The HOTM Command Reference Manual

John Stanley

9 September, 2008

1 Introduction

HOTM is the program developed and used by the Coastal Imaging Lab in all versions of Argus III stations. It is an updated and vastly improved version of the program "m", which was used on the SGI-based Argus II stations. Since the new program works exclusively with IEEE1394 cameras (or "firewire"), it seemed appropriate to name it HOTM.[1]

HOTM was orginally intended to be a one-shot program. It would be run once for every intended data collection set. It was also developed to be modular, so that each collection type would be a loadable library module. E.g., there would be a "timex" library to perform time exposure collections, a "stack" library for stacks, etc. Each library would have a function called "processFrame", which, when passed the input frame, would do whatever it was that was required.

The camera control software was also developed to be modular, so that each model of camera would have it's own library of control functions. Early designs called for functions "init", "getFrame", "releaseFrame", etc.

This framework proved unworkable. It was simply too long a task to start a camera from scratch each time data was needed. Unlike NTSC or PAL video cameras of the Argus II era which provided frames of video at a never-ending 30 or 25 frames per second even when the computer is off, firewire cameras require a controlling process to accept data, and early versions of cameras could not perform autoiris or autoexposure processing on their own while externally triggered. Starting a camera to collect data thus required a significantly long period of time initializing the camera and then achieving a proper exposure setting.

However, it was soon realized that a simple[2]library module could replace the specific processing libraries and turn the HOTM program into a camera control demon. In computer parlance, a demon is a program that runs in the background doing some task or serving some function, starting once and running virtually forever. By running HOTM as a demon process, it could control the cameras continuously and avoid the long startup delays.

[1] In early development of HOTM, a "parent" program, intended to control multiple invocations of HOTM, was developed. It was called, appropriately, HOTMomma.

[2] Ha! Simple he says.

This demon module, in practice, simply loops forever accepting frames and doing one of three things with them. It either ignores the frame (releasing the memory immediately for reuse), saves the frame in raw format to a designated file, or extracts pixels and generates a stack file in rasterfile format. The demon is controlled by a set of commands delivered through system messages. Each demon maintains an input queue to receive messages, and a ten entry list of actions that it has been instructed to perform. Some command messages are immediate, in that they are processed and completed immediately. Some require a one-frame delay prior to completion. The others control data collection and are time-based.

Other than the demon commands which are delivered by system messages, HOTM is controlled by either command line switches or parameter:value pairs read from text files specified on the command line.

2 HOTM Command Processing

Other than the demon commands which are delivered by system messages, HOTM is controlled by either command line switches or parameter:value pairs read from text files specified on the command line. These options and values are read at startup and processed by either the HOTM main code or one of the loaded modules.

The parameter:value pairs are exactly that: a string identifying the parameter, a colon, and a string giving the value. For example. "sitename: argus08". By convention, camera control parameters start with the string "camera", and other module's parameters start with the appropriate string. Because of the large number of symbolic names assigned to IIDC[3] compliant camera parameters, parameters setting camera functions may have string values using those symbolic names instead of the extremely obvious numeric codes they map to.

Parameter names are case-insensitive.

2.1 HOTM Command Line Switches

The following command line switches are available. Note that many of these can also be set by parameter:value entries in input command files. In most cases, any settings here will be made in the parameter:value files.

2.1.1 -d

Turn on the hotm debugging flag. This flag is a boolean contained in the global struct hotmParams.debug. The parameter name for this flag is "debug". I.e, setting this flag in a text command file would require a line like "debug: 1".

[3]Instrumentation and Industrial control working group for Digital Cameras standard for IEEE1394 digital cameras. This standard defines the commands and functions supplied by firewire cameras that are IIDC conforming. The current standard as of this writing is 1.31.

2.1.2 -v n

Sets the verbose level of hotm to 'n'. N is the number generated by or'ing the following flags as defined in hotm.h:

```
#define HVERB_COMMANDLINE      0x0001
#define HVERB_CAMERA           0x0002
#define HVERB_MOMMA            0x0004
#define HVERB_UIDMAP           0x0008
#define HVERB_ALLCOMMANDS      0x0010
#define HVERB_COMMANDFILE      0x0020
#define HVERB_PROCESSIN        0x0040
#define HVERB_PROCESSTIMES     0x0080
#define HVERB_PROCESSOUT       0x0100
#define HVERB_MEMORY           0x0200
#define HVERB_FINDCOMMAND  0x10000000
#define HVERB_LISTCOMMANDS 0x20000000
```

So, to request diagnostic messages about processing times and camera information, N would be 130 (0x80 or 0x02, in decimal.) The parameter name for this flag is "verbose".

2.1.3 -version xxxxx

Sets the program version identifier to the string 'xxxxx'. This string appears in text in the top line of the snap, timex, and variance images. The parameter name for this flag is "programVersion".

2.1.4 -s xxxxx

Sets the site name to 'xxxxx'. This string also appears in the text in the top line of the images, and in the stack file. The parameter name for this flag is "sitename".

2.1.5 -e epochtime

Sets the 'base' epoch time for data collections. Obsolete in the demon version of the program.

2.1.6 -p pid

The process id of the controlling "HOTMomma" process. This is also obsolete in the demon version of the program. Good code is valuable in itself; a good programmer doesn't delete good code just because it isn't needed anymore.

