weight values is TBD.

$$\vec{e} = w_1 \vec{e}_{TTP1} + w_2 \vec{e}_{TTP2} \tag{2.12}$$

2.6.1.1 Bias Subtraction

A bias is subtracted from each fiber reading before calculating the centroid on each probe. The default value for the bias is zero. Is to the user to keep the bias numbers up to date. One would expect the user space application to take care of that matter.

2.6.2 Digital Controller

The digital controller is implemented by the *tt_1KHz_task()* routine. The task continuously integrates the x-y tilt error, and then sends the result to the M3 servo mechanism in El-Az coordinates.

The z-transform of the chosen compensator[3] is

$$G_{C1}(z) = \frac{K_{C1}}{(z - 1)} \tag{2.13}$$

The difference equation describing the filter used is

$$\gamma,\beta(k) = \gamma,\beta(k-1) + K_{C1} \times \Delta\gamma,\beta(k-1) \tag{2.14}$$

The implementation allows to enable and disable the integrator by making the second term in (2.14) zero, causing the integrator to hold the last output value produced. The motivation for this feature is the operational need of temporarily holding the last mirror position while having the guider x-y stages do an automatic compensation of a telescope offset (*follow mode*) without loosing the object.

The gain K can be adjusted using the **tt m3 KC1** command. For more details read SDN-8307.

2.6.3 X-Y to M3 Coordinates

The relation between the integrator output command in x-y coordinates, and the actual M3 command in El-Az coordinates, is given by the reconstructor matrix $\mathbf{H^*_{M3}}$ which is dependent on the M3 elevation gain G_{EL} in ADU/arc-second, the M3 azimuth gain G_{AZ} in ADU/arc-second, the mechanical rotator angle A_r, the telescope mount elevation angle A_e, and a zero rotation angle A_z.

$$\vec{c}_{ElAz} = H^*_{M3} \vec{c}_{xy} \tag{2.15}$$

3 Compensator selection was based on M. Warner simulations of the TT control loop.

Internally, for doing its calculations the RTCORE use Guide Probes x-y coordinates in arc-seconds to reference the M3 command, independent of the tilt error source selected (WFS or Guide Probes). When the WFS is selected the tilt error is first transformed to GP x-y coordinates using (2.16).

$$\begin{bmatrix} X_{GP} \\ Y_{GP} \end{bmatrix} = \begin{bmatrix} 0.016743 & -0.314435 \\ -0.314435 & -0.016743 \end{bmatrix} \begin{bmatrix} X_{SD} \\ Y_{SD} \end{bmatrix} = WFS_{scale} \times \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(93^o) & -\sin(93^o) \\ \sin(93^o) & \cos(93^o) \end{bmatrix} \begin{bmatrix} \cdot \end{bmatrix} \tag{2.16}$$

The reconstructor matrix in (2.15) can be decomposed in a zero rotation reconstructor term $\mathbf{H^*_{M3,z}}$ and a rotation matrix term **R**. For SAM that relation is given by (2.17).

$$\begin{bmatrix} EL \\ AZ \end{bmatrix} = \begin{bmatrix} G_{EL} & 0 \\ 0 & G_{AZ} \end{bmatrix} \begin{bmatrix} \cos(T_{EL} - T_{ROT}) & -\sin(T_{EL} - T_{ROT}) \\ \sin(T_{EL} - T_{ROT}) & \cos(T_{EL} - T_{ROT}) \end{bmatrix} \begin{bmatrix} X_{GP} \\ Y_{GP} \end{bmatrix} \tag{2.17}$$

$\mathbf{H^*_{M3,z}}$ is precomputed and passed to the RTCORE at boot time (2.18). **R** is computed every 2[s] based on the current rotator mechanical angle and mount elevation, and multiplied by $\mathbf{H^*_{M3,z}}$ to obtain $\mathbf{H^*_{M3}}$.

$$H^*_{M3,Z} = \begin{bmatrix} 1170.3 & 0 \\ 0 & 1638.4 \end{bmatrix} = \begin{bmatrix} \dfrac{65535}{4 \times 14''} & 0 \\ 0 & \dfrac{65535}{4 \times 10''} \end{bmatrix} \tag{2.18}$$

2.6.4 M3 Command Generation (Only for PDAQ DIO Board)

The RTCORE updates the set-point of the M3 servo mechanism using the command interface described in the Tertiary Mirror Assembly interface document. To generate the command signals, the computer has installed a PowerDAQ multichannel DIO board. The driver module for the board is provided by the vendor[4]. A small wrapper tailored for the RTCORE application was written (*m3Lib.c* and *m3Lib.h*), capable of configuring the DIO board in the appropriate operation mode, and of encapsulating the packaging of the command words. The interface to the wrapper is given by the following routines:

m3_init() – initializes data structures and hardware

m3_cleanup() – stops signal generation a free data structures

m3_send_command() – packs and sends the two words representing tip tilt position angles.

The DIO board has 4 ports of 16 channels each. The m3Lib.c implementation uses channels 0, 1, and 2 from port 0 to output the command signals. The strobe signal is generated through channel 0, the data bits through channel 1 and the clock signal through channel 2 (Table 2.2). To accommodate all three signals the number of points per channel was chosen to be 35: 1 start bit (low), the 32 data bits for the two 16 bits commands, 1 bit high for the strobe signal, and then 1 end bit (low). The 35 points are accommodated in a buffer of size 70 bytes (Figure 2.3); for each point to transmit, two memory bits

4 For details on installation of the PowerDAQ driver modules read document [6].

have to be allocated to accommodate a complete clock period (high and low).

Tabla 2.2: Channel assignments for the DIO board. All channels belong to port 0.

Channel	Description
0	Strobe Signal
1	Data Bits
2	Clock

The points for all three channels are stored in the buffer using the PowerDAQ 32bit data format; bits 31-19 are unused, bits 18-16 represent the target port number (from 0 to 3), and bits 15-0 hold the binary data to output (LSB is channel 0).

The 1KHz task fills the buffer and send the command to the M3 by calling the routine *m3_send_command()*. The routine packs the two 16 bit command words into the output buffer using the format above and then triggers digital output by the DIO board.

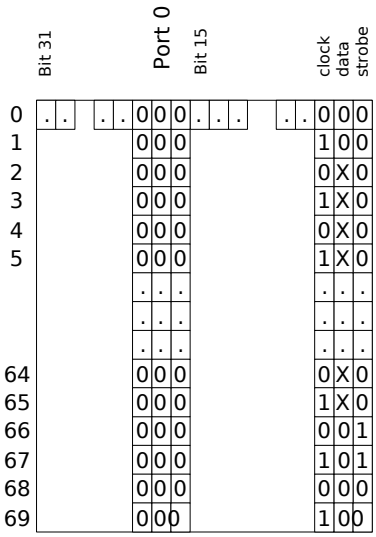

Figure 2.4: The number of points per channel is 35. Each point is represented by two consecutive 32 bit integers in the output buffer (one clock period).

