EDOC014

EUROGAM PROJECT

NSF DATA ACQUISITION SYSTEM

Event Builder and Sorter Control

Edition 1.0
June 1990

Nuclear Structure Software Systems Group
Department of Physics
University of Liverpool

The aim of this document is to describe the software design of the Event Builder and Sorter subsystems. The two subsystems are discussed together as they have similar requirements on an abstract level, and also some common components. A common design strategy should result in fewer distinct software components, more uniformity across the system as a whole and hence result in easier maintainability. The common components, at least initially, will be the crate controllers (MVME147), the processing cpus (MVME165) and the interface controllers connecting the two subsystems.

The primary design goal for each subsystem is to provide a structure that is efficient in terms of data flow, modular (allowing component change with the minimum of fuss) and capable of failing gracefully. The last criterion allows, for example, one particular processor to crash and be restarted by the crate controller without affecting the others.

The requirement for submodule change is taken care of in several ways. Symmetry between the input and output interfaces enables the same code to function in the output interface of the Event Builder and the input interface of the Sorter. The form of the I/O interfaces needs to be defined in terms of the minimum requirements to be supported (e.g. each interface has its own controller cpu with the correct connections to reduce to one the number of data passes over VME or VSB bus). Code specific to a particular interface should only be coded in the controller cpu associated with that interface.

Crate-wide Controls

The overall control of crate activities will be the responsibility of the crate controller cpu. This cpu will run (amongst other tasks) a crate controller application control process (CCACP) to coordinate the activities of the interfaces and processors. This process has several responsibilities. It has to ensure that all components are activated in a safe manner on reset. It will provide a watchdog function to make sure all components are active. It will pass commands to the components as necessary. Thus the CCACP will probably have several subtasks to ease the burden of coping with specific components.

The CCACP will auto-configure for the set of boards in the crate. The current offline sort engine already uses this scheme for both processor and memory boards. During initialisation the CCACP will find all the components present at known VME addresses, and take that as the correct working set. This information will be visible for users to inspect. If a board failed and no spare was immediately available, then the system would continue to function without software change, with reduced capabilities.

All components will be regularly checked by a watchdog system in the CCACP to try and diagnose failures. This will be mainly for the processors. The CC will keep a local copy of the downloaded code and data to enable failed cpus to be restarted. This will also allow power-up reset to result in a working crate.

During reset initialisation the CCACP will check that all components are active. Then relevant information will be passed to defined data regions in each processor. This information will enable the various processors in the crate to communicate as necessary. Once all the components in a crate have been initialised then messages will be sent to activate each component.

Intra-crate communications

We require efficient message passing communication between all cpus within a crate. Ideally it would be good if the same method applied to all cpus. However, this is not very easy as there is no VME standard mechanism for message passing. We have to invent one (or two!).

The whole system has to be studied as to message passing requirements. For example, can we assume that there will be only one message outstanding between any two processors. In the case of the input interface controller and a processing cpu, will there only be one data block message and no control messages simultaneously ? Do we need a priority system for the CCACP to interrupt data transfer activity ?

A further investigation revolves around whether interrupts are necessary to inform the recipient of a message arrival. The usual form this would take would be a mailbox interrupt. However, the cpu boards under consideration do not support a "true" mailbox interrupt (i.e. writing data to a location in dual-ported memory). A more generally available capability involves writing to a globally visible control register on the remote board in order to generate an interrupt. In this case the data has to be written separately to agreed locations in dual-ported memory. It should be noted that boards that support the "true" mailbox interrupt can also be used in this manner.

I/O interfaces

Each interface subsystem will be under the control of one cpu only. The input interface to the Event Builder will be the HSM8170 controlled by a FIC8230. The output interface of the Event Builder and the input interface of the Sorter will be fibre optic in nature, the particular boards as yet undecided. The output interface of the Sorter is the connection to the tape server. At Daresbury this will be a parallel interface to the GEC, the details of which are not yet fixed.

All of the interface controller cpus will be coded in the same style. The code will divide into three sections. There will be a cpu board specific initialisation section. There will be an interface specific control section. Finally, there will be a section that communicates with the CCACP for control and with the processing cpus for data transfers.

