Ultracam software design overview

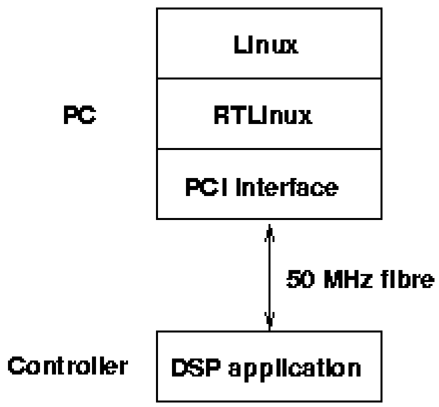
With previous (vxWorks- and VME-based) SDSU controller systems the high degree of coupling between controlling vxWorks code and the applications running on SDSU VME interface and timing boards, otherwise called the controller or *camera*, is unsatisfactory.

Addition of minor functionality at the controller end could require corresponding additions at several other of the software layers. The resulting control and data-acquisition software is often unwieldy and complicated.

For the Ultracam system we are required to abandon the established architecture, because of the movement of the San Diego group towards PCI bus as replacement for the VME interface between the controller and higher-level components. We envisage using standard PC hardware to interface the controller, via the SDSU PCI card, with an associated reduction in the hardware cost and increased freedom in the selection of software infrastructure.

1 The real-time and PCI interface layers

The Ultracam software design introduces the Linux operating system, well established and robust on standard PC hardware, with the kernel modifications required to utilise one of the real-time Linux varients: RTLinux or RTAI.



Analysis of the proposed camera system reveals that the requirement for real-time software performance is in fact minor, localised and specific. The primary requirement of the control and data-acquisition software is that it is *fast*. This requirement will be satisfied by adequate performance of the PC hardware and sufficient care by the software designers, not by using a real time operating system. The only *real-time* requirement is that we remove image data from memory on the timing and PCI cards reliably enough to allow the timely acquisition of subsequent images.

This fresh examination of the requirements of the camera system results in the following philosophy for the real-time and PCI layers: that they are an *interface* between the controller and higher level software (passing commands, responses and data), nothing more. The boundaries here are very well-defined: these interface layers are not required (in fact, they are not *allowed*) to configure or control the timing board. They must, however, be designed to efficiently receive and forward any messages passing to or from the timing board, on real-time timescales.

2 The linux camera control software

This decision to differentiate the communication and control requirements of the software serves also to localise the *control* software, in our case into the standard Linux layer. During analysis of the system, one of the first candidate objects to be identified was christened the “camera object”. This object has survived several iterations of the software design and has become an important part of the design philosophy. It has the following characteristics:

- Exactly one camera object is instantiated for each SDSU controller in the system and, therefore, for each associated PCI interface card. Ultracam will require only one camera object.

- Each camera object, when instantiated, creates its own real-time task (or set of tasks) with which it will communicate with its PCI interface card.

- There is no requirement for the camera object to support real-time functioning outside of the real-time *interface* requirements set out above.

- The camera object provides access to the set of commands understood by the camera. As detailed below, the set of commands being implemented for Ultracam is limited. Execution of all the behaviour required of the Ultracam camera will be achieved using only these commands.

- Aside from its role in controlling the camera, the camera object also serves, wherever possible, as an up-to-date *record* of the current state of the camera. The object therefore supports functions that allow it to report the status of the camera to a user, *which will be reported without additional communication with the camera*. This allows, for instance, the user to examine the status of the camera even when the camera is occupied reading out frames.

It will be noticed that the camera object is designed to appear, and behave, to the outside world as if it *were* the camera. This is intentional. It is hoped that this will alleviate the need, present in previous systems, for higher level software to understand the specific workings of the controller hardware or, indeed, the internals of the control software itself. Its role in facilitating the production of multi-camera systems should also become evident later in this document.

3 Implementing Ultracam functionality on the SDSU controller

In previous systems much of the complexity and, one might argue, unreliability of the development and delivered product arose from the need to support a large variety of operations on the timing board. This is typically achieved by writing a single, monolithic DSP code application, which can be downloaded onto the camera controller and does everything the user requires. The controlling software needs to understand all such functions within the DSP code.