2.6.5 User Defined Waveforms

User defined waveforms can be injected using the device character interface */dev/soartt*. The way it work was already described in section 2.4.12.

2.7 SOAR Mount Control Loop

The mount control loop is implemented by the *tt_1KHz_task()* routine. The task passes the command signal to the M3 by a low pass filter and then multiplies its output by a simple gain. The result is made available through the **tt tp 3** command to the user space application to be sent to the TCS

at 1[Hz] (see section 3.3.17). For more details read SDN-8307.

2.7.1 Digital Controller and Low Pass Filter

The z-transform of the low pass filter and gain is

$$G_{LP2}(z) = K_{C2} \times \frac{K_{LP2}}{(z - a_{LP2})} \tag{2.19}$$

The difference equation describing the filter is

$$x,y(k) = a_{LP2} \times x,y(k-1) + K_{C2} K_{LP2} \times \Delta x,y(k-1) \tag{2.20}$$

When the M3 control loop is open, ůx,y(k-1) corresponds to the x-y tilt error measured directly by the WFS or Probes. The value of the gain can be adjusted using the **tt KC2** and **tt LP2** commands.

2.7.2 X-Y SAM coordinates to X-Y SOAR TCS coordinates

For doing its calculations the RTCORE use Guide Probes x-y coordinates to reference the Mount commands, independent of the error source selected (2.16).

For sending the Mount command the RTSOFT use SOAR TCS coordinates. For SAM that relation is given by (2.21) where *PA*[5] is the instrument position angle reported by the TCS.

$$\begin{bmatrix} \Delta X_{TCS} \\ \Delta Y_{TCS} \end{bmatrix} = \begin{bmatrix} -\cos(-PA) & -\sin(-PA) \\ -\sin(-PA) & \cos(-PA) \end{bmatrix} \begin{bmatrix} \Delta X_{GP} \\ \Delta Y_{GP} \end{bmatrix} \tag{2.21}$$

2.8 LLT M3 Control Loop

The LLT M3 control loop is implemented by the *tt_1KHz_task()* routine. The task integrates the residual tilt measured by the WFS (see section 2.5.1 above) under LGS mode and applies the resulting command to the LLT M3 mechanism. For more details read SDN-8307.

2.8.1 Digital Controller

The z-transform of the integrator is

$$G_{C3}(z) = \frac{K_{C3}}{(z - 1)} \tag{2.22}$$

The difference equation describing the equation above is

$$x,y(k) = x,y(k-1) + K_{C3} \times \Delta x,y(k-1) \tag{2.23}$$

5 Angle between North and and Y_{GP} axis

The implementation allows to enable and disable the integrator by making the second term in (2.23) zero, causing the integrator to hold the last output value produced. The gain *K* can be adjusted using the **tt llt KC3** command.

2.8.2 X-Y SAM Coordinates to LLT M3 Piezoelectric-Voltages

For doing its calculations the RTCORE use Guide Probes x-y coordinates to reference the LLT M3 commands. For sending the LLT M3 command the RTSOFT use Piezoelectric Voltage coordinates. For SAM that relation is given by (2.24).

$$\begin{bmatrix} AZ \\ EL \end{bmatrix} = \begin{bmatrix} hz11 & hz12 \\ hz21 & hz22 \end{bmatrix} \begin{bmatrix} \cos(T_{EL} - T_{ROT}) & -\sin(T_{EL} - T_{ROT}) \\ \sin(T_{EL} - T_{ROT}) & \cos(T_{EL} - T_{ROT}) \end{bmatrix} \begin{bmatrix} X_{GP} \\ Y_{GP} \end{bmatrix} \tag{2.24}$$

The reconstructor matrix in (2.24) can be decomposed in a zero rotation reconstructor term $\boldsymbol{H^*_{LM3,z}}$ and a rotation matrix term $\boldsymbol{R}$. $\boldsymbol{H^*_{LM3,z}}$ is precomputed and passed to the RTCORE at boot time. **R** is computed every 2[s] based on the current rotator mechanical angle and mount elevation, and multiplied by $\boldsymbol{H^*_{LM3,z}}$ to obtain $\boldsymbol{H^*_{LM3}}$.

2.8.3 LLT M3 Command Generation

Command generation is done using two spare channels (30 and 31 in board #2) available in the AO boards used to generate voltages to control the DM. Read section 2.5.3 above for details.

2.9 LLT M1 Control Loop

The LLT M1 control loop is implemented by the *tt_1KHz_task()* routine. The task filters and integrates the tilt command to the LLT M3. The result is made available through the **tt tp 4** command to the user space application to be sent to the LLT M1 at 1[Hz]. Read SDN-8307 for details.

2.9.1 Digital Controller and Low Pass Filter

The z-transform of the integrator and low pass filter is

$$G_{LP4}G_{C4}(z) = \frac{K_{LP4}}{(z - a_{LP4})} \times \frac{K_{C4}}{(z - 1)} \tag{2.25}$$

The difference equation describing the equation above is

$$x,y(k) = a_{LP4} \times x,y(k-1) + K_{LP4}K_{C4} \times \Delta x,y(k-1) \tag{2.26}$$

The implementation allows to enable and disable the integrator by making the second term in (2.26) zero, causing the integrator to hold the last output value produced. The gain and filter parameters can be adjusted using the **tt llt KC4** and **tt llt LP4** commands.

When the LLT control loop is open, the output x-y command is set to zero.

Chapter 3: The RTSOFT LabVIEW Application

3.1 Source Code

The source code for the RTSFOT LabVIEW Applications lives in directory *Rtsoft/LV2010Modules*. To access the code is recommended to start by opening the main VI first:

```
% cd RTsoft
% labview Rtsoft.vi
```

Some of the modules include C code that has to be compiled and installed as shared libraries. Read below for more details on how to build the shared libraries.

3.1.1 FITS

```
# cd FITS/private/
# make
# make install
```

3.1.2 LV2010-RTAI3.8

```
# cd LV2010-RTAI3.8/private/3.8
# make
# make install
```

3.1.3 MEMLIB

```
# cd MEMLIB/private
# make
# make install
```

3.1.4 NI660X

```
# cd NI660X/private
# make
# make install
```

3.1.5 SDSU-III

```
# cd SDSU-III/private/astropciAPI_LIB/LINUX
# make
# make install
# cd ../../astroIIILib/
# make
# make install
```

3.1.6 STFLIB

```
# cd STFLIB/private/c
# make
# make install
```

3.1.7 TTCOMMSLIB

```
# cd TTCOMMSLIB/private
# make
# make install
```

3.1.8 UNSCRLIB

```
# cd UNSCRLIB/private
# make
# make install
```

3.2 Architecture

The LabVIEW part of the RTSOFT is made up of several individual tasks that together built up the complete application functionality. These tasks communicate among them by means of global variables and queues. Each task is basically a VI in which its block diagram is a while loop that runs until the application exits.

