HIGH-LEVEL LANGUAGE PROGRAMMING ENVIRONMENT FOR PARALLEL REAL-TIME TELEMETRY PROCESSOR

John R. LaPlante
Steve G. Barge

LORAL INSTRUMENTATION
8401 Aero Drive
San Diego, California 92123-1720

ABSTRACT

The difficulty of incorporating custom real-time processing into a conventional telemetry system frustrates many design engineers. Custom algorithms such as data compression/conversion, software decommutation, signal processing or sensitive defense related algorithms, are often executed on expensive and time-consuming mainframe computers during post-processing. The cost to implement such algorithms on real-time hardware is greater, because programming for such hardware is usually done in assembly language or microcode, resulting in:

- ❑ The need for specially trained software specialists
- ❑ Long and often unpredictable development time
- ❑ Poor maintainability
- ❑ Non-portability to new applications or hardware

This paper presents an alternative to host-based, post-processing telemetry systems. The Loral System 500 offers an easy to use, high-level language programming environment that couples real-time performance with fast development time, portability and easy maintenance. Targeted to Weltek's XL-Serles 32 and 64 bit floating point processors, delivering 20 MFLOPS peak performance, the environment transparently integrates the C programming environment with a parallel date-flow telemetry processing architecture.

Supporting automatic human interface generation, symbolic high-level debugging and a complete floating point math library the System 500 programming environment extends to parallel execution transparently. It handles process scheduling, memory management and data conversion automatically. Configured to run under UNIX, the system's development environment is powerful and portable. The platform can be migrated to PC's and other hosts, facilitating eventual integration with an array of standard off-the-shelf tools.

KEYWORDS: TELEMETRY, HIGH LEVEL LANGUAGE, PROGRAMMING ENVIRONMENT, REAL-TIME, C, WEITEK, DATA FLOW, UNIX, SOFTWARE DEVELOPMENT, FLOATING POINT

INTRODUCTION

This paper addresses the programming environment available as an option on Loral's System 500 for generating custom real-time algorithms executable on the system's Field Programmable Processor (FPP) Module for telemetry data acquisition, processing, distribution, and display.

The availability of a high-level language programming environment that includes integrated debug and rapid prototyping support, greatly increases programmer productivity [1]. An original design goal of the FPP and the System 500 was to produce a general purpose computation engine that combined high speed with improved programmability [2]. For years, a major obstacle to producing applications has been the difficulty of programming available hardware. A typical design trade-off involved accepting either slow processing or microcode/assembly language development. The additional support required for building custom user interfaces further hampered this development.

The FPP in Loral's System 500 uses the Weitek 8032 floating point chip set [3], chosen for its performance and for its development support. Weitek makes available Fortran and C compilers as part of its development package, which includes assembler, linker, locater, parallelizer, simulator, and support for run-time debugging [4,5] .

DATA FLOW ARCHITECTURE

The System 500 employs an architecture known as dataflow [6,7,8]. The dataflow architecture allows any number of special purpose hardware modules to communicate via a common bus, known as the MUXbus, by identifying data with a separate identifier field known as a "tag." All data with the same tag value can be considered a separate channel and is known as a parameter". These data-tag

packets can be broadcast onto the MUXbus by any hardware module at random, following a simple handshake protocol. Modules can be set up to accept certain parameters. These parameters are buffered on the module in a FIFO; other parameters are ignored by the module. All bus interfacing and buffering is handled by special purpose hardware on each module. Thus the remainder of the module hardware is free to process data and return data to the MUXbus without regard to other module activity. In this way, any number of modules can coexist on the MUXbus, performing computations in parallel.

It is the dataflow architecture's event-driven nature that forms the basis of the FPP programming environment. Central to the environment's design goal is prompt response to data occurring on the MUXbus and shielding the programmer from the details of data tagging.

FPP ARCHITECTURE

The FPP accepts data streams from the MUXbus, performs high-speed processing on them, and returns the results as new data-streams to the MUXbus [9]. It has 64k x 32 bits of data RAM shared with a workstation host for all initialization and communication via Ethernet. It also has 32k x 64 bits of code RAM that is either EPROM or static RAM writeable from a host workstation via Ethernet. For the standard programming environment, code and data initialization is performed dynamically over Ethernet. Code-compile-test turnaround time is therefore very fast due to the high speed of the Ethernet link.

ALGORITHM ARCHITECTURE

Because any data processing on an FPP must occur in an event-driven manner, FPP algorithms are designed as if they were interrupt driven. They are activated when desired parameters occur on the MUXbus. In the FPP programming environment, the C Language concept of a "main" function, or starting point of the program, is meaningless.