A careful examination of the typical requirements of such systems (focussing on the Ultracam system) resulted in the following observations:

- that, whilst speed of execution is an issue in the design of such systems, its scope is limited only to those operations which generate either an image or a sequence of images

- for Ultracam, therefore, we require that the readout of an exposed frame, and the return of the system to a state in which it can repeat this operation for subsequent frames in the sequence, be as rapid as possible

- for all other operations, speed becomes a design consideration *only* when its absence would inconvenience the user. For example, whether it takes 1/10 second or ½ second to switch power to the CCD before a long sequence of frames is unlikely to be of any interest to the user, but whether it takes ½ second or 5 seconds to take a single image will matter to the user who is trying to configure the camera.

The design changes that result from these observations are as follows:

- that a distinction will be drawn between the *applications* that can be run on the controller and the *commands* that are supported by it.

- *commands* are implemented as early as possible in the project and are unchanged thereafter. They are not specific to the Ultracam system. DSP code for supporting these commands will be held in persistent memory on the timing board (ie. they are part of the boot code) and, as such, will be available at all times when the board is functioning properly.

- the board will support only five commands: read and write to memory, start and stop for the currently loaded application, and reset to the default state.

- *applications* provide "useful" functionality at the camera. A standard location in timing board memory is identified for all valid applications, and applications must be generated and compiled to work at this location. A set of applications must be written to generate the required Ultracam behaviour.

- commands do not send data to the controlling software, but they *reply* when invoked. Applications do not reply when run or stopped, but may send data during their execution.

- applications are loaded into timing board memory using the primitive "write memory" command. The current state of application parameters can be examined using the "read memory" command.

- when instructed to "run application", the timing board DSP will run whichever application is currently loaded at the correct location. The timing board must respond to the "stop application" command, in order that long-running applications may be safely terminated.

- *all* functionality in the camera is implemented using applications. Whilst it would be possible to, say, switch the CCD power on using a carefully timed sequence of "write memory" commands, this will not be supported by the higher level software. Instead, an application must be written to perform this operation. After this application is downloaded and run, the camera object must know to check power status on the controller. If the state has successfully been modified, the camera object will update its *record* of the camera power status. Thereafter, a user or higher level software may ask the camera object to report its record of the power status.

These design decisions considerably simplify the task of designing DSP code for Ultracam. If extra functionality is required of the camera controller, and this functionality is not time-critical, a new single-function application need only be designed and made available to the software system. If required, this new function can be sequenced with others by higher level software. A set of checks to verify that the operation was successful must be supplied, as must a set of conditions that must be met before the application may be run.

The creation of *commands* and *applications* also considerably simplifies the requirements of the camera object. The object needs no record of a large number of possible (and variable, during the development stage) commands, with associated reply and/or error messages. It now must handle only five such commands and know how to interpret the replies thus generated.

The camera object must also know how to download and run applications, using the commands it has available. It must also be able to perform the required checks before and after the application is executed, and will report errors that result from these checks. A failure to start or stop the execution of an application will be reported in the relevant command responses. A error that occurs in a running application can be recorded in the application's memory space, to be examined after the application is shut down cleanly. The object must be informed of the format and quantity of data output by the application, and be able to update its record of the camera status or recognise indications of an error condition, based on this data.

In short, the camera object has *static* record of the commands implemented on the controller (with the allowed responses) and a *dynamic* record of the application that is currently resident on the controller (with the data that will be output when the application is run). In this, it is valid representation of the state of the camera.

4 Running an application using the camera object

So far, we have identified several properties of an application that must be recorded and supplied to the camera object:

- the DSP code, with associated memory locations, that must be loaded into timing board memory
- the format of data generated when the application is run
- a set of conditions that must be in place before a user is allowed to run the application
- a set of conditions that indicate that an application completed successfully, with a human-readable error message for each violation of these conditions.