Table 3.1 lists the LabVIEW tasks that built up the RTSOFT application. The main VI *Rtsoft.vi* initializes the global variables and then launches the different tasks that will serve background jobs as well as local and remote commands. Some of the tasks will run only on user demand; these tasks typically represents the GUI front ends.

3.2.1 Consumer Producer Model for Data Acquisition

To serve the purpose of acquire data (e.g. slopes, voltages, CCD frames), analyze the data, present the data and even close some very slow control loops based on that data, the RTSOFT has been built using a consumer-producer model. In this model one task, called the producer (P), generates data by some mean and makes it available to other tasks for them to use it, the consumers (C). RTSOFT has basically three producers: the AO Loop Data Task, the TT Loop Data Task and the Frame Data Task.

3.2.2 Execution Threads

LabVIEW provides access to multi-threading mechanisms for the applications developed under its environment. RTSOFT benefits of multi-threading by executing time critical tasks in their own thread and by assigning priority to tasks within the same thread.

There is a limited number of execution threads available though: user interface, standard, instrument I/O, data acquisition, other 1, other 2, and same as caller. The same with the available priorities: background, normal, above normal, high, time critical, and subroutine. Table 3.1 summarizes the execution thread and priority assignments for the RTSOFT tasks.

The effective use of multi-threading, calls for marking all VI's that are shared among threads as re-entrant, thus high priority tasks can preempt lower priority tasks while running. RTSOFT uses re-entrant VIs whenever possible, except for global variables. The LabVIEW native implementation of global variables has proved to be unreliable, so the available workaround had to be used: type II global variables. The penalty in the use of the type II global variables is that by definition they are non re-

entrant, which opens space to situations in which a task will have to wait for other task to release the variable despite its higher priority.

Table 3.1: RTSOFT tasks. The application is built grouping several tasks that together bring all the needed functionality.

VI Name	Type[6]	Queue	Running	Thread	Priority	Description
RTsoft			Always	User interface	Normal	Main VI
rtsoft_task_loop_data	P		Always	Data acquisition	Time critical	Reads AO Data from rtf4 @10Hz.
rtsoft_task_frame_data	P		Always	Other 1	Normal	Reads last frame @10Hz.
rtsoft_task_tt_loop_data	P		Always	Data acquisition	Time critical	Reads TT data from rtf3 @20Hz.
rtsoft_task_mount			Always	Data acquisition	Normal	Send mount correction @1Hz
rtsoft_task_modal_optimization	C	1	Always	Data acquisition	Normal	Loop time, residual error, atmospheric parameters.
rtsoft_task_tt_performance	C	11	Always	Data acquisition	Normal	Loop time, flux.
rtsoft_task_rotator_angle			Always	Other 1	Normal	Sets rotation angle in the RTCORE @0.5Hz.
rtsoft_task_LLT			Always	Data acquisition	Normal	Sends LLT correction @1Hz
parse_task			Always	Standard	Normal	Parse and execute string commands.
cx_task_server			Always	Instrument I/O	Normal	Receive command from the ICS.
cx_task_client			Always	Instrument I/O	Normal	Send commands to the TCS as a remote client.
hist_task			Always	Standard	Background	Log alarms and events.
rtsoft_simul_task			Always	Standard	Normal	Simulator engine.
rtsoft_gui_dm	C	6	On Demand	User interface	Background	DM user interface
rtsoft_gui_loop			On Demand	User interface	Normal	AO loop control user interface.
rtsoft_gui_wfs	C	5	On Demand	User interface	High	WFS user interface.
rtsoft_gui_display	C	7	On Demand	User interface	Background	WFS frame display.
rtsoft_gui_slope_statistics	C	4	On Demand	User interface	Background	Min, max, mean and variance of x-y slopes.
rtsoft_gui_histogram	C		On Demand	User interface	Background	Last WFS image histogram.
rtsoft_gui_loop_data_recorder	C	3	On Demand	Data acquisition	Normal	User interface to save an AO data sequence.
rtsoft_gui_recerror			On Demand	User interface	Normal	Reconstruction error plot and value.
rtsoft_gui_sub-apertures	C		On Demand	User interface	Background	Sub-aperture flux and background. Spot size.
rtsfot_gui_noise_propagation			On Demand	User interface	Normal	Noise Propagation for current WFS geometry.
rtsfot_gui_modal_coefficients	C	8	On Demand	User interface	Normal	Slopes, open and close loop modal coeficients.
rtsoft_gui_bias_calibration			On Demand	User interface	Normal	Take/save/apply bias calibration frames.
rtsoft_gui_mask			On Demand	User interface	Normal	Create/save/apply pixel masks.
rtsoft_gui_weights			On Demand	User interface	Normal	Create/save/apply weight templates.
rtsfot_gui_cc_centroid			On Demand	User interface	Normal	Create/save/ apply cross correlation templates.
rtsfot_gui_tt_loop			On Demand	User interface	Normal	TT control loop user interface.
rtsoft_gui_tt_centroid_statistics	C	14	On Demand	User interface	Background	Probes x-y error signal charts.
rtsoft_gui_tt_loop_data_recorder	C	13	On Demand	Data acquisition	Normal	User interface to save a TT data sequence.
rtsoft_gui_tt_flux	C	12	On Demand	User interface	Normal	Probes flux charts. P parameter for each APD.

6 P for producers and C for consumer tasks.

3.3 Implementation Reference

3.3.1 AO Loop Data Task

The AO Loop Data Task is implemented by the VI *rtsoft_loop_data_task*. The VI continuously reads AO loop data coming from the RTCORE through RTFIFO 4 (*/dev/rtf4*), and injects the data in independent queues for each consumer task (Table 3.1). If a consumer task is not running, then the loop data task skips the queue and proceeds to the next consumer task in the list. The process repeats until the application shuts down. As shown in Table 3.1, the AO Loop Data task is marked as a time-critical priority task given the high need for its availability.