Algorithms in the FPP programming environment can accept one or more different parameters from the MUXbus. Because MUXbus parameters occur at random and at different rates, providing a "read" function to access MUXbus input is very inefficient. Rather, for each parameter, the programmer specifies a function in his program to service that parameter. These input functions are known as "dedicated" functions. After system initialization, when a desired input parameter occurs on the MUXbus, control is immediately transferred to the respective dedicated function. The parameter data value is passed automatically to the

dedicated function as a function argument. Dedicated functions are called only when their respective parameter data arrives.

Data passed to a dedicated function can be manipulated or stored for future computation. In general, this is how a multiple input algorithm synchronizes several independent input parameters. The programming environment allows a programmer to declare a data structure of any size and configuration as scratch pad storage. This storage area is known as a Parameter Control Block (PCB). When any of the algorithm’s dedicated functions are invoked, the function always receives a pointer to its PCB. The PCB is shared by all dedicated functions of an algorithm.

In a multiple-source algorithm, typically one dedicated function is designated as a “trigger” function. The trigger function is usually chosen because it services the last source parameter in the PCM stream. All other dedicated functions store their incoming data in the PCB when invoked; when the trigger parameter arrives, the trigger function uses the stored inputs as well as its incoming value to perform its calculations.

Figure 1 shows the basic structure of a typical FPP algorithm source file, in English-like pseudo-code. The algorithm has two dedicated functions to handle two MUXbus parameters. The algorithm adds two separate MUXbus parameters and outputs the sum as a third parameter. The function “adder_1” services one parameter and stores the incoming data value, passed as the “MUXbus value”argument, in the PCB. The function “adder_2" services another parameter and serves as the trigger function. When invoked, it takes its incoming value, passed as the “MUXbus value “ argument, adds it to the value stored in the PCB, and then outputs the sum to the MUXbus.

This simple example underscores the flexibility and efficiency of the protocol. Although the programmer must handle intermediate data storage as illustrated, the alternative of automating such storage by the system would incur additional and possibly unwanted overhead. Allowing the programmer to choose appropriate action makes the system most flexible and efficient. Although only one function in this example is used as a trigger to perform processing and output, computation and output can take place within any function. Indeed, any function can perform computation or output freely. It is the programmer’s prerogative where to appropriate resources.

The example in Figure 2 again takes two source parameters and performs a simple, 1st-order EU conversion on the first parameter, and accumulates it as an ongoing sum. When the second parameter, the “trigger,” arrives, the accumulated total is averaged by dividing by n and is output.

Data values can also be buffered for future calculations requiring more than one occurrence of a parameter. This approach is used for digital signal processing applications such as FFT and FIR filter processing. Figure 3 illustrates a typical scheme that might be required to perform an FFT.

USER INTERFACE

The System 500 offers a system of menus for defining parameters on the MUXbus [10]. Users create parameters in the system via a menu called a “parameter page”. The user creates a parameter page for each MUXbus parameter that occurs on the bus.

Parameter pages come in many different types - generally each data source module on the MUXbus supports its own parameter page type, unique to that module’s function. Decommutation modules allow a parameter to be defined according to its position in a particular PCM stream. Tape and I/O interface modules define parameters coming to the MUXbus from a particular external device. Processing modules, such as the FPP, define processed parameters which occur as outputs from calculations based on previous MUXbus inputs.

A parameter page creates and makes known the existence of a parameter to the rest of the system. In most cases it also provides special configuration information needed to define the parameter. That is the case for FPP parameters. All FPP algorithms are written to be general purpose; the programmer does not specify which MUXbus parameters are inputs to an algorithm-the user does this on an FPP parameter page.

The FPP programming environment allows an additional way to configure an algorithm from the parameter page at runtime. Numeric values can be input directly on the parameter page and are then passed to the algorithm via the PCB. These numeric values are known as “arguments” to the algorithm. Arguments can be entered as integers in varying radices, as floating point numbers, or as a selection of options that translate to an integer suitable for branching in a case or switch statement. The latter is known as a “list-pick” item.

PAGE DEFINITION

Because FPP algorithms may vary in their requirements for number of sources, number of outputs, type and number of arguments, each algorithm requires its own custom parameter page. The FPP programming environment takes care of automatically generating the parameter page for any newly created algorithm. It uses a combination of defaults and '# define' statements in the program's source code to lay out the page. The programmer specifies:

- ❑ Number of source parameters required
- ❑ Number and type of any arguments required
- ❑ Number and type of outputs to be produced
- ❑ Screen formatting information, including prompts for each of the above, default argument values, and list-pick selections.

The generated page then interacts with the user within the window environment of the Loral System 500. Mouse and cursor movement, command line editing, and list-pick selection are supported automatically. The specification of custom parameter pages consumes a very small portion of the entire programming effort.