There are further properties that facilitate our reliable use of applications:

- a record of the parameters in use by, and available for use with, the application. These are a set of named memory locations which might correspond to, say, an exposure time, frame count or error status. Parameters can have default values which correspond to those values assigned *at compile time*.
- a list of parameter values which must differ from their default values, according to the requirements of the user. An example here is the exposure time parameter, which will certainly be set every time an application is run.
- a list of parameters that are *persistent* on the controller. In other words, those parameters that are defined by the boot code or camera hardware itself. An example is the CCD power status, which will be read from a latch, hardware mapped to a DSP memory location. These parameters will be stored “statically” by the camera object, using information supplied when the object is instantiated.
- a subset of the pre- and post-application checks require only that parameters are read from the controller (using the read memory command) and checked against a desired value. CCD power status is an example. The other allowed type of check involves verifying that user-supplied parameter values satisfy any criteria imposed on them by the application or camera.
- Any comments or additional information that the user wishes to supply and record for this particular execution of an application.

Using this information, the camera object can setup, run and verify the success of all applications that are written for the Ultracam system. The choice of format for the storage and transmission of this information to the camera object is XML, which is described further in this document.

5 Data Handling modules

Data resulting from the execution of an application (hereafter referred to as “application data”) falls broadly into two types:

- trivial data, resulting from “housekeeping” applications such as power on/off. Such data may be examined but will more than likely be discarded or not generated at all.
- image data. The user will require the data acquisition software to perfom several tasks with this data: return in a format suitable for display, for example, or store to disk.

The modular nature of the Ultracam software means that the developer may produce a set of data handling modules, each of which performs one task that the user requires on the data. For example, data handling modules could be written to perform the following tasks:

- stream all received application data to a disk, in such a format as to allow retrieval of individual images at a later time.

- retrieve the last dataset received, reconstructing images from each region-of-interest in a format suitable for display or examination
- correlate image data with timestamp data stored elsewhere in the software system

It is worth noting here that other data handling modules may function in the absence of the camera hardware. As an example, Ultracam will require a data handling module capable of retrieving archived image data from disk, and supplying each frame in the FITS format.

Application data from a particular execution will have a characteristic size, and will appear a fixed number of times – both of which are determined by application parameters. As an example, the parameters specifying limits for a region-of-interest (window) will change the size of an image dataset, whereas a frame counter may alter the number of images transmitted.

Each unit of such data (we'll call them "frames" for now) has a structure defined by the application, which must therefore be mirrored in the application details sent to the camera object. Following previous designs, two types of information are to be sent per frame: header information, reflecting the current state of the camera or application, and image data. In the Ultracam system, these may be output in any order that can be specified in the XML file for an application. In other words, we require a specification of the location and meaning of all header words, relative to the beginning of the dataset, and of how to reconstruct meaningful image data from the data sections. The camera object will examine any header words to update its record of the camera state and report any errors. Data Handling modules, on the other hand, will typically be interested in the image data.

6 Supply of application data to data handling modules

The real-time and PCI interface software layers will forward three types of message between Linux processes and the controller. The first two of these, timing board commands and the replies generated, are handled solely by the camera object. Access to the third type, application data, is required by the camera object and any data handling modules requested by the user.

As in some previous systems, application data will be streamed into a number of "data buffers", allocated and managed by the real-time layer. The PCI interface code is required to notify a real-time task that data has been sent from the timing board. The real-time layer responds by pointing the PCI interface to the buffer that should be used to store these data. Once these data have been fully placed within the buffer, they are available for use by the camera object and data handlers. Once all of these modules have examined the data, it is available for re-use by the real-time/PCI layers.

Sufficient buffers must be allocated to allow all the data handlers to access each dataset, and buffers must be large enough to hold at least one set of frame (application) data. In practice, they will be large enough to hold *exactly* one set of frame data. In order that the real-time, PCI-interface and camera object layers be application-independent, the amount and size of the data buffers must be configurable, specified in the application details generated when a new application is written. Disposal and re-allocation of data buffers will be performed by the rel-time layer, and will take place (if required) before every execution of an application.

This places an additional requirement upon the software design: that data handling modules must be able to find, and indicate a requirement to access, the data buffers generated for a particular application. The implication is that the camera object is first informed of application details, which results in generation of new data buffers by the real-time layer. After this, data handlers are given application details, which prompts them to "attach" to the required data buffers. When these steps are complete, the camera object is allowed to proceed with the execution of an application. We have identified a need for a *sequencer* layer to co-ordinate the operation of camera objects and data handlers which, it turns out, will be the same software that is serving these processes with application details in XML form.