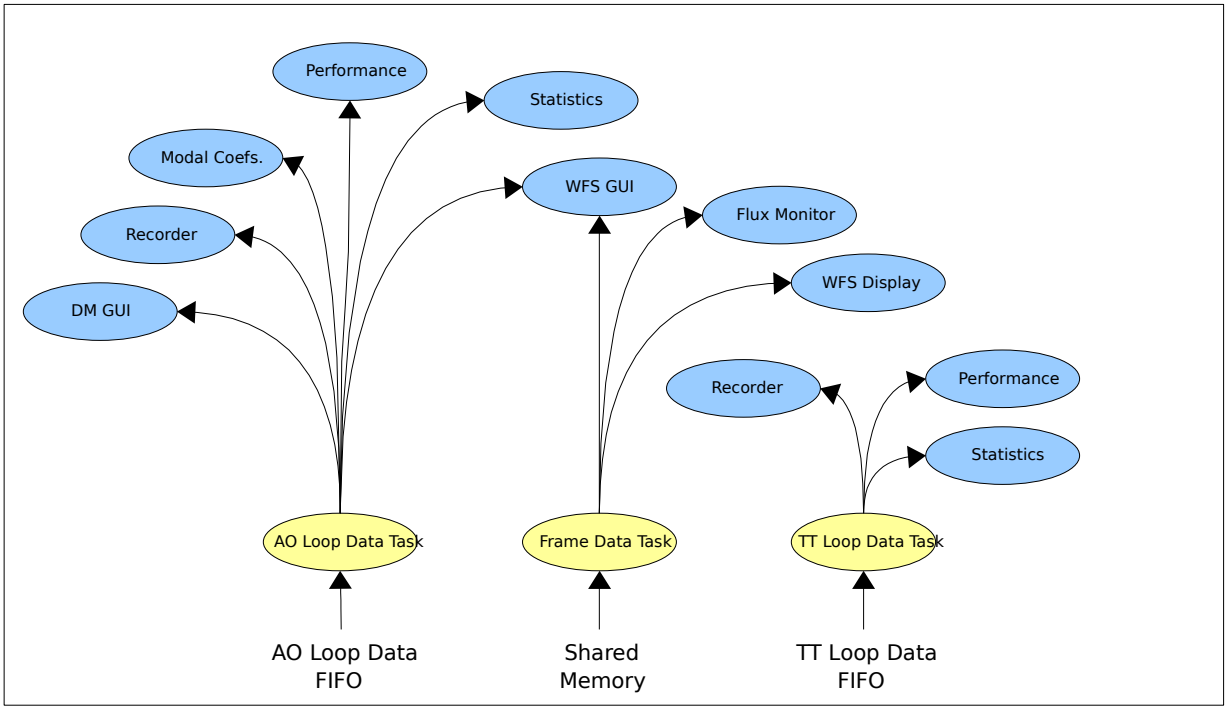

Figure 3.1: Data acquisition in RTSOFT is built using a consumer producer model.

3.3.2 TT Loop Data Task

The TT Loop Data Tasks is implemented by the VI *rtsoft_tt_loop_data_task*, and follows the same scheme that the AO Loop Data Task. The data is read from the RTCORE through RTFIFO 3 (*/dev/rtf3*).

3.3.3 Frame Data Task

The Frame Data Task is implemented by the VI *rtsoft_frame_data_task*. The VI continuously reads a CCD frame from its shared memory buffer, and puts the frame in the global variable *rtsoft_global_typeII_frame* along with other relevant statistics like min, max, mean, standard deviation, total flux, and background. This is not a time critical task, since the data produced by this

task serves mainly for display purposes.

Before updating the global variable, the task does a bias and background subtraction. Pixel to pixel bias subtraction is done by the *rtsoft_disp_subtract_bias* VI, using the bias calibration frame currently loaded into the application. Only then the background is subtracted from each CCD quadrant by the *rtsoft_disp_subtract_bgnd* VI. The background for each quadrant is obtained averaging the background over the sub-apertures that belong to the quadrant, as measured by the RTCORE. The information on what quadrant owns what pixel is in the bitwise frame mask in global variable *rtsoft_globals_typeII_mask*.

3.3.3.1 Reading a Frame

In LabVIEW CCD Frames are read using the API provided by the CAMERALIB library. The PCI interface board moves the image to the host memory in chunks of 1024 bytes. Thus the size of the buffer containing the CCD frame in memory is a multiple of 1024. The *rtsoft_frame_data_task* calls the *cam_get_current_quad_frame* to obtain the last frame in memory.

3.3.3.2 Quad-Unscrambling

The frames transfers to the host computer memory produce interlaced or scrambled images (Figure 3.2). Data must be unscrambled before doing any calculations to obtain coherent results.

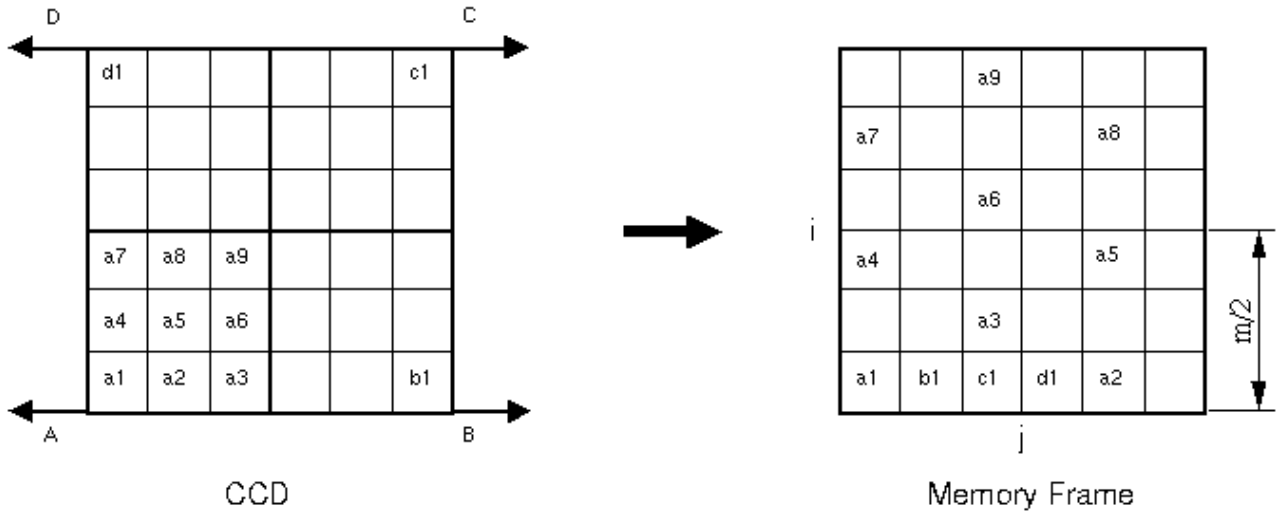

Figure 3.2: Diagram showing the scrambling of the data produced by the quad CCD readout process.

The LabVIEW program unscrambles the frames before using or presenting its data by applying the pseudo algorithm below.

```
for (i=0; i < m; i++) {
  for (j=0; j < m; j++) {
    if (i < m/2) {
      if (j < m/2)                  /* A amplifier */
        dest[m * i + j] = src[(m/2 * i + j) * 4];
      else                          /* B amplifier */
        dest[m * i + j] = src[(m/2 * i + (m – 1 – j)) * 4 + 1];
    }
    else {
      if (j < m/2)                  /* D amplifier */
        dest[m * i + j] = src[(m/2 * (m – 1 – i) + j) * 4 + 3];
      else                          /* C amplifier */
        dest[m * i + j] = src[(m/2 * (m – 1 – i) + (m - 1 – j)) * 4 + 2];
    }
  }
}
```

3.3.4 WFS User Interface

The WFS User Interface is implemented by the rtsoft_gui_wfs.vi. It is a consumer task that consumes data from both the AO Loop Data Task and the Frame Data Task.

