The PRISM 2.2 Real Time Signal Processing System

Kenneth Stanwood & Albert Bourdon
ORINCON Corporation
9363 Towne Centre Drive
San Diego, California 92121

Abstract
The PRISM 2.2 system is a general purpose real time signal processing system designed for very easy development and modification of signal processing applications. Both the hardware and software are flexible and easily modified to support changing project requirements. Configuration files specify the attributes, such as VME and VSB addresses, hardware available, and intended usage of the available hardware. Use of additional hardware requires only an update of the configuration file. Signal processing block diagrams are easily converted to the PRISM script language which maps the processing to the hardware specified in the configuration file. The script language specifies "tools" to be executed, the order of their execution, and the flow of data between tools. The tools are signal processing primitives such as FFT, normalization, and neural network feed-forward. Changes to processing usually require only changes to the script. Code changes are not necessary unless the project requires functionality not provided by the current tool set. To handle this case, tools have a standard interface to the software backplane, so it is extremely easy to add a new tool to PRISM once the tool's algorithm is coded. Standard interfaces also exist for getting input data into the system and for getting results and display data out of the system, making PRISM an excellent core for most real time signal processing applications.

1. Overview
PRISM started with build 1.0 which was developed at ORINCON under the DANTES project. PRISM 1.0 consisted of commercial off-the-shelf (COTS) hardware with application specific signal processing software. Later, the ASIPS program started, with the intention of starting with PRISM 1.0 and "slightly" modifying the software to meet the application specific requirements of the project. It was soon realized that the two projects were too dissimilar for the "slight" modification approach to be feasible. In fact, the next projected build of DANTES was enough different from the first to make the task of modifying PRISM 1.0 to meet the needs appear formidable. Out of these realizations was spawned the idea of a general purpose signal processing system that would support the needs of many signal processing projects, while maintaining real time performance by utilizing a PRISM 1.0 hardware and system architecture.

The ASIPS program developed PRISM 2.0 which provides a script language, supported by hardware configuration files, which allows the same software to perform an extremely diverse variety of signal processing on a variety of hardware configurations. After ASIPS had proven the utility of PRISM 2.0, the DANTES project converted to the new system and further expanded its capabilities into PRISM build 2.2 which fully supports the data rate and processing requirements for DANTES build 2.0.

2. System Architecture
The PRISM system is data driven, distributed, and modular. The hardware consists of one or more VME chassis with COTS boards. Inter-chassis communication is done via ethernet and/or multi-drop reflective memory. Inter-board communication is accomplished via VME bus, VSB bus, Sbus, IB bus, and/or Ethernet. The operating systems are UNIX-based or POSIX compliant. The PRISM software is modular to provide the flexibility to add processors, memory and/or I/O boards or functions into the system with minimal impact. Just as the VME bus allows conforming boards to be plugged into the chassis, the PRISM environment allows conforming software modules to be plugged into a "software backplane" for incorporation into the system.

Since the system is data driven, the handling of data flow through the system is a key component of the PRISM architecture. The flow can be divided into three areas; data input which involves interfacing to the source(s) and formatting the data for processing, internal data transfer between tools and between processors, and data output to display, recorders, and/or other systems. Figure 2-1 illustrates the overall data flow for the PRISM system.

2.1. Raw Data Input
One or more raw data input buffers provide the common input interface into the PRISM system. The number, type and size of each raw data input buffer is determined by parameters in a configuration file which is read in and processed at system initialization. Currently data may be input into PRISM using a variety of methods: A/D boards, high-speed parallel DMA boards, reflective memory, recorded files from disk or tape, ethernet, or FDDI.

1058-6939/92 $03.00 © 1992 IEEE
The Real Time Controller residing on either the SPARC or Hewikon board interfaces to the input device(s) and services its (their) interrupts, if any are used. Input data is reordered and/or de-multiplexed into individual channels (beams) by the RTC prior to placement into the Raw Data Input buffers.