7 Design of the real-time data buffers

The detailed design of this time-critical component has not yet been finalised. The requirements of the buffer management system, however, are known:

- that handing of data buffers to the PCI-interface, camera object and data handlers must be as rapid as possible.

- that data buffers are examined by data handlers in the same order that they were filled by the real-time/PCI layer. This implies that pointers to data buffers will be stored in a circular buffer.

- that the real-time/PCI layer gets access to free, or "empty", data buffers before any data handlers. Whilst this requirement is somewhat obvious, an additional, somewhat desirable characteristic is identified:

- that access to the data buffers can be *prioritised*, in that processes requiring to examine buffers at a particular priority are guaranteed that lower priority processes will pend on a buffer until these operations are complete.

In this model, the real-time/PCI layer requires the highest priority access, the data storage/archiving module uses a lower priority, whilst a data-visualisation module might have the lowest priority. Inclusion of this facility will allow for data handling modules that *manipulate* new data in the buffers, before other modules (that simply read the data) obtain access. It is proposed that the module correlating image data with timestamps be an example of such a data-manipulating module.

Once the real-time layer has allocated the buffers required to collect application data, we need a mechanism by which data handling modules can "attach" themselves to the buffer system at the correct priority.

8 The Ultracam "timestamping" requirement

The Ultracam system has a requirement to generate accurate timing information for each frame generated by the camera. The hardware designed to achieve this is detailed elsewhere, but can be summarised as follows:

- the timing board will be used to generate a hardware pulse at the start (or end) of each frame.

- this hardware line is used to interrupt the CPU via a serial interface board installed in the Ultracam control/data-acquisition PC.

- the resulting interrupt service routine (ISR) call will log the current state of the CPU system clock into a FIFO.

- the system clock is kept as close as possible to UTC using a GPS receiver and the network time protocol (NTP).

The ISR and timestamp FIFO will be set up by the timestamping data handling module, when this is instantiated. The module must then attach itself to the data handling buffers, *at a higher priority than other data handling modules but lower than the real-time system*, and await the first data set from the PCI interface. When the first data buffer is available with image data, there should already be the corresponding timestamp placed in the timestamp FIFO. The data handler will simply insert this timestamp into a (currently empty) location in the data buffer, and release the buffer to any other data handler modules.

Provided the data buffer management scheme can support such operation, this simple data handling module is able to correlate (with a minimum of CPU overhead) accurate timestamping information with every frame generated by the timing board.

9 Configuration of data handling modules

We identify two types of data handling modules

- those that are "essential" for fulfillment of the user's requirements when executing an application. These are modules that must receive, in the correct order, every data frame that is generated by the camera. File saving and timestamping are examples.

- those that are "non-essential" in fulfilling these requirements. Such modules need examine frame data periodically but are not, however, required to examine every dataset. These modules are designed to provide feedback to the user during execution of an application. A module that accepts image data and re-formats it suitable for display in a image tool is an example.

Essential modules must be configured by the sequencer before the camera object is instructed to start the application. However, as they require access to data stored in the real-time buffer system, they must be configured *after* the real-time buffers have been set up for a particular execution of an application.

The implication of this is that we must be able to configure the camera object, and hence the real-time layer, independently of starting the application. Indeed, this is consistent with the earlier assertion that the primary functions of the camera object are to implement the *commands* of the controller. One of the commands, to write to memory, will be used to download an application to the controller and modify any parameters. Hence, this serves to configure the controller and it is at this time that the real-time layer and camera object itself are configured. A separate command, that used to run an application, serves to start the application.

We envisage the same functionality in each of the data handling modules. Each will be configured for a particular application, and must be "started" before the camera object is started. At this point, each of the modules is awaiting the first full data buffer for examination.