3.3.4.1 Reference Positions

The reference positions represent the location of the reference spot for each sub-aperture relative to the center of the calculation box. The RTCORE will measure the spot displacements relative to those reference positions. The relative position is measured in pixels.

The reference spot positions are set in the RTCORE from the LabVIEW application through the **rshift** command (see section 2.4.1)

The procedure to measure these relative positions is implemented in LabVIEW by the VI *rtsoft_wfs_reset_reference*. The algorithm consist of, with the reference spots in the WFS, first average a user defined number of slopes for each sub-aperture, and then add the resulting vector to the current reference positions. Static aberrations, if any, are also added. The resulting vector is the new reference position without static aberration compensation. The values are stored for saving and applied back to the RTCORE plus aberrations.

3.3.4.2 Adding Static Aberrations to Reference Positions

It is a requirement to have the capacity of adding static aberrations to the corrected signal while in closed loop operation. The RTSOFT produce that effect by converting a user defined set of aberrations into offsets to the reference positions. The offset is obtained by multiplying the aberrations by the average gradient matrix of the WFS, which returns a slope vector in units of pixels.

The static aberrations are kept in a text file and are added to the reference spot positions at the moment of passing them to the RTCORE. That can happen when the user measure a new set of reference spot positions (*rtsoft_wfs_reset_reference*), when the application loads a set of reference spot

positions, and finally when the user interactively change and applies static aberrations (*rtsoft_gui_static_aberrations*)

3.3.4.3 Display of Zernike Mode Estimates

Estimation of the Zernike Modes is straight forward by multiplying the inverse gradient matrix G* by the slopes. This is done directly in the block diagram of the *rtsoft_gui_wfs* VI with the slope data produced by the AO Loop Task. The Z coefficients are expressed in units of [um] (standard Zernikes) in the WFS aperture.

3.3.4.4 Bias Calibration

Bias calibration frames acquisition is implemented by the *parse_command_bias* VI. The bias frame is obtained by averaging a user defined number of bias frames and then passing the averaged bias to the RTCORE in interlaced mode.

3.3.4.5 Pockels Cell

Operation of the Pockels Cell consists basically in activating and deactivating the opening and closing pulses to the Pockels Cell high-voltage driver. Two operation modes are available: LGS and TSLGS mode.

When LGS mode is selected the NI660X board in the PXI chassis is programmed to produce two channel independent pulses in synchronism with the laser TRIGGER OUT signal. The delay between the TRIGGER OUT signal and the first pulse is programmable as well as the separation between the first pulse and the second pulse.

When TSLGS mode is selected the NI660X board is also programmed to produce two channel independent pulses, but this time Continuous Pulse-Train Generation is used to produce the first pulse. The period of the pulse-train is programmable as well as the delay between the first pulse and the second pulse.

Selection and activation of either mode is also available remotely via string command interface.

3.3.5 WFS Display

Display of the WFS current image is implemented by the *rtsoft_task_display* VI. The task consumes data produced by the Frame Data Task (*rtsoft_task_frame*). In particular it reads the global variable holding a reduced image data *rtsoft_globals_typeII_frame*, every 80ms. The refresh time decrease proportional to the zoom factor selected to keep CPU usage low.

The display task reads the global variable and then plots the data applying an auto-contrast algorithm. The algorithm consists of passing the image pixel values through a look-up table to adjust its dynamic range. Two algorithms are available to produce the look-up table: linear and sigma. The linear algorithm takes the max and the min values and draws a line between them with min being 0 and max

being (216 - 1). The sigma algorithm use the mean and the standard deviation and draws a line from one sigma below the mean to two three sigmas above the mean. The look-up table is updated every 500ms.

3.3.6 FWHM & Flux Monitor

The FWHM & Flux Monitor is implemented by the *rtsoft_gui_sub-apertures* VI. The task consumes data produced by the Frame Data Task to plot data describing the spot size and flux for each sub-aperture.

3.3.6.1 Spot Size (FWHM)

Though typically a nonlinear fit approach is the usual way to tackle the estimation of the size of the spots, the real-time software followed a different approach to avoid the intensive calculations and the lack of robustness (no convergence of the algorithm).

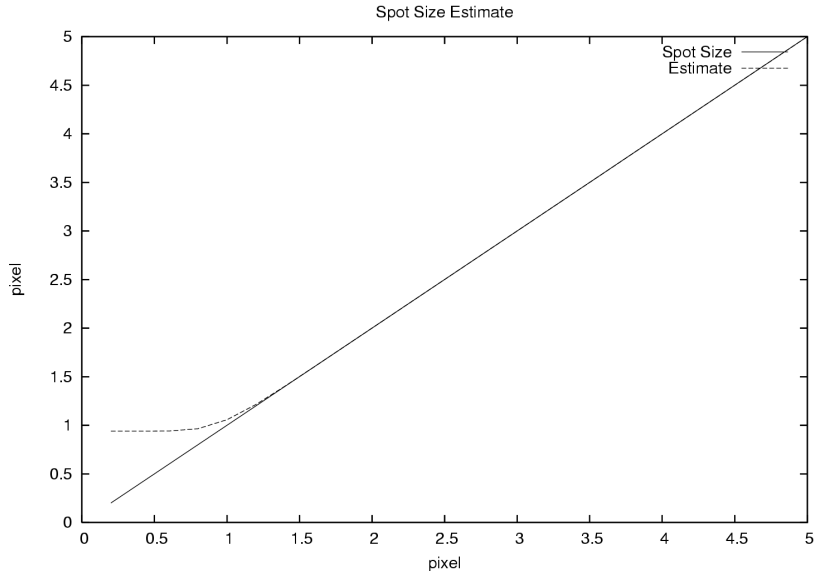

Figure 3.3: Spot size estimate response. The method saturates when the sampling of the spot is less than 1.5 [pixel].

The implementation is provided by the STFLIB module and is based in the total flux for a well sampled Gaussian spot given by (3.1)

$$I_{Total} = I_{max} \times \sqrt{2\pi\sigma^2} \tag{3.1}$$

By first measuring the flux I_{Total} in the sub-aperture and then solving for 燽, the size of the spot can be obtained as (3.2)

$$FWHM = \frac{2.355}{\sqrt{2\pi}} \frac{I_{Total}}{I_{max}} \tag{3.2}$$

The method works reasonably well when the spot sampling is good ($N \geq 2$). For values smaller than that the method saturates. Figure 3.3 shows how the method behave depending on the sampling.

Other limitations that will certainly affect the response of the estimator are the readout noise and the amount of flux collected. Also relevant will be the size of the spot with respect to the sub-aperture size, since for big spots part of their flux will fall outside the sub-aperture boundaries and thus giving a wrong number for the total flux estimation.

3.3.6.2 Sub-Aperture Flux

The sub-aperture flux is read periodically from the global variable *rtsoft_globals_typeII_Flux* which in turn is updated by the Frame Data Task.