2.1.7 -c n

Sets the camera number for this process. The parameter name for this flag is "cameraNumber". This flag is almost always set on the command line by the script that starts the HOTM processes, since that allows the same parameter:value files to be used for all of the cameras.

2.1.8 Anything Else

Everything else on the command line (not starting with a hyphen) is considered to be the name of a startup command file, and the contents of each file in sequence are read and processed looking for parameter:value pairs. It is irrelevant which file contains which parameter:value pairs, the contents are stored in one linked list and used later, after all files have been read. Setting the same parameter in more than one file is undefined behaviour[4].

2.2 Parameter:Value Pairs

There are a large number of possible parameters. Some deal exclusively with the camera, and standard practice is that they will begin with the string "camera". A module named 'show' would have parameter names starting with 'show'.

Some parameters are HOTM controls. Several of these have been listed under the command line flag section.

2.2.1 Loading Modules

There are two parameter names which control which modules are loaded into the HOTM program.

cameraModule: xxxxxxx

This parameter:value pair causes the shareable library named 'xxxxxxx.so' to be loaded. This library is expected to contain functions dealing specifically with camera operations, and a function to initialize a specific type of camera. The function of those functions will be described later.

Through experience, it has been determined that IIDC cameras can almost all be controlled using one module currently named 'micropix'. The MicroPix camera was one of the first very conforming cameras used, and the module was easily extended to conver newer IIDC cameras.

module: xxxxxx

This parameter:value pair loads the shareable library named 'xxxxxx.so', which is expected to contain a limited set of functions that deal with camera images produced by the camera module. The functions will be described later.

[4]Well, of course the result is determinant since the code does the same thing every time, it's just that I forget what it does (use first or last defined) and don't want people relying on any specific behaviour.

2.2.2 Camera Module Parameter:Value Pairs

Many of the camera module parameters are defined by the libdc1394 parameters available for controlling an IIDC camera. In fact, the camera itself is queried to obtain a list of available parameters, by name. First, the string "FEATURE" is removed from each camera feature, then "camera" prepended to form the HOTM parameter name. E.g., the camera parameter named "FEATURE_SHUTTER" becomes "cameraShutter".

Many of the camera parameter values have symbolic names assigned by the libdc1394 library. These are found in the dc1394_control.h header file. It is certainly more convenient to refer to external trigger mode 0 as "TRIGGER_MODE_0" instead of the number 352, which is the value the libdc1394 library sends to the camera. Similarly, FRAMERATE_3_75 means more to me than the value 33. The function that retrieves parameters which have libdc1394 names uses the file 'dc1394ParamList' to map from names to numbers.

The following are IIDC/libdc1394 camera feature names:

```
FEATURE_BRIGHTNESS, FEATURE_EXPOSURE,
FEATURE_SHARPNESS, FEATURE_WHITE_BALANCE, FEATURE_HUE,
FEATURE_SATURATION, FEATURE_GAMMA, FEATURE_SHUTTER,
FEATURE_GAIN, FEATURE_IRIS, FEATURE_FOCUS,
FEATURE_TEMPERATURE, FEATURE_TRIGGER,
FEATURE_TRIGGER_DELAY, FEATURE_WHITE_SHADING,
FEATURE_FRAME_RATE, FEATURE_ZOOM, FEATURE_PAN,
FEATURE_TILT, FEATURE_OPTICAL_FILTER,
FEATURE_CAPTURE_SIZE, FEATURE_CAPTURE_QUALITY
```

Not all features are implemented in all cameras, and unfortunately, not all features are simple values. The following are IIDC/libdc1394-based HOTM parameters that are not "features" of the camera:

```
cameraMode
cameraFormat
cameraFrameRate
```
[5]

I would refer you to the IIDC 1.31 standard document for a fuller description of each 'feature'. Note that I said "fuller", not "understandable".

White Balance Values

The camera white balance setting consistes of two numbers. One controls the red gain, the other blue. It can be encoded as one number in the parameter:value file, but it is more convenient to split the values into individual settings. HOTM has

[5] Whoa! That looks like FEATURE_FRAME_RATE to me! No. Because this parameter is set in the libdc1394 setup function and not sent to the camera later (when the other parameters are set) I differentiated between the 'feature' frame rate and actual frame rate by leaving out the underscore in the latter. Mode, format, and speed are all set in the same setup call, but there is no underscore in "SPEED" so it looks just like the 'feature' parameter.

added the two camera parameter names MY_FEATURE_WHITE_BALANCE_R and MY_FEATURE_WHITE_BALANCE_B, which map into HOTM parameter names cameraWhiteBalanceR and cameraWhiteBalanceB respectively, to allow this.

Non-IIDC Parameter Names and Values

The following parameter:value pairs are available in the micropix camera module.

```
cameraVerbose
cameraNumber
cameraTop
cameraLeft
cameraX
cameraY
```

The latter four define the image top left corner and width and height when using FORMAT_7.

Example Parameter:Value File

This is an example of a control file for a camera module.

```
cameramodule: micropix
cameraNumber: 1
Cameraverbose: 31
#cameraMode: MODE_640x480_RGB
#cameraFormat: FORMAT_VGA_NONCOMPRESSED
#cameraFrameRate: FRAMERATE_3_75
```

Note that the last three lines have been commented out. How does the camera know what to do without them?