An FPP parameter page consists of three sections:

- ❑ A standard parameter page header which is common to all parameter pages. it includes MUXbus interface information such as output tag number and data format. The header defines the OUTPUT of the algorithm.
- ❑ One or more entry fields allow the user to select the parameters to be sources to the algorithm. These fields define the INPUT to the algorithm.
- ❑ Numerical values or list-pick selections which are passed to the algorithm via the PCB. These numerical values are ARGUMENTS to the algorithm.

Figure 4 shows a parameter page from the standard "_logical_func" algorithm created using the FPP programming environment. Figure 5 shows the C language header file used to specify this page. The highlighted fields are filled in by the programmer to specify the page displays and entry fields. The FPP_ARG_TYPE

entries define the “screen” type for each of the corresponding arguments in the Parameter Setup Page. The FPP_ARG_PROMPT entries define the text that is to appear in the Parameter Setup Page to prompt the user for the corresponding argument. The FPP_ARG_DEF entries define the initial argument value that appears in the Parameter Setup Page except when the argument “screen” type is FPP_TYPE_WHEEL, in which case FPP_ARG_DEF specifies the items selectable from a list-pick menu.

AUTOMATED DATA CONVERSION

An FPP algorithm can take as source input, any parameter that occurs on the MUXbus. All Loral telemetry systems support multiple data formats for MUXbus parameters, including unsigned binary, 2's complement, offset binary, sign magnitude, and floating point. The system also supports varying word widths from 1 to 32 bits for most data types. By default the system performs automatic data conversion on all sources coming into an FPP algorithm. Data received by an FPP dedicated function is automatically converted to either 32-bit signed integer or 32-bit IEEE floating point format for consistency with common C programming practice. The overhead of data conversion can be easily disabled if required.

MULTIPLE PROCESSES

An FPP module can be configured to process and produce many MUXbus parameters. It can use the same algorithm for more than one parameter. To accomplish this, a user must create a new parameter page for each such parameter. If each of these parameter pages specifies the use of the same algorithm, this is known as separate “instances” of the algorithm. The system keeps track of separate instances, allocating independent PCBs for each. Although the same copy of algorithm code executes for each of these instances, each essentially functions as a separate “process”, invisible, to other instances, because they have their own independent PCBs. A mechanism is available to communicate between instances via global variables if required.

DEVELOPMENT ENVIRONMENT

The Loral System 500 is a UNIX-based telemetry system [9]. Its FPP programming environment relies completely on UNIX facilities for creating and maintaining source code. Cross-compliation is done on the workstation using the Weitek C Compiler or assembler supplied with the system. Debugging occurs at this stage using either Loral’s symbolic debug environment or Weitek’s assembly language simulator (see Debug Environment). Compiled object code is then inserted into the FPP algorithm libraries using two simple Loral-supplied utilities.

This makes a newly created FPP algorithm automatically available as a selection on an FPP parameter page. Object code loading is handled transparently during system initialization over a high-speed Ethernet connection.

DEBUG ENVIRONMENT

Loral's System 500 includes a powerful simulation environment for interactively debugging algorithm source code using industry standard UNIX utilities. Algorithm source code is compiled using a special debug option. The generated code is then executed on the host workstation, and can be completely debugged using any UNIX resident debugger, such as the DBX debugger supplied with the System 500.

The DBX debugger supports full C source-level symbolic debugging. This allows setting breakpoints at specific lines in a C source file, single stepping, viewing and modifying variables by name, and a variety of other functions. Algorithms can be completely debugged prior to installation this way.

The System 500's arbiter/analyzer module provides additional run-time debug support. This module allows complete analysis of all MUXbus traffic by capturing and storing any or all data/tag pairs occurring on the MUXbus. It records the source of each data/tag pair and relative time of occurrence. The arbiter/analyzer module allows a host workstation to concurrently read and monitor the stored data. It also provides the ability to inject date/tag pairs onto the bus and perform bus loading analysis.

RUN-TIME SUPPORT

A comprehensive set of run-time libraries is included with the programming environment. These libraries supply the entire set of standard C math routines including trigonometric, hyperbolic, exponential, log, bessel, and error function support. Standard string manipulation and type conversion routines are supplied as well as support for system time and MUXbus output. Because of the real-time nature of FPP algorithms, the System 500 does not support standard I/O or system related calls such as memory allocation, file access or process manipulation.

PERFORMANCE

Typical performance for standard compression algorithms such as EU format conversion and limits checking written in C ranges from approximately 150 - 500k words/SEC. For comparison, the same algorithms written in optimized assembly language deliver approximately 200-700k words/sec.

