The HOTM Command Reference Manual

John Stanley

07/09/2018

Contents

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.

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.

HOTM has now been expanded to deal with GigE cameras in two different versions. The Point Grey cameras that use a control library called ’flycap’, and the newer versions that use the Spinnaker library. While implementing these versions, as much of the existing infrastructure was reused as possible. If you ever look at the code you may wonder why something is done the way it was. It’s historical.

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

[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.

[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.

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".

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'. If you use the same parameter in multiple files ("cdisk" and "cam1", e.g.) the result is undefined. You will get one or the other, never both.

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. There are currently two GigE camera modules based on the two control libraries: gige.so and gigeSpin.so. Remember to leave off the ".so" when entering the name as the cameraModule value. [5]

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. There is a table that can contain up to ten such modules; in normal use, you need just one.

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".

[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.

[5]A reported bug in the Point Grey Spinnaker library prevents it from being used in a loadable module, so there has been a temporary modification to the structure of the HOTM program to deal with this. You cannot set the cameraModule value to "gigeSpin", it must be set to "linkedCamera", and the hotm program itself must have gigeSpin linked in. The makefile handles this; it creates an executable called "hotmSpin", and the link that points to src/hotm must point instead to src/hotmSpin.

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
```
[6]

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 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.

[6] 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.

```
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[7]. 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
```

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
#define VERB_SET    256
#define VERB_TIMING 512
```

2.2.4 The Camera Module Functions

The camera module is expected to contain the following functions.

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

_init

This function performs all actions necessary to access and initialize the camera. It must convert the requested camera number into the IEEE1394 UID[8], 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 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[9].

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]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.

For GigE cameras, the identification IS the serial number.

[9]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 DEMON2 Module

The demon2 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 now-obsolete demon.so module is still in the repository, but it has been superseded.

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[10]. For that reason, stack creation was moved into the demon module itself.

Also, as processor speeds improved, it became more conventient for HOTM to produce the image products directly, and the current version of HOTM does this. (There is a flag to select 'png' format instead of the defalt 'jpg'.) When running HOTM on a lesser-powered computer, or in an environment where any delay in collecting frames can result in dropping them, HOTM can also output a format now called ".dump". Instead of all the images, it contains the intermediate data (snap, sum, sum of squares) from which the image products can be calculated, if necessary. If you want the raw data, that can also be output.

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. Yes, as complicted as the demon module is, it has just one parameter:value pair. How DO we control it, then?

[10]Determined by actually creating absolutely huge raw files that were just deleted after a few pixels were extracted from each frame. A simple 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.

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[11] 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[12] 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 some of the most important commands the current version of demon understands, starting with the most important. Many of the existing commands are obsolete or so special purpose that you will never use them or need them. There will be information at the end of this section on how to determine what commands haven't been documented here.

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. I.e., you can have up to ten collections scheduled or active at one time. [13] "Add" takes the following parameters:

```
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 for camera 1 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,

[11]Nothing in Unix is ever truly simple.

[12]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'.

[13]Because the output file names are based on when the collection takes place, it is really bad karma to schedule two things for exactly the same time. For example, if you schedule one ten minute timex to start at minute 0 of the hour, and a 20 minute timex to start at the same time, you will probably get one timex when all is said and done – the twenty minute one. Depending on the order that data is transferred back to your host system, you may get the ten minute one. The solution is easy: schedule one for minute 0, one for minute 1.

```
    ./doCamAndSend 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. 'doCam' manages this for you. It will instruct the demon to load the pixel list directly from the file you tell it.

NOTE: there is a "../" prepended to the pixel list. Why is that? Pixel lists are intended to be stored in the "hotm" or "hotm2" directory. 'doCamAndSend' is a wrapper around 'doCam' that manages the collection and tranmission of data. As such, it creates a temporary working directory for HOTM to collect data into, tells the system to send that data to the remote host, and then deletes it. Since the program is operating in the sub-directory, you must tell it to look "one level up" for the pixel list.

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. It returns "pong".