Each raw data input buffer consists of two components; a circular data queue (channel buffer) and a control record (channel header). The size of the basic data element for the queue (e.g., 1 byte, 2 bytes, etc.) and the number of data elements which can be held in the queue are specified in the configuration file. In addition, the configuration file also specifies the board on which each circular queue is to be located. The control record for each circular queue consists of both static and dynamic data. The static data contains the VME and VSB addresses of the circular queue, the size in bytes of the circular queue, the size in bytes of the data element in the circular queue, and a flag word which indicates the processor elements which are to read data from this circular queue. The two dynamic data items are the write index which points to the last data element written into this circular queue and a wrap count which indicates how many times this circular queue has been filled.

A set of control records for all of the raw data input buffers is initialized on the board designated by the configuration file as the main DRAM. Subsets of the control records are provided to boards where the raw data input buffers are located and to processing elements which write into (producers) or read from (consumers) the raw data input buffers. Only a single producer writes into a given circular queue and its associated control records. In the current version of PRISM, each array processor has a complete set of the control records in its local memory. The producer(s) uses the control record flag word to determine which copies for the control record are to be updated. This approach minimizes the array processor overhead and also minimizes bus traffic for consumer polling of the control record for data input.

A set of library functions have been implemented in the RTC for use by producers. The script processor determines which consumers are using each raw data input buffer and sets the appropriate bits in the corresponding control record flag word.

Each consumer reads a control record to determine if enough data elements have been written into the associated circular queue since the last read of data to meet the number of data elements (scan size) required by the current script for input to the tools. This feature allows for multiple consumers to input data at varying rates and scan sizes independent of each other and the producer. Using the write index and wrap count, each consumer is responsible for determining if data has been lost due to the producer overwriting a portion of circular queue prior to it being read by this consumer. A get_data tool which performs all of the above functions required to input data from raw data input buffers has been implemented for use by the script builder.

2.2. Internal Data Flow

Internal data flow is partitioned into two types. The first type is data flow between tools within a script and the second type is data flow between scripts. Data flow between tools within a script is accomplished through the use of each tool's output buffer. Each tool has a set of parameters identified in the script and two of these parameters are the number of output buffers needed and the size of the output buffers. The script also specifies each tool's inputs by the associated tool number(s). When the script is processed at system startup, PRISM allocates local memory space for each tool's output buffer(s) and then computes the number of inputs and a set of input pointers for each tool. The result is the each tool has a set of input pointers which actually point to the output buffers of the tools which the user specified as inputs.
Data flow between scripts is accomplished by specifying a dump_data tool in the producer script and a get_data tool at the beginning of the consumer script. This is the same get_data tool as described for the raw data input buffers. Which dump_data tool outputs a get_data tool reads is determined by the dump_data tool numbers inserted in the script’s list of input channels. The input channel list for a script may contain both raw data input channel numbers and dump_data tool numbers. Each instantiation of the dump_data tool takes the output from another tool within the same script and writes the data to corresponding pseudo-channel buffers. When the scripts are processed at system startup, PRISM generates pseudo-channels based on the parameters for each instance of the dump_data tool.

Like the raw data input buffers, each pseudo-channel consists of two components; a circular data queue and a control record. The size of the basic data element for the queue and the number of data elements which can be held in the queue are specified by the output buffer-size and numbers-of-scans parameter in the associated dump_data tool’s parameters. The dump_data tool’s board parameter specifies the board on which each circular queue is to be located. The pseudo-channel control record for each circular queue consists of both static and dynamic data which is identical to the structure and layout to a raw data input control record. A set of control records for all of the pseudo-channels is initialized on the board designated by the configuration file as the main DRAM. Subsets of the control records are provided to boards where the pseudo-channels are located and to processing elements which write into or read from the pseudo-channels. Only a single dump_data tool writes into a given circular queue and its associated control records. The dump_data tool uses the control record flag word to determine which copies of the control record are to be updated. The script processor determines which consumers are using each pseudo-channel when processing the input channel list for each script and sets the appropriate bits in the corresponding pseudo-channel control record flag word.