The design of the Event Builder and the VME Sorter necessitate that all interfaces except the output interface of the Event Builder be connected to VSB. The input interface to the Event Builder requires data transfer over VSBbus to optimise transfer rates. Both interfaces in the Sorter require transfers over VSBbus to free VMEbus for spectrum access.

Whilst VSB will be necessary for data transfer reasons, it is expected that all control communication will be via VMEbus.

Input Interface - Processor transfers

The FIC8230 (a contender for both input interfaces) will transfer data blocks to the processors by DMA. Making the interface controller as opposed to the processors responsible for the data transfers will be the most efficient way. This leaves the processors free to do only what they are supposed to do - process data.

Therefore, to be efficient, the interface controller must be allowed to transfer a data block to a processor whilst it is already processing a previous block. This implies that each processor must allocate at least two input data buffers. In practice, two data buffers will be sufficient.

Upon activation by the CCACP, each processor will send a message to the input interface controller informing it of the address of a buffer in its own dual-ported memory space. The interface controller will process these messages in rotation. Upon completion of a transfer it will send a message back to the processing cpu informing it of the availability of the data block. Thence, whenever a processor has completed processing of a data block, a message will be sent to the input interface controller with the address of the free buffer.

Processor - Output Interface transfers

In the same way as for the input, the output interface controllers will be responsible for transferring data blocks from the processors. This function has to be performed as efficiently as reasonably possible due to the high data rates involved, and so some form of DMA transfer will be useful.

Whenever one of the processors output buffers becomes full it will send a message to the output interface controller asking for the block to be transferred. Therefore a two buffer system will be needed as for input. The processor will switch to using its other buffer immediately (as long as it is free) and continue processing.

Message passing scheme

The aim is to provide a reasonably universal and simple scheme. Each cpu will have a block of memory at a known fixed address known as a message buffer for each message type it is involved with. At initialisation, the CCACP will allocate these blocks as required. The message types will be control messages and data transfer messages. This differentiation is purely because data transfers occur between interfaces and processors, and that control messages pass between these and the CCACP.

A message passing will consist of the following sequence of events. The message sender will write to the corresponding message buffer in the recipient cpus dual-ported memory space. The write will consist of a flag word (16bits) and a data word (32bits). This will then be followed by a write to an address that will cause a local interrupt to the recipient cpu. The local interrupt will be either a mailbox, signal, fifo or abort interrupt depending on cpu board type. The interrupted cpu will then execute the message request and reply with an acknowledgement message to the sender. The reply will consist of flag and data words in the same way. followed by writing to a suitable interrupt address. The receiver of the message will have a status word (16bits) which will be used to provide an indication of the state of execution of the request if necessary.

The control messages will only pass between the CCACP and the other cpus in the crate. There will not be any control messages passing between the other cpus. Examples of control messages would be reset, enquire status, halt processing, start processing ... In the case of control messages the flag word will just be set to indicate the presence of a message. The data word will be an operation code. The recipient cpu will respond to the request and return to its corresponding message buffer in the CCACP an acknowledgement in the data word.

Data transfer messages will be organised in the following way. Each processor will have two data transfer buffers for input and (at least) two data transfer buffers for output. For input, a particular processor will know the address of its two message buffers in the input interface controllers dual-ported memory. Similarly, the controller will know the address of the corresponding message buffers in the processor in question. The processor will write the address of one of its data buffers into the data word of the message buffer. The flag word will indicate message presence. Upon starting to execute the queued transfer, the controller will set its status word appropriately. Upon finishing the transfer, a message will be sent to the processor indicating transfer complete.

Interrupts to signal message arrival are not strictly necessary for data transfers between the processors and interface controllers. The above design is an unordered but controlled queue. Whenever an interface controller is ready for a transfer it will search its (small) message buffer list for an available buffer. Whenever a processor is ready for the next data block its will inspect its message buffer. This would work because the cpus have no other tasks to accomplish and can spend time polling their local message buffer(s).

The CCACP is a more complicated communication problem. It will form part of a multi-tasking system with several tasks to accomplish. In this case, the control messages will require interrupt usage. The CCACP will require immediate response to some control message types sent to the processors, and may not be prepared to wait until the next end of block for the processor to check its control message buffer.

Whether its sensible to use interrupts in one case and not in the other is open to question.