3.3.4 Status

A much more complicated test to see if the demon is still running. This command will return a veritable slew[14] of information about what the demon is 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. The autoexposure code in the demon[15] 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

[14]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.

[15]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.

3.3.8 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"

3.3.9 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.

3.3.10 pauseRaw

Suspend all raw data file collection.

3.3.11 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.

3.3.12 stopStack

3.3.13 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.

3.3.14 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'[16]). Useful for debugging to see that what you put in is what the demon thinks you said, but not much use otherwise.

3.3.15 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. There is an ancillary script called "lastS" that asks each camera in turn for the last seen frame using this command, adjusts each value based on the expected frame rate (e.g. 500,000 microsecond period), and reports any large errors. This is useful for checking of all your cameras are truly synchronized to each other.

3.3.16 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.

[16]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.

3.3.17 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.

3.3.18 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 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.

3.3.19 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. This command is present so that you can have a camera number 0 if you want.

3.3.20 setcamera

Modify the specified camera parameter in the specified way.

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), CAM_ADD (add the value) 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[17] found in a file named "cam0X.startup" which is processed by the script that initiates the HOTM program[18].

3.3.21 openSHM/closeSHM

This pair of commands manages the shared data memory operation of the demon module. OpenSHM creates three separate shared memory areas: the image header, the camera module data, and the image 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.

[17]In CIL-managed Argus stations.

[18]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.

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.

3.3.22 doDump

Setting this flag to "1" (./send 1 doDump 1) will tell "hotm" to output the semi-raw "dump" file containing intermediate image data. This is faster than doing the full processing, and may help if you see a lot of dropped or missed frames in your collection. Setting it to "0" will return to normal internal processing.

3.3.23 writePNG

Setting this flag to "1" will force "hotm" to create PNG output images instead of JPG. Zero does the opposite. This option was added when support for a 12 bit image from an IR camera was needed; the 12 bit code has not been tested recently and may require update.

3.4 Undocumented Demon Features

Not bugs. Commands that aren't listed here. They're pretty easy to add, so you may have a version of DEMON2 that has commands not listed here. How do you find them?

Fortunately, you have the source. Just look there. To help you out, here's a shell command you can use to look for commands. The command parser has been programmed in a similar way for each command it knows, so a simple search can find them all.

```
grep strcasecmp demon2.c
```

This will produce a series of lines such as:

```
else if( !strcasecmp( "setcamera", token ) ) {
```

This starts the section of code that deals with the "setcamera" command. If you look in the source you can see what it does.

3.5 Undocumented Camera Features

Just as there is a way of finding undocumented demon commands from the source, you can identify camera parameters (for parameter:value pairs) from the camera demon source. You need to identify which camera source you need to look at. For example, for firewire it will be "micropix.c". For Spinnaker GigE, it is "gigeSpin.c". This command will show you the available parameters:

```
grep get gigeSpin.c | grep Param
```

All parameter:value lookups are done using one of three functions with the names "getXParam", where "X" is either "Long", "Double", or "String". The grep command will return lines like:

```
getLongParam( &cameraModule.cameraNumber, "cameraNumber", module );
```

This asks for the parameter "cameraNumber" (case insensitive) and stores any value it finds in the variable cameraModule.cameraNumber.

The specific parameters will depend on the camera contol library and the camera type. For example, the Flycap Point Grey library had an issue with determining the correct Bayer pattern for the sensor; there is a "cameraRawOrder" string parameter to allow setting this. The Spinnaker library does not have this problem and that parameter is not in the gigeSpin.c file.

By the way, undefined or misspelled parameters are ignored. The code asks for the values it wants to find when it needs them, it does not process every line as it comes in.

4 Postshow

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 or GigE signals. There is no simple display device you can plug a firewire camera into to watch what it is doing.

The program “postshow” provides this function. It connects to the shared memory segments that were created by the HOTM demon when the “openSHM” command was received, formats the raw image data according to the collection parameters, and displays it on the X console.