Each consumer determines pseudo-channel data availability, inputs pseudo-channel data, and determines if data has been lost in a pseudo-channel in the same manner as for a raw data input. Multiple consumers may input pseudo-channel data at varying rates and scan sizes independent of each other and the producer. The get_data tool performs all of the functions required to input data from pseudo-channels.

2.3. Data Output

Data may be output from PRISM using a variety of methods: display buffers, pseudo-channels, and special purpose tools. These allow data to be transferred from the embedded array processors to a more general purpose host that has either need of the information or the ability to distribute it further. Display buffers, despite their name, are the preferred method of data output. Originally designed to support real time displays, hence the name, display buffers are generic circular queues matching a data type to a data source according to attributes specified in the configuration file. The queues are constructed on a DRAM board in the chassis, providing access to most processors in the chassis. Each queue has an associated two word header containing the write status (i.e. wrap count and next write slot) for the queue. Consumers of the data maintain a corresponding read status. Consumers poll the write status, comparing it to their read status to determine data availability, quantity, and whether any data has been lost. It is up to the consumers to keep up with the data. The producer has no knowledge of the number or identity of consumers (if any). A data structure describing the location and attributes (# of slots, slot size, etc.) of the display buffers is stored in DRAM, referenced by the DRAM header, for use by the producers and consumers. A library of utility routines is available for use by consumer code.

Pseudo-channels (described in the previous section) may also be used to output data. As with display buffers, a description of the location and attributes of pseudo-channels is referenced by the DRAM header. Consumers may poll the header of a pseudo-channel as they would the header of a display buffer or may be interrupted when data is available. Pseudo-channels, however, are normally not used this way. The DRAM copy of the pseudo channel headers are, for efficiency, normally not updated, being used only at initialization to setup internal copies on the array processors. Also, utility libraries are not currently available to help consumer code use pseudo-channels.

In special cases, an array processor may be tasked with a writing data to another board in the chassis in a special format, and then notifying the consumer via interrupt. This method is sometimes chosen as a quick way to integrate legacy consumer code from another system or when the consuming processor cannot see the address space of a DRAM board over either the VME or VSB busses.

In all cases, the initial consumer of the data is another computer in the same chassis. That consumer may process the data or send it on via any method it has available (e.g., ethernet, rs-232, etc.). In the future IO daughter boards will be attached directly to the array processors, allowing direct transfer of the data outside the chassis.

3. Hardware Architecture

PRISM is a distributed architecture which utilizes a set of 6U/9U VME boards as the processing and control elements. The SPARC-1E is used has the host for the i860 processors and to perform initialization and script processing. Heurikon HK80/V960E boards are used for IO and archiving functions. Micro Memory ual ported memory boards hold the raw input data, selected pseudo-channels, and display output buffers. Mercury MC860-
VB and MC860-VS boards provide the processing elements for execution of the scripts. Table 4-1 lists the IO and graphics boards currently supported by PRISM. A minimum system consists a VME chassis with a VSB overlay, SPARC-1E with 16MB and 1GB disk, MM-63263/64, HK80/V960-8MB, MC860-VB-8, and an IO board or peripheral such as an 8MM tape drive. PRISM can output the processed data to disk, tape, graphics workstation, graphics frame buffer, and/or another computer system.

<table>
<thead>
<tr>
<th>Board</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Megatek Frame Buffer</td>
<td>X-CELLARATOR/6U</td>
</tr>
<tr>
<td>SUN SBus Frame Buffer</td>
<td>4E60-GX</td>
</tr>
<tr>
<td>Virga SBus Frame Buffer</td>
<td>VS-12</td>
</tr>
<tr>
<td>Virga A/D &amp; D/A</td>
<td>MML-210</td>
</tr>
<tr>
<td>VMIC Reflective Memory</td>
<td>5550-210</td>
</tr>
<tr>
<td>VMIC Parallel IO</td>
<td>485-DMAL</td>
</tr>
<tr>
<td>BIT3 Parallel IO</td>
<td>413</td>
</tr>
<tr>
<td>Bancom IFRG-B Decoder</td>
<td>PC09V</td>
</tr>
</tbody>
</table>