In addition, we have stated that it is possible to "stop" the camera object – causing the currently running application to be aborted cleanly. We add this functionality to data handling modules also, but anticipate using this in conjunction with a further signals from the real-time layer. In order to cleanly stop the operation of the (already data-driven) data handling modules without the potential for data being ignored by a module, we must not only use the sequencer to explicitly stop a module after the camera has completed an execution. Instead, we anticipate adding an extra word to each data buffer, reserved to hold a flag for the data handlers. The flag is set by the camera object upon receipt, from the controller, of the reponse to a "stop" command, and indicates to both the real-time system and data handlers that a particular buffer contains the final data frame. Essential data handlers can be instructed to "stop", but will wait until this last data buffer has been examined before sending a response to this command.

Non-essential modules, however, behave in a entirely different manner. These modules may be started whilst the camera is currently busy running an application, and may be stopped before the execution is complete. Modules of this type will gain access to and examine, once or periodically, the latest full data buffer. These modules configure by examining the static data structures that describe the current data buffer setup. They "start" by attaching to the latest available full frame.

Non-essential modules of this type do not require their operations to be sequenced with those of other modules. Instructions from the user to configure, start or stop these modules must be handled asynchronously by the sequencer.

10 Data handling for the data reduction system

This module is a non-essential data handler that is unusual in that it doesn't require an operating camera object in order to function. It must gain access to every stored frame, but may be started and stopped at any time. All access to image data by the data reduction system is using data stored on Ultracam's RAID disks (ie. data that has already passed through the data saving module). The data storage format is likely to be RDF, but only the file saving/retrieving modules need to know how to manipulate these data.

The data retrieval module shall access a stored data record, formatting and returning the image data in whatever format is required by the data handling software. The default format is FITS but, since the data handling system is modular, it is fully acceptable to write further data handlers that generate other image formats.

11 Interface to the camera object and data handling modules

As stated above, XML is chosen as the storage format for details about Ultracam applications that are relevant to the control and data-acquisition software. The list of relevant details is given above.

It is also decided to use XML as the transmission format for all messages that pass between the camera object (or data handlers), and higher level software. The messages will be passed over HTTP. No messages are passed between the camera object and data handler modules. Rather, the configuration, starting and stopping of all such components is sequenced by higher-level software. Co-ordination of *data access* between these components will be inherent in the data buffer management scheme.

Furthermore, it is decided that the camera object and all data handlers be passed *the same format of message* when an application is to be started. Each component must extract those application details required to execute its function, and may discard irrelevant information.

The intended advantages of these decisions are as follows:

- integration of new modules into the system is as simple as possible, involving only the generation of a new module that meets the interface requirements. In addition, it will be necessary to generate a new interface component (see interface requirements, below). Neither the other data handling modules, nor the camera object, need know of the existence of the new module.

- generation of the new module is simplified, as we have a standard published interface to all modules. Generation of the interface component must, of course, be customised to the new module.

- the sequencer software can blindly configure the system according to the requirements of the user, and need not know the specifics of any one data handler. It will interact with all such components in an identical fashion

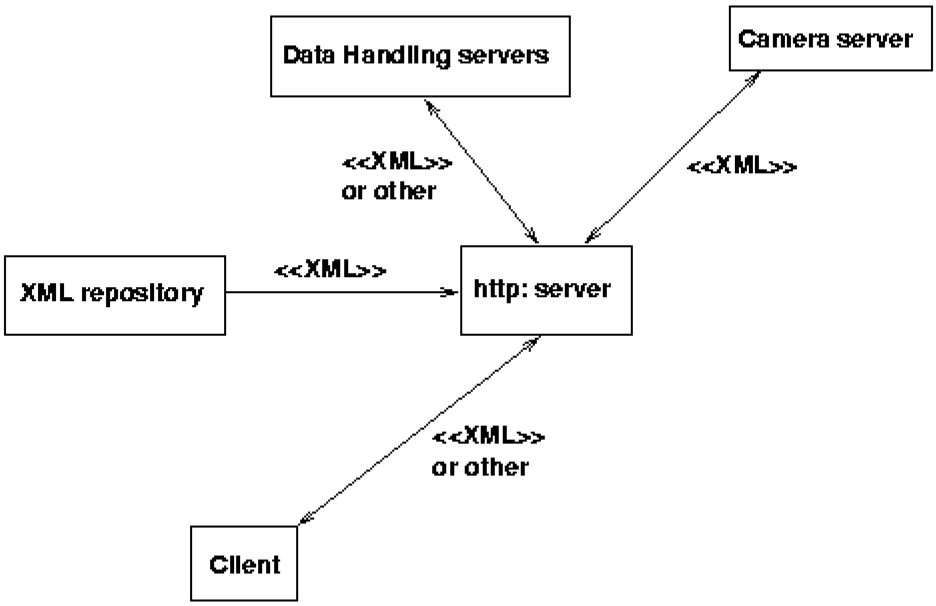
- the system is easily configurable by the user. If a data handler is required to archive image data, then it will be started by the sequencer. If the user is not saving but only examining data, this handler is either not started or will be inactive.