Processor code organisation

The Event Builder and Sorter processors are very similar in terms of code organisation. Both require data block input and output with a single task processing the data. External control is required for both and will be supplied by the CCACP using the same commands. There will be extra application specific commands which will not apply to both system, but there is a basic common set to allow code downloading etc...

Since the same cpu board type (MVME165) will be used in both systems (at least initially) it seems sensible to provide a single solution for both systems. The code existing on each of the processors will consist of two parts. There will be a small bootstrap section in PROM. This will contain cpu board specific initialisation code, and a section to process commands from the CCACP. It will provide a board independent platform to support a single code module.

Each system will have code downloaded into the processors under the control of the CCACP. The code will be produced by different means, but there is no reason to make the underlying "system" different. The current offline sort engine has a minimal bootstrap code in PROM. The same idea will be used and enhanced for our purposes. The code in PROM provides cpu board specific initialisations and supports a small number of simple commands (reset, go, halt, etc ...).

A typical sequence of commands involves the CCACP sending a halt command and then writing the new downloaded code and data into position, followed by a go command. The local cpu needs to know the start address of the downloaded code.

There will be one code and variable data module downloaded to a defined address in dual-ported memory. Other data transfers will be necessary in order to change data used by the applications.

For the Sorter the basic structure already exists. Some changes will be required to allow for extra online requirements. These mainly relate to data layout to allow for parameter changes "on the fly".

For the Event Builder the code will be generated on the UNIX workstations using the GNU C compiler which has been configured to function as a cross-compiler running on the SPARCstation and producing 68k series code. This has the advantage that the compiler can be ported to other workstations easily in the future since the source is available. More importantly it enables Event Builder code to be tested and debugged entirely on the UNIX workstation using standard debugging tools. Once the code is adequately tested it will be recompiled and linked ready for downloading. This technique of using a test harness separates the processing code from the control and data transfer code. The latter can be seen to be stable and relatively simple consisting of only three or four C functions and requires debugging only once. The part of the system that requires more regular debugging is separated out into a good test environment. Examples of these functions would be "get_data_block", "send_data_block" and "process_command".

Event Builder code production

The main requirement will be to process blocks of input data as efficiently as possible, and to output re-constructed data blocks. Since the code may be relatively complicated, it is necessary to use a high level language. The most appropriate language to use is C. This is because it is the most widely supported language on the type of computing platforms we will be using.

The requirements of code running on an Event Builder cpu are quite limited. There is no point in using general I/O in a multi-processor environment. Many of the library functions commonly supplied with computing systems would not be used. The only appropriate I/O would be data block transfers and a few control commands. If functions were written to cope with these requirements, then the code could be debugged externally. The functions would look something like ...

```
input(bufptr);
output(bufptr);
send_command(id);
```

The current sort engine has a similar communication method. A foundation layer of software exists in each cpu, and contains the code to execute these functions. Equivalent functions could be written on the SUN to allow testing of the basic processing code. These functions would read data from a file, write to a file and communicate commands with the screen.

To this end it is proposed to use the GNU C compiler to generate the processing code. When optimised, the resultant code is acknowledged to be very efficient. The GNU C compiler is readily available to use on most UNIX machines. It is a sophisticated product with several attractive features. It has builtin the ability to use inline macro expansions. We can define our own set of macros where necessary. Assembler statements can be included easily and use C variables. We have the source, so porting to other machines should be possible.

Work has already taken place using the SUN SPARCstation to re-configure the GNU C compiler and linker to function as a 68k cross-compiler to use on a "raw" cpu board. This consists of the basic C language together with a limited set of functions that would be required in such an application. The functions include the maths library, string manipulations, ctype, malloc ...

The three interface functions described above could be provided as inline expandable macros producing, for example ...

```
input:
```

```
jsr [$400]
move.l +(sp),bufptr
```

The foundation layer of code supporting this function would have its entry address placed at a fixed address (e.g. $400).

Alternatively the functions could be provided in GNU C in the form of a library to be linked together with the users program. This way reduces the size of the foundation code layer.

The equivalent SUN code would be a C function. Instead of handling VME/VSB bus transfers, normal file I/O would be used on the SUN for testing purposes. This allows the standard SUN debugging tools to be used for program development.

This technique requires no operating system or kernel on the processing cpus, only on the crate controller.