Table 3-1. Graphics and IO boards

4. Software Architecture

The PRISM software architecture allows specific signal processing applications to be developed in much the same manner an engineer would design the signal processing. First, the engineer lays out the block diagram for his processing. When satisfied, the block diagram may be expanded into a block diagram specific to the tools available. At this stage, a script is written which determines which processing resides on which processors. This script is "executed" by the prism system to carry out the signal processing needs of a project. Any changes to the processing may be made at any of the previously mentioned levels of abstraction and filtered down to the script. The following paragraphs describe the software architecture that allows this smooth conceptual flow from signal processing concepts to a real-time system.

4.1. Software Backplane

The major software architecture goals for PRISM were: (1) Use open environment software as much as possible. Application code was to be written in C or C++ using standard signal processing libraries, the operating systems were to be UNIX or an extension of UNIX (e.g. VxWorks or SUN OS), graphics in X windows (via OpenWindows), (2) Partition functional requirements into modules with clearly defined interfaces and map these modules onto the appropriate hardware processors. (3) Maintain a software design strategy to be upward compatible with future builds; no need to redesign or rebuild. (4) Develop software that must be reconfigurable for different data sources, different numbers of channels, and changes in algorithms. This requires development of a software architecture which provides utilities and procedures for data transfer and/or signaling between modules. These utilities must be independent from the applications using them.

To support these goals, a toolset approach was used that allowed functions (tools) to be independently developed and tested. In turn, these tools would communicate with each other via a Software Backplane. The Software Backplane defines a uniform method for initializing, invoking, controlling and combining multiple tools to perform signal processing tasks.

4.2. Tools

To support the design goals of PRISM, tools are modular building blocks which may be pieced together to implement the desired signal processing chain. Tools fall into two main categories: signal processing functions (e.g., FFT, hamming window) and support tools which provide functions such as IO, data formatting, and data synchronization. New tools are added by projects as the need arises, and projects may easily have tailored sets, adding project specific tools while leaving out tools the project will not need.

To aid in adding new tools, tools are handled in an object-oriented manner. Tools have an execution method, an optional initialization method, and soon will have an optional reset method. The standard calling interface for tools is shown in Figure 4-1. All three methods use the same calling interface. The actual code for a specific tool may accept the in_ptr, out_ptr, wk_space, and params arguments as types more closely coupled with the tool's function. The control mechanism simply treats these as pointers to buffers of specified sizes.

![Figure 4-1. Tool Interface](image)

Since the processing of n channels may be spread across m processors, the n_beams parameter specifies how many channels the tool will be processing. All PRISM
tools are written to handle multiple channels simultaneously. The beam_ids parameter is an array that contains n_beams channel IDs, indicating which channels this tool is processing. The n_ptrs parameter specifies how many input data buffers are coming to this tool. Usually this is n_beams or a multiple. The in_ptrs parameter is an array of n_ptrs addresses of input buffers. These input buffers are physically the output buffers of the tools that produce the data used by the tool. The wk_space parameter is the address of the work buffer pre-allocated for this instance of the tool. The n_out parameter specifies the number of output buffers produced by the tool. Again this is usually a multiple of n_beams. The n_params parameter specifies the number of parameters contained in the params block from the script. The params parameter is the address of the block of parameters specified for this instance of the tool in the script. The first three words of the parameter block have meaning to the script compiler and are fixed in their usage. The first specifies the number of words of work space to allocate in the tools wk_space buffer. The second specifies the number of output buffers this tools will need, and the third specifies the length, in words, of the output buffers. (Due to addressing considerations in the i860s, PRISM does all allocations in words.) The remainder of the params block is tool specific. Selected tool specific entries may be changed while PRISM is running, allowing real time tuning of signal processing. The last two calling parameters, pid and sid, are the IDs of the tool and its script. They are used mainly for debugging purposes for identifying the source of error messages. The importance of this is more obvious when one envisions debugging a system of 20 embedded processors all sharing the same window for printouts.