Default Camera Operating Parameters

The default operating parameters for various models of cameras are stored in the file "cameraData", which is read once the camera module knows the model of camera it is controlling[6]. Each line in that file is a colon delimited list of:

```
name
default format
default mode
default frame rate
image width
image height
four format 7 parameters
```

[6]How can it possibly know that? By asking the camera itself, of course. How cool is that?

For example:

```
Scorpion SCOR-14SOC:
FORMAT_SVGA_NONCOMPRESSED_2:
MODE_1280x960_MONO:
FRAMERATE_7_5:1280:960:1280:960:1:1
```

(That's all on one line.) This line instructs HOTM to set the camera into the listed format and mode and frame rate, unless otherwise specified in the parameter:value file. This makes it easier to reduce the default frame rate for all cameras on a system (editing one file instead of one file for each camera). Different models can have different values, of course.

The remaining camera 'features' default to power on values, unless otherwise set. For example, GAMMA is off at power on.

2.2.3 Camera Module Verbose Flag Values

The following are the bit-encoded verbose flag values used in the camera module.

```
#define VERB_START    1
#define VERB_STOP     2
#define VERB_GETFRAME 4
#define VERB_FORMAT   8
#define VERB_FEATURE 32
#define VERB_CAMERA  16
#define VERB_MAPPING 64
#define VERB_AUTO   128
```

2.2.4 The Camera Module Functions

The camera module is expected to contain the following functions.

_init

This function performs all actions necessary to access and initialize the camera. It must convert the requested camera number into the IEEE1394 UID[7], call the libdc1394 functions to allocate memory buffers for images, and set appropriate camera parameters (trigger mode, gain, shutter, etc.) Some of these parameters may be modified by other camera module functions (gain and shutter under

[7]Each device on the firewire bus has a Unique ID (UID) programmed into it. (Some hubs violate this requirement and have UID of 0.) Devices are accessed via this UID. HOTM uses the file 'cameraMapping' to map camera numbers into the UID value. This mapping file contains lines with the format "n: 0xaaaaaaaaaaaa", where 'n' is the camera number and 'a...' is the camera UID. For this reason, it is irrelevant where on the firewire bus a camera is installed, as long as it is communicating it will be found. The UID for a camera can be determined in a number of ways. Perhaps the easiest is to simply plug the camera in and examine the logs ('dmesg' command) to see what UID just showed up. The UID is NOT the serial number written on the outside of Pt. Grey cameras.

autoiris mode for example), but some are static for the life of the demon (frame rate, e.g.) Because of the limitations of the shareable library system, this function MUST be named _init. The remaining functions may have any name; a pointer to each is stored by the _init function.

startCamera

Because there can be significant delay between the _init function and the actual ability of HOTM to process incoming frames, and because this delay can cause buffer overflows, the camera is not actually instructed to start sending frames until this function is called.

stopCamera

This function instructs the camera to stop sending frames. It is called prior to HOTM exit.

getFrame

Retrieve and return the next available frame.

releaseFrame

Deallocate the buffer containing the current image so that the libdc1394 library may reuse it for another incoming frame. If you do not release the frame when finished with it, the buffers will fill and the program will crash[8].

setCamera, getCamera

These two functions are related. One can be used to set specific camera parameters, the other to get them. They allow modules outside the camera module to access camera functions. Their most common use is in the autoshutter code (in demon.so) that changes gain or shutter time values.

2.2.5 Data Processing Modules

Data processing modules are expected to contain the following five functions which will be called by the main loop in HOTM. Even though two are obsoleted by the demon module, all are still necessary. As with the camera module, the initialization function MUST be named _init, and the other four will be accessed by pointers stored by _init.

_init

Allocate any required memory, and store the pointers to the other four functions in a globally accessible struct.

[8] A bad thing.

processFrame

Do whatever it is that is required to the frame passed into this function. This function is called once per frame.

processPixel

This function was intended to be called once per pixel in a frame. Because it is so SLOOOOWWW to process a frame this way, it is obsolete. If you need your user-supplied module to process pixels this way, include this function. Otherwise, have _init set the pointer to NULL.

closeFrame

Perform any functions necessary at the end of pixel by pixel frame processing. NULL if not required.

saveResults

A function that is called when collection is completed and the results need to be stored. Obsolete under demon mode.

3 The DEMON Module

The demon module is the workhorse of the current HOTM system. It controls all data collection, as well as managing things like camera gains and integration times.

The original version of HOTM was intended to include all the data processing functions as loaded modules. When HOTM finished, you would have the results packaged for immediate consumption. When demon mode was designed, it became impossible to predict all the kinds of processing that would be required, and beneficial to allow users to define their own processing if they desired without requiring them to understand the intricacies of HOTM. With that in mind, the demon module was designed to write just one kind of output file: a raw format containing the HOTM parameters and frame data. A followup processing program (currently called 'postproc') was written to process this file and produce the 'standard Argus products' of snap, timex, variance, and more recently bright images. Users can (and are encouraged to) write their own postprocessing program to create their own special products.