There are two parameters to “postshow”. The first is the desired camera number. It is specified by using the “-c” option. You can watch more than one camera at a time, using multiple “postshow” commands.

The second option is a “modulo” value. This sets the subsampling of the image before it is displayed. Using “-m 2”, for example, will sample every other pixel (every 2nd). A camera that is collecting a 2448x2048 image will be displayed as 1224x1024. You may want to use values of ’3’ or ’4’ depending on your terminal. Because “postshow” uses standard X windows display commands, you can view images through a remote X session (“slogin -XY my.data.station.com”) or on a VNC remote display. Using larger numbers for the modulo will improve transmission speeds.

Example: “postshow -c 2 -m 3”

A known issue with “postshow” is that clicking on the “X” in the display window will not kill the program. You must control-C or otherwise end the program from the command line where you started it.

5 Important Helper Scripts

Starting and controlling “HOTM” is somewhat complicated. To make life easier, it comes with several scripts to perform common tasks. These can be modified locally once you understand what they are doing.

5.1 startup.pl

This script starts up an instance of “hotm”. It takes one parameter on the command line: a list of camera numbers. It handles several important tasks for you.

1. Sets a pointer so that the required shared modules (camera, demon, etc) can be found.
2. Executes the “hotm” program, setting the camera number (-c), and passing it the “cdisk” and “camN” file, where N is the camera number.
3. Waits until “hotm” has woken up enough to create the message queue (and is ready to accept commands.)
4. Sends the commands contained in the file “cam0N.startup”, where N is the camera number.
5. Sends the commands contained in the file “cam0X.startup”, where X is a literal “X”.
6. Loops back to do the same for the next camera number on the input list.

5.1.1 Common Failures

“startup.pl” can fail in the most magical and silent ways. The most common failure is if “hotm” cannot execute at all and dies with a “segmentation fault” or other major system error. “startup.pl” will simply sit waiting for a queue that never happens. If you see this result, you may need to test “hotm” yourself, manually, to see what is wrong. You can do this under the bash shell using the following command:

```
LD_LIBRARY_PATH=. ./hotm -c 2 cam2 cdisk
```

Note that YOU must tell “hotm” where to look for modules (the LD_LIBRARY_PATH setting) and then give it the camera number and control file names. The order of the files on the command line is not relevant.

You may also see a response from “hotm” like: “hotm2: cannot find any cameras! whazzap?” You’ve got a problem in your hardware.

5.2 doCam/doCamAndSend

The 'doCam' script manages the collection of data, converting a relatively simple set of commands from the user into the "right stuff" for the demon to act upon. It takes the camera number, number of frames to collect, a "skip count" (i.e. "take every Nth frame"), either an absolute or relative time to start, and an optional pixel list, and creates the appropriate "add" command to send to the demon. E.g.:

```
doCam 1 1200 0 minute
```

or:

```
doCam 2 2048 0 10 1430000000.c1.pix yes
```

The first command results in "hotm" for camera 1 being instructed to do a 1200 frame (10 minutes at 2Hz) standard image collection starting at the top of the next minute. The second version instructs camera 2 to start a 2048 frame stack collection in 10 seconds.

There are two "word" times. One is "minute", which is "start at the top of the next minute", and the other is "hour", which does the same for the top of the next hour.

The script "doCamAndSend" is a wrapper around doCam that creates a temporary working directory and then sends the data to the host system (using the script "send_data") after the collection ends.

There are two optional flags to doCam (and doCamAndSend). If you want to collect raw frames, use the "-r" flag. If you want to produce stack output in netCDF format instead of the older raster file, use the flag "-ns". For example:

```
doCam -ns 2 2048 0 10 1430000000.c1.pix yes
```

5.3 ipcstat

To determine the running status of all your "hotm"s, use "ipcstat". This script will look for all message queues that should have a running "hotm" listening, determine if there is a "hotm", and if so will show the status from that hotm. If a process is missing, it will report that.