Since tools are the building blocks of complex signal processing chains, it is necessary to allow multiple instances of the same tool to simultaneously reside on a single processor. Each instance of the tool may have its own data that must be preserved from call to call. To prevent multiple instances of a tool from over-writing each other's data, tools must follow some simple rules to make them "re-entrant". Tools are not allowed to have global variables, static or not, that are not explicitly intended to be shared between instances of the tool. Similarly, tools are not allowed internal static variables. Any data that is intended to be available from call to call must be a part of the work space provided with the wk_space parameter or must be contained in the tools output buffers. Different instances of the tool may have different amounts of workspace as specified by their params blocks, and each is guaranteed its own is safe from other tool instances. Tools are not allowed to change the contents of their input buffers. Two tools may both have the same input buffers. Also, their input buffers are the output buffers of another tool which has been guaranteed their integrity. Changes by one would corrupt the data for the other.

4.3. Scripts
Scripts are the programming language of PRISM. They are the means by which a signal processing chain is specified to the processors for execution. Scripts specify which tools implementing a signal processing chain execute on which processors. They denote the data flow from tool to tool and from processor to processor. They also denote which tools receive the raw input to the system. For each tool in a script a (not necessarily unique) parameter file is specified. This file governs the characteristics and actions of this particular instance of the tool. The parameter file's contents can be used in the tool's initialization and are passed to the tool via the n_params and params calling arguments described previously.

For a particular task, there are multiple scripts, at least one per array processor used, contained in the script file. These scripts are compiled by the host and downloaded to their respective array processors where the control program executes them continuously. Section 6 steps through the development of a simple script.

4.4. Configuration Files
A configuration file is a text file which specifies the hardware configuration and the input and output data structures for PRISM. It consists of eight distinct system descriptions which allow a user to provide the PRISM software with picture of the hardware layout, communication paths, memory partitions, software to hardware mappings and desired system options. Each description has a keyword identifier and a set of attributes defined by keywords. The file is read in at PRISM startup and decomposed by a "yacc" application which facilitates changes and additions to the description attributes. A user defines the following attributes as required by either modifying an existing configuration file or creating a new one.

"IO.input.attributes" - defines the raw data input buffer sets. Each set requires the number of channels to allocate on a board; data element size in bytes, number of data elements per slot and number of slots per channel which define the buffer size; and the board id where this set of channels is to be located.

"Display.processes" - defines the maximum and actual number of display output processors to be active for this configuration.

"Board.attributes" - defines each board/external processor for this configuration. Each board requires an id number, category (AP, Memory, External Processor, etc.), type within a category, communication mode (mc_sockets, VME, VSB, commserver, etc.), communication id number, size of memory, and VME/VSB base addresses.

"DRAM.SW.attributes" - defines the dram id number, type of DRAM (Master/Slave), and associated board id number for each board being used as PRISM memory.

"AP.SW.attributes" - defines the characteristics needed for down loading each array processor. Each AP requires
an ap_id number for logical-to-physical mapping, a device id number to associate it with a device driver, associated board id number, onboard shared memory size, stack and heap sizes, and the name of the image to be downloaded.

"Conserver.attributes" - defines the communication path between processors which are not APs, and the host. Each processor requires a client id number, the name of the file defining its client parameters, and the names for the major and minor receiver/sender functions. The conserver is a set of PRISM utilities which provide the user with a message-based means to communicate between tasks, both intra-processor and inter-processor.

"Display.input.data" - defines data item sizes, number of data items per slot, number of buffer slots, and data size for each display output buffer.

"System.sw.attributes" - defines the overall operating state for PRISM in this configuration. These states are: user interface - terminal mode or Xwindows, conserver - on or off, startup - automatic or manual transition to run state, recall capability - on or off for recording data for single channel playback during real time, archive - on or off for recording input data to tape, and audio - on or off for capability to generate audio output.