However, it was determined that saving a raw format file that would be later processed into a stack was horribly wasteful and would create absolutely huge files that would just be deleted[9]. For that reason, stack creation was moved into the demon module itself.

[9]Determined by actually creating absolutely huge raw files that were just deleted after a few pixels were extracted from each frame. A simply calculation: a normal stack is 1024 frames. A Scorpion camera frame is 1.2Mbytes. A raw file for such a stack is more than a gigabyte in size. Much more efficient to do the stack directly.

The current version of the demon module results in one of two outputs: the requested raw data file or the stack file in rasterfile format.

3.1 Demon Module Parameter:Value Pairs

3.1.1 diskVerbose

Sets the verbose flag for the demon module. See the section on the command line verbose flag for what the values are.

3.2 Dealing With The Devil

Ok, a bit alliterative, but close to the truth. Once the demon module is loaded and HOTM has finished processing all the parameter files, there is no way to change the command line or the file contents. All communication must take place some other way.

The demon module creates a Unix message queue which it listens to to receive command input. Commands are sent in simple text, but require a bit of overhead[10] to indicate length and other parameters. The Perl script "send.pl" (or sometimes just "send") takes care of this overhead. The "send" command has the following format:

```
send cameraNumber command ...
```

where cameraNumber is the physical camera number[11] and 'comand' is the command string. The command can contain spaces. "send" just packs up the entire rest of the command line and sends it to HOTM. The demon is programmed to return a response to each command, which "send" dutifully waits for and reports to the user.

3.3 Demon Commands

All this time, and here's the section you wanted all along. These are the commands the current version of demon understands. Starting with the most important.

3.3.1 Add

Add instructs the demon to collect some bit of image data. It "adds" the requested collection to an internal list of collections. This list is currently statically allocated to contain ten actions. "Add" takes the following parameters:

[10] Nothing in Unix is ever truly simple.

[11] I tried hard to figure out a better way of saying this, but can't. A camera has a 'physical' number, which becomes the message queue identifier, and a 'virtual' number, which appears in the data file names. Because there can be no message queue 0 there can be no physical camera 0, but the camera can be virtual 0. Some management scripts assume all camera numbers are less than 10, thus limiting the range of physical cameras to 1 through 9. More on this later, under 'masquerade'.

```
output file name
number of frames
number of frames to skip
start time (epoch time)
pixel count
pixel list filename (if stack)
pixel deBayer flag (if stack)
```

Note that the last two parameters are used only for stacks. An example of an add command, including the "send" script call, is:

```
./send 1 add /tmp/foo.raw 200 1 120000000 0
```

which tells the demon to collect 200 frames, skipping every other 1, starting at epochtime 120000000. An example of a stack collection is much more complicated and involves counting the pixels and sending them to the demon. For general user interface, the perl script "doCam" manages all of this and more. For example,

```
./doCam 1 200 1 0 myPixList.pix
```

would result in camera 1 being requested to take 200 frames, skipping every other one, starting NOW (0 delay), creating a stack file using pixels in the file myPixList.pix.

Also note that the pixel count parameter is used to signal a stack collection. If the value is 0, there is no stack. If the value is positive, it is the number of pixel pairs the demon will expect to be sent through the message passing system. If the pixel count is negative, then the absoltue value represents the number of pixels the demon will load directly from the file. This is MUCH faster than sending the pixel data through the IPC channel, but requires local access to the file.

Fortunately, doCam takes care of this.

3.3.2 Exit

Self explanatory? Stopping a running demon: "./send 1 exit" stops camera 1.

3.3.3 Ping

A simple test to see if the demon is still running.

3.3.4 Status

A much more complicated test to see if the demon is still running. This command will return a veritable slew[12] of information about what the demon is

[12] As opposed to a veritable slough, which would only get your feet wet as you traversed it. Yes, right now, it is very late in the day. Or very early in the morning.

doing, including shutter and gain values, status of other flags, and a list of collections either scheduled or in process. The output is readable directly, but a script called 'ipcstat' is available that queries all cameras it finds and returns a better formatted output.

3.3.5 endAll

Instructs the demon to end all currently active collections (by setting the number of remaining frames for each collection to 1). If you start a bunch of collections by accident, this command will cleanly terminate anything that is active. It will not end anything not yet active.

3.3.6 cleanSlot

Removes the scheduled collection from the specified "slot" in the list. "./send 1 cleanSlot 4" removes the entry in slot 4 for camera 1. This is how you remove collections that haven't started yet.

3.3.7 autoShutter

Enable or disable the autoexposure processing in the demon. This command takes three parameters. The first is the lower limit, the second is the upper limit, and the third is the "skip" value. Perhaps a longer explanation is necessary? The autoexposure code in the demon[13] operates by averaging the value of every "skip"th pixel and comparing it to the low and high limit. If the average is below the low limit, the autoexposure code first tries to increase the integration time, and if the integration time exceeds the limit (see "intLimit" coming next), it increases the gain. If the average is above the high value, it first tries decreasing gain (until it gets to zero), and then decreases integration time.

Setting the high limit to 0 disables the autoexposure code.