5.4 statusCheck

This script protects you from trying to run multiple copies of hotm on the same camera. This is not detected by startup.pl, and it will cause the other copy to crash as the new copy tries to take control of the camera. To avoid this, use "statusCheck" instead of "startup.pl" for regular operations. It takes the same list of cameras to check, and if there is no "hotm" running for that camera it will run startup.pl for it.

5.5 send_data

This is the script that takes your Argus data and causes it to be sent to the host system. At its simplest, it is just a UUCP command to copy the data. With a bit more embellishment it will compress stack files (that need it), send image data to a high-priority UUCP queue, send stack and other data to a lower priority queue, copy all the data to a temporary holding area as protection against UUCP losing something, and then delete the originals. It takes no parameters.

5.6 argus.expire

When you have a backup area where copies of your data are stored "just in case", you MUST clean that area up every so often. Argus.expire takes two parameters. It needs the number of days of data to keep, and a directory to scan for expiration. E.g.:

```
argus.expire -7 sent
```

will scan the subdirectory "sent" for anything older than 7 days and delete it.

6 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.

6.1 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'. Also, hotm needs two camera definition files, 'cameraMapping' and 'cameraData'. Startup.pl will also look for the files 'cam01.startup' and 'cam0X.startup' for runtime camera parameters. Each of these files is represented in the SVN repository with as an ".example" file to help you get started. YOU MUST CREATE THE ACTUAL FILES before you can run "hotm".

6.2 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: argus3.0
debug: -1
verbose: -1
module: demon2
```

6.3 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
```

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.[19]

[19]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.

6.4 cam01.startup

This is an example of the 'cam01.startup' file. The lines in this file are read by startup.pl and sent 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.

For GigE/Spinnaker cameras, the white balance numbers are ratios and not absolute values. A value of "1.5" means amplify the red or blue channel by 1.5. It has also been noticed that the Spinnaker library appears to handle automatic white balance even when the cameras are externally triggered, and currently the fixed white balance code in hotm is not active.

6.5 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
4  1432567
```

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.

Camera 4 in this example is a GigE camera where the value is the serial number.

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,[20] 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. And you may find that ALL cameras on the firewire bus will crash, because ALL of them may be reassigned their port and node when you unplug one or plug one in.

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.

6.6 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 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[21], and then exit because it doesn't find a match.

7 Extending HOTM

You may have already looked at the code in demon2.c to stare in wonder at the beautiful code contained therein, which is designed to do all of the standard things that an Argus station need to do. It accepts commands to take data, calculates images (like the variance image), outputs stack files, and many other wonderful things. If you look at HOTM from the outside, you may think that this is the right place to add your extended functions because this is the part of HOTM that loops forever. But you would be wrong.

Demon2 is a plug-in module. When your "cdisk" file contains the line "module: demon2", it is telling HOTM to find the sharable library "demon2.so" and load it. By default, this executes the function "_init" contained in that library. Part of the initialization is this small snippet of code:

```
me->processFrame = _processFrame;
me->processPixel = NULL;
me->closeFrame = NULL;
me->saveResults = _saveResults;
```

What this code does is save pointers to two functions within "demon2" in a "processModule" table. And shortly thereafter, "_init" returns. How is the actual processing in demon2 initiated?

Once HOTM has finished setting things up, it enters a loop called "processLoop". The "processLoop" function first calls the function to get the next frame (using the pointer stored in the "cameraModule" struct

[20]But not beyond the scope of "FireWire System Architecture" by Don Anderson, published by MindShare.
[21]You may need to set verbose mode to cause camera parameters to be listed to get the model name output.

"getFrame"), and then steps through the linked list of processModules calling the "processFrame" functions listed there. Demon2.c has registered it's "_processFrame" function in this list, and thus each time through the "processLoop" loop demon2 function "_processFrame" is called.

EVERY module loaded using the "module:" parameter can register a frame processing function. And every module loaded using the "module:" parameter can get access to incoming frames, the same way demon2 does.

7.1 "focus.so"

"focus.so" is an example of extending the function of HOTM by using your own loadable module. You can use the source ("focus.c") as a starting point for your own extensions. The important functions to include in your extension are listed now.

7.1.1 _init

This is the function that will be called when your process module is loaded. It needs to do any initialization that your task requires. It will be called once, immediately after the processModule structure is allocated for your process and then the library module is loaded.

For "focus.c", there are only three things to do:

- 1. Find the processModule table entry for "me", which will always be the last one.
- 2. Determine the value of the "verbose" flag for later processing.
- 3. Put entries into the processModule structure to point to local functions for processing a frame.

7.1.2 _processFrame

Because this function is called by a pointer and not name, you can actually name this anything you want. It's just convenient to name it based on what it does.

It accepts as input a pointer to your processModule structure and a pointer to the incoming frame. The frame is not the frame as returned by the camera frame collection call, it is a standardized raw byte array. (E.g., the frame from a Spinnaker camera has its own format and "extra stuff", but _processFrame doesn't get that data directly).

The example "focus.c" calculates the variance of a small box at the center of the image and prints the results.

7.2 Important Considerations

It is important to keep in mind that your processFrame function will be called once per frame, for every frame that is collected. (It will not see frames that are skipped.) Your function MUST complete its processing in less time than occurs between frames. E.g., at 2 fps you have less than 500ms – a lot less, since demon2 is also using some of that time. Anything that will require longer than 100ms to perform may be better done outside of HOTM. You might consider saving the raw data you need in your processFrame function and then postprocessing it later. [22]If you consume TOO much time you will start missing images, and missing images apply to ALL HOTM processing, not just your function. The Firewire frame management libraries are forgiving of intermittent delays in requesting frames, but GigE is not. If you miss a GigE frame, it is gone forever.

As a debugging option, you may wish to "gettimeofday" at the beginning and end of your processFrame function and store the difference in microseconds in the processModule "processMicros" variable. Setting the HOTM -v flag to 128 (0x80) will cause processLoop to output the processing time for your module for each loop.

[22]This is how demon2 gets around slow CPUs for creating image products. It can dump all the intermediate data (sum of I, sum of squares, etc) to a temporary file which is then postprocessed into the Argus image products.

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

8.1 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. catitapianca@gmail.catitapianca@gmail.comcom

As a second approximation[23], 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 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.[24]

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.

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

[24]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.

8.2 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[25].

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

8.3 GigE (Gigabit Ethernet) and the Solution To All Our Problems

Excuse me for a moment while I get back up off the floor.

GigE cameras have data limits, too. They are not isochronous, and thus there is no reserved amount of bandwidth for cameras, nor is there a well defined time when each camera will send its data.

There are two parameters that have proven to be important for GigE cameras using the Flycap library. They may be relevant for Spinnaker, but that has not yet been determined. The first is "frame size". This is also referred to as "jumbo frames", or more technically "MTU" (maximum transfer unit). For normal ethernet, this value is around 1500 bytes. That's the maximum size of each packet. Data that is larger than this size is fragmented and sent as multiple packets. Each packet requires addressing and routing, so splitting one large transfer up into a lot of small ones slows things down.

[25]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.

To mitigate this issue, some ethernet systems can handle frames of up to 9000 bytes. These are "jumbo". This is an important buying decision for gigabit switches and interfaces. I have yet to find an interface that cannot handle it. I have found gigabit switches that claim to handle jumbo frames but actually do not.

The second parameter that impacts throughput is the "interpacket interval". It is the amount of time the camera waits between sending frames for an image. The longer the interval, the slower the transfer.

For Flycap cameras, the two camera paramters are "cameraMTU" and "interPacketDelay". If you have a Flycap camera and you see consistent image problems, try setting the cameraMTU value to 1400.

The Spinnaker library appears not to have these issues. It has a "bandwidth" allocation setting which is currently being managed by the gigeSpin demon. It depends on the image size and the desired frame rate. It is set automatically whenever the frame rate is changed. NOTE: if you are using external triggering, the camera module does not know your intended frame rate, it knows only what it was told. MAKE SURE you set a frame rate (using setcamera, e.g.) that is higher than your desired trigger rate.

Index