5. Performance
The PRISM 2.2 system has been tuned for speed while preserving generality. There are three major areas of concern in performance: tool speed, control overhead, and data movement. For tools the process was simple. Dynamic allocation is limited to initialization. Algorithms were analyzed for efficiency of implementation and extensive use was made of Mercury supplied micro-coded vector calls, allowing all tools but byte to float conversion to be written entirely in "C" for ease of reuse. Byte to float was micro-coded since Mercuries don't handle byte operations well. Timings for some tools are shown in Figure 5.1.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte to Float</td>
<td>92 usec</td>
<td>512 samples</td>
</tr>
<tr>
<td>FFT</td>
<td>194 usec</td>
<td>1024 pt</td>
</tr>
<tr>
<td>Power Spectrum</td>
<td>517 usec</td>
<td>1024 pt, inc hammer wind</td>
</tr>
<tr>
<td>Log Base R</td>
<td>95 usec</td>
<td>256 pt</td>
</tr>
<tr>
<td>Bid Neural Net Ret</td>
<td>103 usec</td>
<td>1400 pt retina</td>
</tr>
<tr>
<td>Naur Net Feed Fwd</td>
<td>2.2 msec</td>
<td>1400 pt retina</td>
</tr>
</tbody>
</table>

Table 5-1. Tool Timings

The control program does a great deal of initialization of scripts prior to processing. Tools within a script are initialized such that the input buffers of a tool are physically the output buffers of the tools from which it receives data. Tools are called only if they have enough data to execute. The overhead for transitioning from one tool to the next, including determination of data availability, is less than 10 usec. The transition between scripts on the same processor takes 12 usec. Since tools process multiple channels in the same call, the impact of the control overhead is further minimized.

Data movement is the most complex and flexible of the performance issues. Data going to or from Mercuries automatically travels via the fastest method available to the Mercury. Table 5-2 shows the Mercury performance via the methods available. As can be seen, the other party in a transaction with a Mercury is the limiting factor. Where data resides is specifiable via the scripts and config file. Data can reside on the producer, consumer (faster), or any other addressable memory in the system. This flexibility allows tradeoffs between throughput speed and memory availability. In one implementation, simply relocating the pseudo-channels from the DRAM to the Mercuries reduced pseudo-channel VME bus traffic by 13% of the original load. The relocation of the pseudo-channels only involved changing dump_data tool parameters in the script.

<table>
<thead>
<tr>
<th>Bus</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>VME 32D32</td>
<td>37 Mbytes/sec</td>
</tr>
<tr>
<td>VME 32D64</td>
<td>71 Mbytes/sec</td>
</tr>
<tr>
<td>VSB</td>
<td>37 Mbytes/sec</td>
</tr>
<tr>
<td>Quad-Board Cross Bar</td>
<td>160 Mbytes/sec</td>
</tr>
<tr>
<td>IB Bus</td>
<td>160 Mbytes/sec</td>
</tr>
</tbody>
</table>

Table 5-2. Mercury Bus Throughput

6. Example Usage
This section steps through the creation of a simple script from a signal processing diagram. First, we design our desired processing. Figure 6-1 gives a block diagram similar to what a signal processing engineer might provide. Assume that the data is coming in as 10 channels of 8 bit signed integer samples and that all 10 channels require the same processing. Also assume that the output of the two processing paths is to be synchronized and displayed side by side on a waterfall type display.

In Figure 6-2 we see an expansion of this processing chain to conform to the tools necessary to implement the details of the chain. The diagram assumes the input data rate is such that the entire processing chain can be accommodated by a single processor. Note the addition of the sync tool. Even though the two chains have an output every 512 point input scans, the overlap tool consumes one of these in its path. The sync tool can be used to throw away the first output of the other path to synchronize the two paths prior to display packaging. The script generated is shown in Figure 6-3.
Figure 6-1. SP Block Diagram

Figure 6-2. Revised SP Diagram

Figure 6-3. Script