Example: ./send 1 autoShutter 80 120 193

3.3.8 intLimit

Specify the integration time upper limit for the camera auto-exposure mode. In other words, the camera integration time will not be longer than this value in seconds. Caveat: there is about a ten percent overshoot possible in the autoexposure code, and you can SET the shutter to a value longer than this manually. Example: "./send 1 intLimit 0.01"

[13] Why does the demon need autoexposure code? Doesn't the camera have an auto setting for integration time and gain? Well, two reasons. First, some IIDC cameras will not do autoexposure at all if they are in external trigger mode. Second, the autoexposure settings are indirect and difficult to predict results. Third, shutter and gain are recorded with each stack line and raw image frame, so I'd have to ask the camera what they were each time anyway. And fourth, I apparently can't count to three.

3.3.9 AESpot

Obsolete and deprecated command that attempted to emulate the Sony camara, which allowed the user to select regions of the image to control autoexposure.

3.3.10 dumpAutoList

Dumps the pixel list being used to select pixels averaged by the autoexposure code. Interesting for debugging autoexposure code, not so interesting to the normal user.

3.3.11 blockAverage

Average the intensity of all but the first 100 pixels in the next frame and return the value. For the life of me, I can't recall now why I wrote this command then. Maybe I'll remember by the time this manual appears in public. No, still haven't remembered.

3.3.12 offsetTest

Obsolete, but I remember why I wrote this one. Sony cameras (and some firewire controllers included on motherboards) tended to drop packets from incoming frames. Each frame is sent from the camera to the CPU as a series of packets of data, instead of the entire frame at once. If one packet is lost, the image that is produced has an easily detected (by eye) shift. The edges of the image appear in the middle, and in cases of serious shifts, the top and bottom are in the middle, too.

The "offset test" calculated the sum of each image column, then determined the difference between adjacent sums and reported the maximum. In theory, a bad image would have a vertical structure that would result in a large difference. In practice, Point Grey cameras did better, and some scenes contain a vertical structure that fools the offset test. The command remains, the code has been commented out.

3.3.13 pauseRaw

Suspend all raw data file collection.

3.3.14 takeRaw

Take the specified number of raw frames. E.g., "./send 1 takeRaw 20" would take twenty raw frames and then suspend. This command, when combined with 'pauseRaw', allows fine control over data collection, with other parameter changes in between collections, but keeping one data file. E.g., after starting a raw collection of 10000 frames and then immediately issuing a 'pauseRaw', the user (me) could issue gain or shutter setting commands, followed by 'takeRaw 10' to gather ten raw frames. Another gain or shutter change, another takeRaw...

This allows easy programming of diagnostic images (ranges of gain and shutter for the same scene) and results in just one output file to save and process.

3.3.15 stopStack

3.3.16 goStack

Same concept as pauseRaw and takeRaw, except goStack does not take a count of stack lines to take, it just continues the stack that was paused by stopStack.

3.3.17 dumpPixlist

Dumps the contents of the internal pixel list for all of the current or scheduled stacks to the output log file ('/tmp/cam0X.log'[14]). Useful for debugging to see that what you put in is what the demon thinks you said, but not much use otherwise.

3.3.18 lastSeen

Send back the epoch time (in fractional seconds) of the last frame processed by the demon. The output is reported in seconds and microseconds, without a decimal point.

3.3.19 noAutoWhenCollecting

A boolean flag which enables or disables autoexposure mode while a data collection is in progress. "./send 1 noAutoWhenCollecting 1" enables it, 0 disables it.

3.3.20 noAutoOnStacks

Same concept as noAutoWhenCollecting, except limited to stack collections. Note that a stack being collected at the same time as any non-stack will result in no autoexposure for the non-stack data as well. This flag simplifies stack processing significantly if you are trying to do quantitative intensity work. I could have said that about the previous command, too.

3.3.21 flushLog

Close, truncate, and reopen the camera log file (/tmp/cam0X.log). A daily flushLog is good for the soul, and for your /tmp directory too. Notice that some distributions of Linux (or Unix) do a cleanup of /tmp every so often, which can result in the open log file being removed from the directory but not

[14]Did I mention, the HOTM process maintains a log file in /tmp named cam0X.log, where X is the physical camera number. It records several things, mainly autoexposure changes, but also logs missing frames. This log file can grow to be large over time, so see "flushLog" coming up.

from the disk. If you ever see an Argus station where the root disk says 100% full and you cannot find the offending files, try a flushLog.

3.3.22 masquerade

Pretend to be a different virtual camera than your assigned physical number. Use the masquerade number in output file names, and all visible recorded parameters, but keep the same message queue id.

3.3.23 setcamera

Modify the specified camera parameter in the specified way. Ok, more info needed.

Setcamera takes three parameters. The first is the action requested. The second parameter is the feature to be set. The final parameter depends on the requested action. Allowed actions are CAM_SET (set the feature value to the third parameter) or CAM_SCALE (multiply the current value by the third parameter.) For example: "./send 1 setcamera cam_set feature_shutter 0.01" instructs the demon to set the camera shutter to 0.01 seconds. "./send 1 setcamera cam_scale feature_gain 2" causes the camera gain to be set to two times the current value.