3.3.7 AO Loop Control User Interface

3.3.7.1 Interaction Matrix

The interaction matrix H relates the WFS error signal to the DM voltages. The procedure for measuring the interaction matrix is implemented in LabVIEW by the VI *rtsoft_gui_measure_H_full_range*. The algorithm is as follows

1. Set all DM electrodes to mid voltage range value.
2. Poke an electrode to the positive test voltage.
3. Average a user defined number of slope measurements.
4. Poke now the electrode to the negative test voltage.
5. Average a user defined number of slope measurements.
6. Subtract the two slope measurements and divide the result by twice the test voltage.
7. Repeat 1 to 6 for each DM electrode.
8. Reset all DM electrodes to mid voltage range value.

The number of slopes measurements to average is a parameter in the *rtsoft.cfg* configuration file.

3.3.7.2 SVD Reconstructor

The SVD reconstructor is obtained by inverting the interaction matrix using singular value decomposition. All singular values below certain threshold are set to zero. These small values correspond to weak modes that are rejected. A reconstructor with the average slopes subtracted is computed as well for use in the LGS and TSLGS modes.

$$H^* = H^{-1} \tag{3.3}$$

$$H^*_{nas} = (H - AverageTilts)^{-1} \tag{3.4}$$

The implementation of the procedure to obtain the SVD reconstructor is done in LabVIEW by the VI *rtsoft_rec_SVD_matrix_inversion*.

3.3.7.3 Modal Reconstructor

When using a modal reconstructor the error signal delivered by the WFS is converted into modal amplitudes by the inverse gradient matrix ***G****, and then the amplitudes are weighted and transformed to voltages by the inverse projection matrix ***P****.

The implementation of the procedure to obtain a modal reconstructor matrix is done in LabVIEW by the VI *rtsoft_rec_modal_reconstructor_P_unknown.vi*.

The projection matrix ***P*** is first approximated by multiplying the inverse gradient matrix ***G**** and the interaction matrix ***H*** (3.5). Then the ***P*** matrix is inverted using singular value decomposition; singular values below a user defined threshold are dropped. The inverse projection matrix ***P**** is then multiplied by the inverse gradient matrix ***G**** to obtain the modal reconstructor ***H****[8]. A reconstructor with the average slopes subtracted is computed as well for use in the LGS and TSLGS modes.

$$P = G^* H \tag{3.5}$$

$$H^* = P^* G^* \tag{3.6}$$

$$H^*_{nas} = P^* (G - AverageTilts)^{-1} \tag{3.7}$$

3.3.7.4 Optimal Modal Reconstructor (Deprecated)

The implementation to obtain the modal reconstructor in section TBD, uses the simple least-squares inverse to obtain the **G***. The optimal implementation takes into account the known information on the statistics of the disturbance signal and the correlated centroid noise (3.8) [9].

$$G^*_{opt} = \left(G^T C_n^{-1} G + \alpha C_{Noll}^{-1}\right)^{-1} G^T C_n^{-1} \tag{3.8}$$

The procedure for obtaining the optimal inverse gradient matrix **G*** is done in LabVIEW by the VI *rtsoft_opt_G**. The VI handles both optimal and least-squares inverse. The result is then fed to the VI *rtsoft_rec_modal_reconstructor_P_unknown* together with the interaction matrix to obtain the modal reconstructor matrix.

3.3.8 AO Loop Data Recorder

The loop data recorder is implemented in LabVIEW by the VI *rtsoft_loop_data_recorder*. The VI implements a state machine with 4 states. Change of state is determined by a binary variable that represents the command to record and to stop. The initial state is IDLE. The state will be IDLE until the rec command is given. At that time the transition is done to OPENFILE state to then fall to the WRITE state. The state will remain at WRITE until the stop command is given. At that time the state will change to CLOSEFILE state to then fall back to the IDLE state.

Data is written to file in binary format as a sequence of data structure elements of type _AO_ERROR [4]. The structure contains 7 elements: a time stamp, the x-y tilt error, the x-y slopes, and the DM voltages.

3.3.9 Reconstruction Error Analysis

Reconstruction error analysis as described in section 3 of document [8], is implemented in LabVIEW by the VI *rtsoft_rec_compute_reconstructor_error*. This VI basically computes the statistics of the residual Zernike amplitudes given by the covariance matrix C_e (3.10). The sum of the diagonal elements of this matrix is also computed to find the effective compensation N_{eff} (3.11).

An important note here. When analyzing the modal reconstructor with average slopes subtracted, the G matrix with its average tilt subtracted is used.

$$E = I - PRG \tag{3.9}$$

$$C_e = EC_zE^T \tag{3.10}$$

$$N_{eff} = \left\{ \frac{\sum C_{e,jj}}{0.2944} + N_z^{-\frac{\sqrt{3}}{2}} \right\}^{-\frac{2}{\sqrt{3}}} \tag{3.11}$$

3.3.10 AO Loop Performance

AO loop performance analysis is implemented in LabVIEW by the VI *rtsoft_task_loop_optimization*. The VI consumes data produced by the loop-data task (see section 3.3.1). The task collects in local buffers (shift registers in the loop) sequences of slopes and voltages until 5 blocks of 512 sets are obtained. At that point the slopes and voltages are converted into sequences of modal coefficients [8] and the averaged power spectrum for each mode is obtained.

The VI implementing the *2T* delay between slopes and voltages sequences deserves a bit of attention. The VI is called *rtsoft_opt_delay_2Dsequence* and its input array is supposed to contain a time sequence of vectors. In the RTSOFT this vectors contain Zernike modes. Typically the size will be 2560x55 (5 blocks of 512 loop cycles, 55 Zernike modes). When the VI first run it creates a vector array of length delay with all its elements equal to the first vector in the input array. The delay is *2T* long for the AO loop. The input vector array is then appended to the vector array just created and the

last *delay* elements (last 2 vectors typically) are deleted. That last part is saved in a shift register for the next time this VI gets called.

3.3.11 Modal Coefficients Charts

The real-time software is capable of presenting the user with the modal coefficients representing both the uncompensated (open loop modal coefficients) and the compensated atmospheric disturbance (WFS modal coefficients).

The implementation of the procedure to obtain the modal coefficients is done in LabVIEW by the VI *rtsoft_opt_modal_coef_reconstruction*. The slopes are multiplied by the inverse gradient matrix **G*** and the voltages are multiplied by the projection matrix **P**. The open loop modal coefficients are obtained by adding those two results. A *2T* delay is considered in between slopes and voltages to take into account the AO loop delay (3.12) [8].

$$a_{OpenLoop}(k) = G^{*}s(k-2) + Pv(k) \tag{3.12}$$

The WFS modal coefficients are simply the first term in equation (3.12), that is the result of multiplying **s** and **G***.