- the system is easily configurable by the developer. When developing a new instrument, the core infrastructure (camera object, data handler interface, sequencer, user interface specification, application specifications) are unchanged. New applications are written to implement the required camera functionality, with new data handlers and interfaces as required.

- the architecture is scaleable (to multiple controllers or interfaces, for instance).

- the use of HTTP for internal message passing enables the software to be distributed for larger camera systems.

The top-level interfaces can be summarised as follows:



The HTTP server on this schematic serves as a placeholder for the instrument sequencer software and XML server. It provides the interface to all external "clients", which might be command line interfaces, Java-based GUIs, or the data reduction software. An HTML server, to generate a web-browser based user interface, would also be termed "client" in this model.

As stated above, the instrument sequencer software is responsible for starting and configuring data handling modules, based upon the requirements of the user, for each execution of an application. Any feedback of the camera status, visual or otherwise, is configured by the HTTP server also.

An XML server is used by the sequencer layer to interrogate the XML repository. The sequencer passes XML fragments to relevant interface components for modification (by the user), verifying the returned fragments before passing them on to the data handling modules and camera server.

12 External interfaces

"Clients" in the schematic above represent all components that interface to the Ultracam system. This will include any command line interfaces, java-based graphical interfaces, possibly an HTML server for browser-based interaction with the system, and the data reduction system interface.

Final details of the client interface are not decided. However, a requirement of all such interfaces will be that they are HTTP clients. They are not required to be HTTP servers. These clients must post an HTTP request that contains an XML-format command to the sequencer. Such commands will instruct the sequencer to configure either the camera object or a data handler, or to start an application running.

Replies from the sequencer will take the form of an XML document.

13 Usage of the XML repository

The XML repository serves several purposes in the Ultracam system. The design is not dependent upon any specific XML technology, although some appear more appropriate than others at this time. The XML repository will be either a set of document files coupled to an XML parser and file management software, or an XML database with its associated interface.

A list of the intended file, or record, types is as follows:

- details of the PCI interface installed into the system. This record would contain, for instance, the PCI bus address, reset address and boot code version.

- details of the timing board hardware and boot DSP code. This record will contain information such as the parameters (with hardware addresses) which are inherent to the controller but independent of any application, as well as the list of commands accepted by the timing board. These first two documents will be handed to the camera object during instantiation.

- application details. As given above, information about the DSP code, parameters and data output format are contained in these documents. They are collected when required by the camera object and data handlers.

- definitions of the camera object and data handling modules, including executables, command format expected and response format generated. These records will also specify information to the sequencer about the relative scheduling priorities for all essential modules. They will be used by the sequencer to start up these modules at initialisation or when requested by the user.

- template documents for passing to the camera object and data handlers, specifiying the user's requirements of an application. These must be modified before use and specify, for instance, the application to be executed and parameters to be changed before execution.

14 Requirements of the instrument sequencer

Many of the requirements of the sequencer are implied above, during the discussion about essential and non-essential data handlers. The sequencer must:

- start up a new camera object when the user, through whichever interface, requires to use the system

- start up a data handling module for the data reduction or timestamping systems, when instructed to do so.

- start up and connect new non-essential data handlers whenever required, whether or not an application is currently being executed.

- send configuration messages to the data handlers whenever parameters are changed by the user.

- configure and start, in the correct order, the camera object and data handlers with the user supplied application details upon receipt of a command to start the application

- verify (validate) each XML message from a client.

- forward responses from the camera object and data handlers and hand them back to the originating client. Such responses will contain, for instance, the URIs that may be used by clients to obtain further information from the data handlers