This command is critical for setting the camera to a known starting condition following the imposition of an integration limit, for example. You can use the intLimit command to set a limit, but if the camera is already above that limit, it will stay there (until autoexposure brings it down). So, the sequence of commands:

```
autoShutter 80 120 168
intLimit 0.10
setcamera cam_set feature_shutter 0.0001
setcamera cam_set feature_gain 0
```

will put the camera into a valid range for gain and shutter after starting autoexposure and imposing the integration time limit. These commands are normally[15] found in a file named "cam0X.startup" which is processed by the script that initiates the HOTM program[16].

3.3.24 openSHM/closeSHM

The entire next section of this manual (The Show Module) deals with a module written before this pair of demon commands was added.

This pair of commands manages the shared data memory operation of the demon module. OpenSHM creates three separate shared memory areas: the

[15] In CIL-managed Argus stations.

[16] A perl script called startup.pl, which sets a path for library searches, starts the HOTM program (with elevated priority), waits for the demon message queue to be created, and then sends the commands in the cam0X.startup file.

image header, the camera module, and the data sections. The camera module shared memory area is a copy of the cameraModule structure used by HOTM to remember camera parameters. The image header area is a copy of the image header attached to each frame, and the data area is a copy of the frame itself.

The program "postshow" was written to access these shared memory regions and display the image contained therein.

Unlike the 'raw' data file produced by the demon, there is no guarantee that a program accessing data via the shared memory will see every frame. The demon sets a semaphore indicating the data are valid when it is copied into the memory area. The processing program clears the semaphore when it is done accessing the data. If a frame would be collected by the demon while the semaphore is still set, it will not be copied to the shared memory. It will still be written to the raw file.

4 The Show Module

You might have noticed that there is no convenient way to look at the images coming from the cameras anymore. With an NTSC analog camera, you could simply tap the analog signal off and display it on any standard NTSC monitor. You cannot tap the firewire signal. There is no simple display device you can plug a firewire camera into to watch what it is doing.

The 'show' module was written to help with that problem. It is a demonstration of how to write a data processing module and an image display method in one! It uses the SDL (Simple Display Library) to open an X window and then paint each incoming image into the frame. Because it uses X, it cannot be run remotely without an X connection. That means it can be run over a tunneled SSH X connection, but it is VERY slow.

4.1 Show Parameter:Value Pairs

The 'show' module has the following parameter:value pairs.

4.1.1 showModulo

Set a modulo value for the pixel increment. I.e., "every other pixel", "every third", etc. showModule takes a single integer value, defaulting to 1 if unset.

4.1.2 showVerbose

Set the verbose output flag to the value specified.

5 Example Configuration Files

Here are a set of example configuration files typical to CIL Argus stations. The decision of where to put a parameter:value pair is often arbitrary. Remember,

it doesn't really matter in what file they appear, it is a convenience for the admin to keep camera paramters in a camera file, and other module paramteres in another file.

5.1 Command Line Options from Startup.pl

HOTM is executed with the following command line options from normal version of startup.pl, using camera 1 as an example:

```
hotm -c 1 cdisk cam1
```

From that command, you can determine that we need two parameter files, one named 'cdisk' and one named 'cam1'. Startup.pl will also look for the file 'cam01.startup' for runtime camera parameters.

5.2 cdisk

This is an example 'cdisk' file. Note that it contains information common to all cameras at a site.

```
# all common params
sitename: testfoo
programVersIon: argus2.0
debug: -1
verbose: -1
module: demon
diskVerbose: -1
module: show
showmodulo: 3
#module: focus
#diskframes: 30
```

5.3 cam1

This is an example 'cam1' file. It contains paramters specific to the camera.

```
cameramodule: micropix
cameraNumber: 1
Cameraverbose: 31
#cameraMode: MODE_640x480_RGB
#cameraMode: MODE_640x480_YUV422
#cameraFormat: FORMAT_VGA_NONCOMPRESSED #cameraFrameRate: FRAMERATE_3_75
#cameraBrightness: -3
#cameraSharpness: 64
#cameraGamma: 1
#cameraIris: 4
cameraGain: 0
```

```
cameraShutter: 3
#cameraExposure: -3
#cameraWhite Balance: 15388
#cameraHue: -4
#cameraSaturation: 106
```

Note that most of the paramters are commented out. Good programmers don't remove code just because it isn't used anymore. This file overrides the command line specification of camera number; we could comment that line out of this file. The camera gain (0) and shutter setting (3) will be overridden later by the autoshutter code in demon and could be commented out, too. That would leave us with just the module name and verbose flag.

This file also demonstrates the deprecated method of setting camera white balance.[17]

5.4 cam01.startup

This is an example of the 'cam01.startup' file. The lines in this file are read by startup.pl and send as commands to the demon one by one.

```
autoShutter 80 120 168
intlimit .10
setcamera cam_set my_feature_white_balance_r 64
setcamera cam_set my_feature_white_balance_b 64
setcamera cam_set feature_shutter 0.000125
setcamera cam_set feature_gain 0
```

This file sets an integration time upper limit of 0.10 seconds an initial integration time of 0.000125 seconds and a gain of 0dB. Because autoShutter is enabled (the upper limit is greater than 0), the integration time will be slowly increased until the autoShutter boundaries are met (up to the limit of 0.1s), and then the gain will be boosted, if necessary.