3.3.12 Noise Propagation

The implementation is provided in LabVIEW by the STFLIB module. The VI *stf_modal_noise_covariance_matrix* returns the diagonal elements of the modal noise covariance matrix C_a (3.13), where C_n is the covariance of slope measurements errors.

$$C_a = G^{*}C_n(G^{*})^{T} \tag{3.13}$$

If the C_n matrix is not available (e.g. all errors are assumed uncorrelated) then the noise propagation coefficients returned by the VI are simply the diagonal elements of $G^{*}(G^{*})^{T}$.

3.3.13 DM Voltages User Interface

3.3.13.1 ROTFLAT

This command implementation loads and apply the previously recorded flatten DM voltages including a correction delta to compensate for flexure at different rotator angles. The voltage to apply to a given electrode is calculated as

$$V_i = A + B\times(1 - \cos(ROT)) + C\times\sin(ROT) \tag{3.14}$$

where *A* is the voltage as recorded in the file *flattendm.cfg*, and B and C are fixed coefficients recorded in the file *Dmflat_rot.txt.*

3.3.13.2 Static Aberrations

The implementation is done in the VI *dm_Zernike_to_DM_voltages*. The procedure consists of multuplying the projection matrix P by a user defined array of standard Zernike coefficients. The resulting vector is then added to the current electrodes voltages.

3.3.13.3 Mirror Shape Display

The implementation is done in the VI *dm_mirror_shape*. The procedure consists of multiplying the current voltages by their corresponding influence function (pre-loaded at startup) and then adding them all. The result is then passed to a standard LabVIEW intensity graph for display.

3.3.14 TT Loop User Interface

3.3.14.1 Bias calibration

The implementation is done in the VI TBD. The procedure consists basically in passing to the RTCORE the average flux over a few seconds for each APD channel as computed by the TT loop performance task. Is up to the user to ensure the collected values are representative of a bias sample.

The implementation also takes care of storing the values in a text file for persistence so next time the RTCORE Labview™ application runs the last bias calibration is correctly applied.

Manipulation of the bias subtraction state and values is done using the **tt bias** and **tt setbias** commands.

3.3.15 TT Loop Performance

TT Loop performance analysis is implemented by the *rtsoft_task_tt_performance* VI. The VI consumes data produced by the TT Loop-Data Task (see section 3.3.2 above) once a second and records the average value for several parameters including flux, loop period and X-Y error. The task collects in local buffers (shift registers in the loop) sequences of X-Y errors until 5 blocks of 512 are obtained. At that point the average power spectrum is obtained and plotted.

3.3.16 “Wobble” Tool

An action command was created to support the “wobble” tool to determine the seeing-dependent coefficient (gain) between dimensionless signals from the guide probes quad sensors and the actual displacements in arcs-seconds.

The command MODULATE has three input parameters: radius R (in arc-seconds)m temporal modulation period P (seconds) and the number of modulation cycles NC. The output are three arrays A, B and C with 8 elements each.

The command is implemented by the *parse_command_modulate* VI. The VI configures the

RTCORE to start wobbling the M3 and reads data produced by the TT Loop Data task. The VI then loops until the specified amount of samples is read and calculates the output arrays.

3.3.17 Mount Control Loop Task

The Mount Control Loop Task is implemented by the *rtsoft_task_mount* VI. It samples at 1Hz the output of the mount filter in the RTCORE and sends it in units of arc-seconds to the TCS.

3.3.17.1 X-Y SAM coordinates to Ra-Dec SOAR TCS coordinates

For doing its calculations the RTCORE use Guide Probes x-y coordinates to reference the Mount commands, independent of the error source selected (2.16).

For sending the Mount command the RTSOFT use SOAR TCS Ra-Dec coordinates. For SAM that relation is given by (3.15) where *PA*[7] is the instrument position angle reported by the TCS. No negative feedback here because the TCS will add that on its side.

$$\begin{bmatrix} \Delta\alpha \\ \Delta\delta \end{bmatrix} = \begin{bmatrix} \cos(-PA) & -\sin(-PA) \\ \sin(-PA) & \cos(-PA) \end{bmatrix} \begin{bmatrix} \Delta X_{GP} \\ \Delta Y_{GP} \end{bmatrix} \tag{3.15}$$

3.3.18 LLT Control Loop Task

The LLT Control Loop Task is implemented by the rtsoft_task_LLT.vi VI. It samples at 1Hz the output of the LLT integrator in the RTCORE and sends it in units of TBD to the LLT control system.

3.3.19 Command Parser Task

The application implements a command parser task to support user commands and macros. The task is implemented by the *parse_task* VI. A list of the available commands follow

AO <OPEN | CLOSE | STATE > - Open/Close the AO loop.

BGND <ON | OFF> - WFS background subtraction.

BIAS <ON | OFF | ACQ | APPLY | STATE > - WFS bias subtraction control.

ACQ <*frames*> - Takes a WFS bias calibration by averaging *frames* number of frames and then save the result to disk using two names: the protected name *wfsbias.fits* and the date formatted name *wfsbias-YYYYMMDDTHHMMSS.fit*

APPLY – Loads the file *wfsbias.fits* from disk and sets the RTCORE to use it.

CAMERA <exposure | ldf | power | readout | reset | set | mancmd | STATE >

exposure [stop|abort]

7 Angle between North and and Y_{GP} axis

ldf [util|pci|tim]

power [on|off]

readout[abort|stop-idle]

set binning <val1> <val2>

set exposure <val>

set frames <val>

set gs <val1> <val2>

set readout <val>

set size <val1> <val2>

set trigger <val>

set roi <val1> <val2> <val3> <val4>

set panoramic]

DM <APPLY | FLATTEN | RECFLAT | ROTFLAT | STATE >

FLATTEN – Load and apply the voltages stored in the file *flattendm.cfg* to the DM.

APPLY <V_0 V_1 V_2 ..... V_{59}> – Apply the given voltages to the DM

RECFLAT – Save the currently applied DM voltages to disk using the protected name *flattendm.cfg* and *flattendm-YYYYMMDDTHHMMSS.cfg*

ROTFLAT – Load and apply the voltages stored in the file *flattendm.cfg* including a correction factor based on the current Nasmyth rotator mechanical angle, and coefficients in file *DMflat_rot.txt*. See section TBD for a detailed description.

ECHO <ARG_1> <ARG_2> ... <ARG_N> - Return all arguments.

EXEC <MACRO> - execute the specified macro.

FLATTEN – Deprecated, use “DM FLATTEN” instead.

LOAD [APDMAP | WMAT | XGRID | YGRID]

APDBIAS <FILENAME> - Load the bias for each APD.

APDMAP <FILENAME> - Loads the quad sensor geometry

WMAT <FILENAME> - Load the weights for WcoG.

XGRID <FILENAME> - Load x relative positions for WcoG.

YGRID <FILENAME> - Load y relative position for WcoG.

LLTM1 <CLOSE | OPEN> - LLT M1 loop control.