The red and blue white balance values are both set to 64. The proper values need to be determined for each camera and site. The best way to do this is to observe non-clipping white objects in the image (i.e., R, G, and B all less than 255). Determine the factor to multiply the red pixel by to make it equal to green. Multiply the current R white balance number by that, and use that in place of 64 for the white balance R. Do the same for the blue. Your results will be very close, but may require some adjustment.

Of course, different models of cameras have different unity levels for white balance. You will need to study the documentation for your camera model to see what the unity value should be, and replace '64' in the above discussion with that. If you use the Point Grey or Coriander camera software, you can also perform white balances and record the exact numbers they report.

[17] And because it is deprecated I am not even going to tell you what the value means or how to generate it. Use the cam01.startup file to set it in a logical way.

5.5 cameraMapping

For HOTM to be able to locate camera 1, there must be an entry for it in the 'cameraMapping' file. Here is a short example.

```
1   0x0001687307420056
2   0x000A4701010520F3
#2  0x00B09D01003FF1D8
3   0x00B09D010041EC93
#2  0x0800460200060936
#3  0x0800460200060938
```

This file comes from a system where cameras are swapped on a regular basis. Each camera has a different UID and requires an entry in this file. If you mistype the UID in this file, HOTM will complain very loudly that it cannot find a camera with the specified UID on any bus. If you fail to have an entry for a requested camera number, HOTM will also complain loudly and refuse to start.

A further note about UIDs and cameras and the firewire bus

Some Argus Stations require multiple IEEE1394 interface cards. In 1394 terms, each card creates one 'port', also referred to as a 'bus'. All of the firewire connectors on an interface card are connected to the same bus. Each device connected to a bus becomes a 'node'. Whenever a node is connected (or disconnected), the entire bus goes through a reset procedure which may result in the node number changing for some or all of the devices. This reset process is rather complicated and beyond the scope of this manual,[18] but it includes bus mapping and speed detection. Devices are accessed by port and node number. They are identified by their UID and other information that can be read from each device, but all reads and writes of data to devices are based on the port and node number.

NOTE: that's why unplugging a device from a bus with a camera being controlled by a HOTM can, and probably will, cause HOTM to crash. It is trying to talk to a camera at a certain node, and you are causing the node number to change. Eventually, HOTM will stop getting frames from that camera and the program will exit. Maybe.

When HOTM starts, it searches all available ports looking for a node with the UID associated with the camera number in the mapping file. Once it locates the port and node that contain that camera, all further accesses are by port and node numbers.

5.6 cameraData

As part of the data returned to HOTM from the system when it locates the camera with the correct UID, along with port and node, is a string identifying

[18] But not beyond the scope of "FireWire System Architecture" by Don Anderson, published by MindShare.

the camera model. HOTM then scans the file 'cameraData' for default camera information. Here's an example (remember, each model is on one line, but for display here lines have been wrapped.)

```
Scorpion SCOR-14SOC:
FORMAT_SVGA_NONCOMPRESSED_2:
MODE_1280x960_MONO:
FRAMERATE_7_5:1280:960:1280:960:1:1
Scorpion SCOR-14SOM:
FORMAT_SVGA_NONCOMPRESSED_2:
MODE_1280x960_MONO:
FRAMERATE_7_5:1280:960:1280:960:1:1
```

These identify the color and monochrome versions of the Scorpion camera that is standard for Argus Stations. If you have a camera that is not listed in this file, HOTM will first report the model name[19], and then exit because it doesn't find a match.

6 Notes About Important Things That Haven't Been Discussed Yet

6.1 A Note About The Time and How It Is Determined

Raw and stack files contain specific information about the time of collection of each line or frame. How is this time determined, and why can it be wrong?

Each frame coming from a firewire camera comes with a "filltime" parameter. This is a unix-style timeval struct, which gives the epoch time in seconds and microseconds. To determine the time, add the seconds value to the microseconds value divided by 1,000,000. This time is dependent upon the CPU system clock AND the amount of time required to send the frame across the bus. It does NOT represent the actual time the frame was collected. As a first approximation, early versions of hotm simply subtracted half the integration time from the filltime, assuming that the transmission time was consistent during collection.

This method has issues, especially when trying to correlate hotm data with other collection systems.

As a second approximation[20], hotm attempted to determine the true bus transmission time for each frame. Here's how. Each frame is broken down into packets. If you know the packet size (or can guess it) you can know how many packets there are. Each packet requires one bus "cycle" to be transmitted. In fact, the timing of one bus cycle plays a large part in the amount of data that can be sent across the bus, since packets cannot take longer than one bus cycle. Each bus cycle is 125 microseconds long; alternatively, there are 8000 cycles per

[19] You may need to set verbose mode to cause camera parameters to be listed to get the model name output.

[20] in a version never used in public, for reasons that will become clear.

second. If you know that your frame is broken down into 100 packets, then you know it took 100/8000 second to cross the bus.

Unfortunately, standard formats for Pt. Grey cameras do not allow one to query the size of packets. Packet size may be queried under FORMAT 7. Without knowing the packet size, one cannot calculate the number of packets per frame.[21]

While Pt. Grey takes some things away, it also gives us something of value. Pt. Grey cameras have a mode in which they insert certain camera data into the first few bytes of every frame. This includes the cycle time that the frame was collected (shutter close). This time is used by Pt. Grey cameras to allow multi-camera synchronization between free-running cameras. It consists of three parts. 1) a second count (0-127), 2) a cycle count (0-7999), and 3) a cycle offset (0-3071) counting a 24.576MHz clock. This cycle time information is maintained from every object on the bus that deals with isochronous transfers, including the bus interface itself.

By itself, this information is insufficient to determine real time. To tie the cycle time value to real time, one must query the camera cycle time register and the system epoch time simultaneously. Then it is trivial to compare the frame time to the cycle time at the known system time and correct the system time appropriately. To reduce errors due to bus and system latency, the cycle time is queried both before and after a request for the system time and the two values averaged. In algebraic terms:

```
E = S + F - (C1+C2)/2
```

where S is the system epoch time, C1 and C2 are the two camera cycle times, F is the cycle time recorded in the frame, and E is the true epoch time when the frame was collected. C1, C2, and F all have the possibility of overflowing, and this is dealt with in hotm. To give the mean time of collection (average of open shutter and close shutter), one half the integration time is subtracted.

One important note for Pt. Grey cameras is that free-running cameras synchronize themselves so the CLOSE of shutter occurs at the same time on all cameras. (Because free-running cameras on the same bus sync automatically, more recent Argus Stations have omitted the expensive external trigger hardware.) Only when using external trigger mode do the cameras OPEN the shutter at the same time.

This latter method is now being distributed to Argus Stations and will become the standard.

[21] And because the number of bytes per packet must be the same for every packet, and must also meet other bus-related limits, it is impossible to know exactly how many bytes are sent across the bus for each frame. It is often more than the simple multiplication of width times height times bytes per pixel.

6.2 A Note About IEEE1394 Busses and Camera Frame Rates

Here's a can of worms. See them squirm? Each firewire bus (or port) has a maximum data rate which limits the frame rate that you can use. These are some of the considerations.

A 400MBit/s firewire bus has a limit of 320MB/s for isochronous data. 'Iso' data is data that is transmitted in a time-sensitive way. That includes camera image data. (Camera commands are asynchronous data.) A camera is instructed to take frames at a certain rate. Even a camera that is told to operate in "external trigger" mode has a regular frame rate assigned, either by default (fastest rate for a commanded mode) or explicitely (by command from HOTM). As long as the camera is able to achieve that rate, frames will be ready to send whether anyone has done anything with the previous frame or not. If the bus is busy doing something else, you can lose frames. To avoid this issue, 80% of the bus bandwidth is allocated to "iso", or time-dependent data. The remainder of the bus is allocated to asynchronous data, such as that from disks. Disk data can be buffered and if you don't get a block right now you don't lose the following one.

Each camera knows that it must transmit its data within a time period less than the frame rate. For cameras operating in triggered mode, the actual frame rate may be anything from zero up to a maximum of the programmed frame rate. I.e., a camera set to send 3.75fps and then put in external trigger mode cannot be triggered faster than 3.75fps, and that camera assumes that it has just 1/3.75 seconds to send it's frame even if it is being triggered at only 1/10 Hz.

A Scorpion 14SO camera has 1280 by 960 pixels, for a total of 1,228,800 bytes. (In 16 bit mono mode, it will require 2,457,600 bytes to send a frame.) That frame at 3.75fps results in a data rate of 4.6Mbytes/s (almost 37MBits/s). At 7.5fps, that's 74MBits/s. On a bus that allows 320MBits/s, you can have only 4 such cameras running at the same time before you overflow the bus capabilities, if they are set to 7.5fps. You get 8 cameras on a bus at 3.75fps. But remember that applies to the Scorpion in 8 bit mode. You can have more than 8 cameras if the frames are smaller.

Uhhh, no. Sorry. Thanks for playing. There's another problem. It's buried deep in the technical standards. Each camera sending data on a bus requires a set of control registers in the host firewire interface called an 'isochronous receive context'. This 'context' controls and defines the isochronous transmission. The Open Host Controller Interface (OHCI) standard calls for a minimum of four receive contexts in a conforming interface – and that's what most OHCI interface cards implement. Four. This is a hard-limit on the number of running cameras on a bus controlled by that interface[22].

There are OHCI controllers that have 8 contexts. Unfortunately, almost

[22]There are other limits on the number of NON-running cameras on a bus, but we've yet to reach that limit. The ISO context limit applies only to cameras that are actively being controlled by a HOTM process.

NO vendors of IEEE1394 cards ever tell you this number, or which integrated circuits ("chipsets") are being used to produce the card (and by looking them up online, the number of contexts.) At least one scurilous chip manufacturer actually reduced the number of contexts in its product with NO notice and NO change in the chip identifier. Just BLAMMO! One copy of their interface card WOULD run 8 cameras, and another apparently identical one would NOT.

One chipset that does support 8 contexts is produced by Agere, called the FW323. One online dealer actually announces that they are selling "Agere" controllers. We loves them. They deserve a place in Heaven.

What is the point of this section? The more cameras you need to support on a system, the slower they must be set to run, or you need to install more 'ports'. The number of ports is limited by the number of expansion slots (PCI, usually) the computer has available. One current Argus system has three ports installed. Two ports is not unusual.