LLTM3 <CLOSE | DISABLE | ENABLE | OPEN> - LLT M3 loop control.

OFFSET <EL> <AZ> - Add the specified offset to the LLT M3 axis (amplification 10x from low to high voltage). With not arguments return the currently applied values.

M3 <CLOSE | DISABLE | ENABLE | OPEN> - SOAR M3 loop control.

MODE <TSNGS | NGS | TSLGS | LGS> - Set RTC operation mode.

MODULATE RADIUS PERIOD CYCLES – Wobble the SOAR M3 mirror the specified RADIUS in arc seconds, at a given PERIOD in seconds, CYCLE number of times to produce coefficients representing the raw flux, sine modulated flux, and cosine modulated flux for each APD. With no arguments return the last three sets of coefficients produces: A_K, B_K and C_K with K=1..8 (see reference document [11] for details).

MOUNT <CLOSE | OPEN> - SOAR mount loop control.

REC FRAMES AOPERIOD TTPERIOD – Use this command to record FRAMES number of wavefront sensor frames, AOPERIOD seconds of AO loop data, and TTPERIOD seconds of tip tilt loop data. With no arguments the default values are used. Set argument to a negative number to skip recording data for it. The command produce four files:

WF-YYYYMMDDTHHMMSS.RAW
WF-YYYYMMDDTHHMMSS.FITS
AO-YYYYMMDDTHHMMSS.DAT
TT-YYYYMMDDTHHMMSS.DAT

The files are saved to an automatically generated subdirectory located in a root directory set in the RTSOFT configuration file.

RGATE <ON | OFF | MODE | DELAY | DISTANCE | PWIDTH | PPERIOD>

MODE <AUTO | LASER> - Use to select between AUTO and LASER sync modes for ON/OFF pulse generation.

DELAY <VALUE> - In LASER mode the delay in [ns] between the falling edge of the TRIGGER OUT signal and the ON pulse to the Pockels Cell high-voltage driver.

DISTANCE <VALUE> - The delay in [ns] between the falling edge of the ON pulse and the rising edge of the OFF pulse to the Pockels Cell high voltage driver.

PWIDTH <VALUE> - The width on [ns] for the ON and OFF pulses to the Pockels Cell high-voltage driver.

PPERIOD <VALUE> - In AUTO mode the time period in [ns] of the pulse-train.

RMFAULT – Clear alarms.

RNOISE <ADU> - Set the readout noise value in the RTCORE. If the max value in a sub-aperture is smaller than three times this value, then the slopes are discarded and zero is returned instead.

RT <ARG_1> <ARG_2> ... <ARG_N> - RTCORE interface.

SDSU <ARG_1> <ARG_2> ... <ARG_N> - LEACH II controller interface.

SETREF < REVERT | OFFSET> - Without arguments attempts to measure the reference positions for each sub aperture.

REVERT - Use the revert option to remove all user applied offsets to the measured reference positions.

OFFSET < ID | ID_N-ID_M >

ID <OX> <OY> - Add a user defined offset to sub aperture ID. Without arguments return the current offset by subtracting the measured reference position to the current reference position value.

ID_N-ID_M <OX> <OY> - Add a user defined offset to a range of sub apertures starting at ID_N and ending at ID_M.

SLEEP <VALUE> - Wait a number of milliseconds and return.

SYTEM <ARG_1> <ARG_2> ... <ARG_N> - RTC OS command.

TT <PSCALE | PWEIGHT | LOCK>

PSCALE <VALUE> - Set the seeing dependent coefficient that relates TT probes error signals to actual displacements in arc-seconds.

PWEIGHT <VALUE> <VALUE> - Set the weight for each guide probe error signal.

LOCK <0 | 1> <0 | 1> - Control guide probe locking.

TTBIAS < ON | OFF | WHO | STATE >

<WHO> <SAMPLES> – Use to select the number of counts to average to estimate the APD bias. Max number is 6000 and min number is 50. Valid values for argument WHO are 0: BOTH, 1: GP1 only, 2: GP2 only.

TTP1, TTP2 [OPEN | CLOSE] Open/Close the TT loop for probe 1 or 2.

3.3.20 Remote Services

Remote services in the RTSOFT are built on top of the middle-ware provided by the SOAR Communication Library (SCLN). Documentation on the basics of the SCLN module are available at the SAM web site archive.

The RTSOFT provides a single server implemented by the VI *cx_task_server*. Only one remote

connection can be served at a time. The usual client of the RTSOFT will be the ICSOFT [1]. Upon receiving a command the RTSOFT returns immediately a string response acknowledging the command. The response starts with one of three possible tokens: DONE, ACTIVE or ERROR, followed by command specifics arguments.

All local services are available through the remote services interface. All of them are served in “fire and forget” mode to avoid blocking the remote client. The only exception is the STATUS2 command.

> **STATUS2** - Returns a cluster containing status information as defined in the configuration file *status.ini*. A remote LabVIEW application can borrow that file to decode the cluster and then access its contents.

3.3.21 Simulator Engine

The Simulator Task is implemented by the *rtsoft_task_simulator*. When the camera simulator mode has been selected (command line option -c) the task loops every 10[ms] producing and artificial pattern, passing this pattern to the RTCORE and triggering the RTCORE control logic to produce a slope measurement.

The core of the Simulator Task is the simulator engine VI *rtsoft_simul_engine*. The engine produces a user defined phase screen, and then adds screen to the DM surface based on the current electrodes voltages (3.16). The result is then passed through a Shack-Hartman screen simulator to produce an image.

$$WF_{Out} = 2 * S_{DM} - WF_{In} \tag{3.16}$$

The *rtsoft_simul_engine* VI works close together with the *rtm_rt_service* VI when the RTCORE is not available. When that is the case the *rtm_rt_service* VI emulates the RTCORE by processing the artificial pattern produced by the simulator engine and producing slopes and voltages measurements.

References

[1] Cantarutti R. SDN-8201 Software Overview, February 21, 2006.
[2] Cantarutti R. SDN-8211 Real-Time Computer Software, February 21, 2006.
[3] Cantarutti R. SDN-8210 RTSOFT LabVIEW Application, May 16, 2006.
[4] Cantarutti R. SDN-8212 Real-Time Core Module, February 20, 2006.
[5] Cantarutti R. SDN-8410 RTAI + CentOS 5.8, May 15, 2013.
[6] Cantarutti R. SDN-8401 PowerDAQ Drivers, October 30, 2005.
[7] Tokovinin A. SDN-2306 Characterization of the tilt fiber splitter, March 2, 2006.
[8] Cantarutti R. Rotation Compensation and Gain Factor, December 20, 2005.
[9] Tokovinin A. SDN-1107 Control loop of SAM: modal control, March 8, 2005.
[10] Tokovinin A. SDN-1113 SAM optimization: part II, March 20, 2006.
[11] Tokovinin A. Wobble tool for SAM tip-tilt guiders, December 3, 2010.