An algorithm that measures FPP loading, from which algorithm performance is directly measurable, is supplied as part of the programming package

RAPID PROTOTYPING

Any or all of an FPP algorithm can be written in assembly language. Inevitably there are applications that require maximum throughput from a given system for various reasons. The FPP programming environment addresses these needs by providing a platform for efficient generation of optimized assembly language code.

Because of the availability of the C programming language, a high performance application can be first coded in C to demonstrate proof of correctness. High-level support can be used to debug and test a version in C, and produce demonstratable code very quickly. Once an algorithm has been checked-out this way, performing subsequent speed optimization is greatly simplified.

The Weitek C compiler can be configured with an option that directly outputs the assembly code being generated. The output is fully compatible with the Weitek assembler. When assembled, it produces exactly the same object code as the C compiler. It is relatively simple for a programmer to use the assembly language version as his source code, and modify it freely for optimal performance. The structure and correctness of an algorithm are less easily corrupted by this approach, and the creation of assembly language source code is simplified.

PORTABILITY AND MAINTENANCE

The portability of the FPP Programming environment and its upward compatibility with future Loral products was a major design consideration. The choice of C as a programming language and UNIX as a development environment facilitates this portability. FPP algorithms written in C will be directly upgradable to future processing modules compatible with the data-flow architecture of Loral systems.

CONCLUSION

System 500 users avoid the compromise of host-based, post-processing telemetry systems. They develop custom real-time applications themselves, without the skills of highly paid software specialists. The FPP programming environment greatly reduces development time and provides higher confidence in the final product. Generation of assembly code is enhanced. Upward compatibility of such applications is ensured due to the portable nature of the C programming language. This results in more predictable schedules, more dependable applications and lower overall cost for the application developer.

```
function adder_1 (MUXbus value, PCB pointer)
{
        store MUXbus value in PCB.value1
}

function adder_2 (MUXbus value, PCB pointer
{
        add inclime MUXbus value to PCB.value1

        output sum to MUXbus
}
```

Figure 1

Example of pseudo-code for two input adder function

```
Function average_1 (MUXbus value, PCB Pointer)
{
        perform 1st order Eu Conversion on MUXbus value

        add converted MUXbus value to PCB.sum

        increment PCB.count
}

        divide PCB.sum by PCB.count

        output the resulting average to the MUXbus

        zero PCB.sum and PCB.count
}
```

Figure 2

Example of pseudo-code for an averaging function with a trigger parameter

```
Function fft_1 (MUXbus value, PCB pointer
{

        store MUXbus value in PCB.buffer[PCB.count]

        increment PCB.count

        if PCB.count = size of PCB.buffer
        {
                zero PCB.count

                calc_fft (PCB.buffer)
        }
}
```

Figure 3

Example of pseudo-code for algorithm that buffers data for FFT calculation

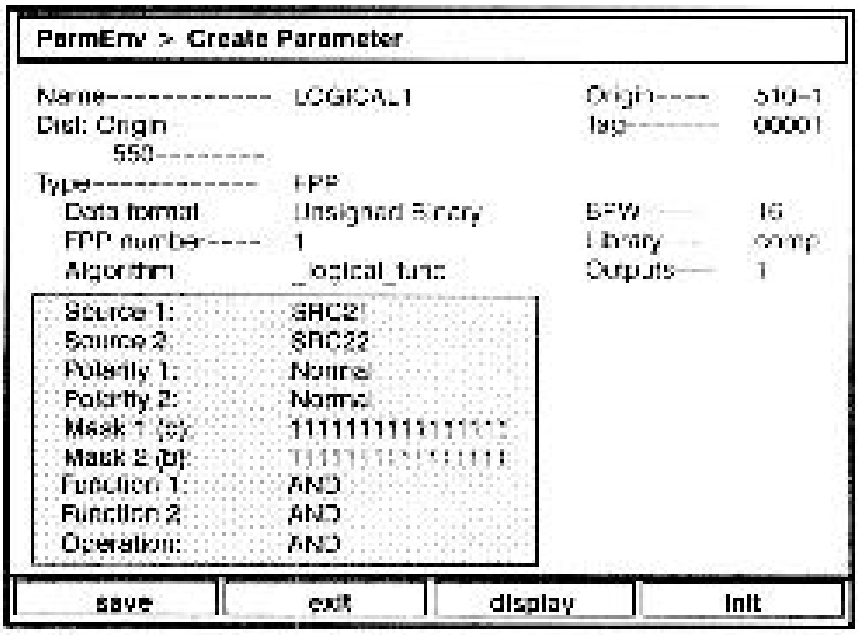

Figure 4

Parameter Setup Page for _logical_func algorithm.

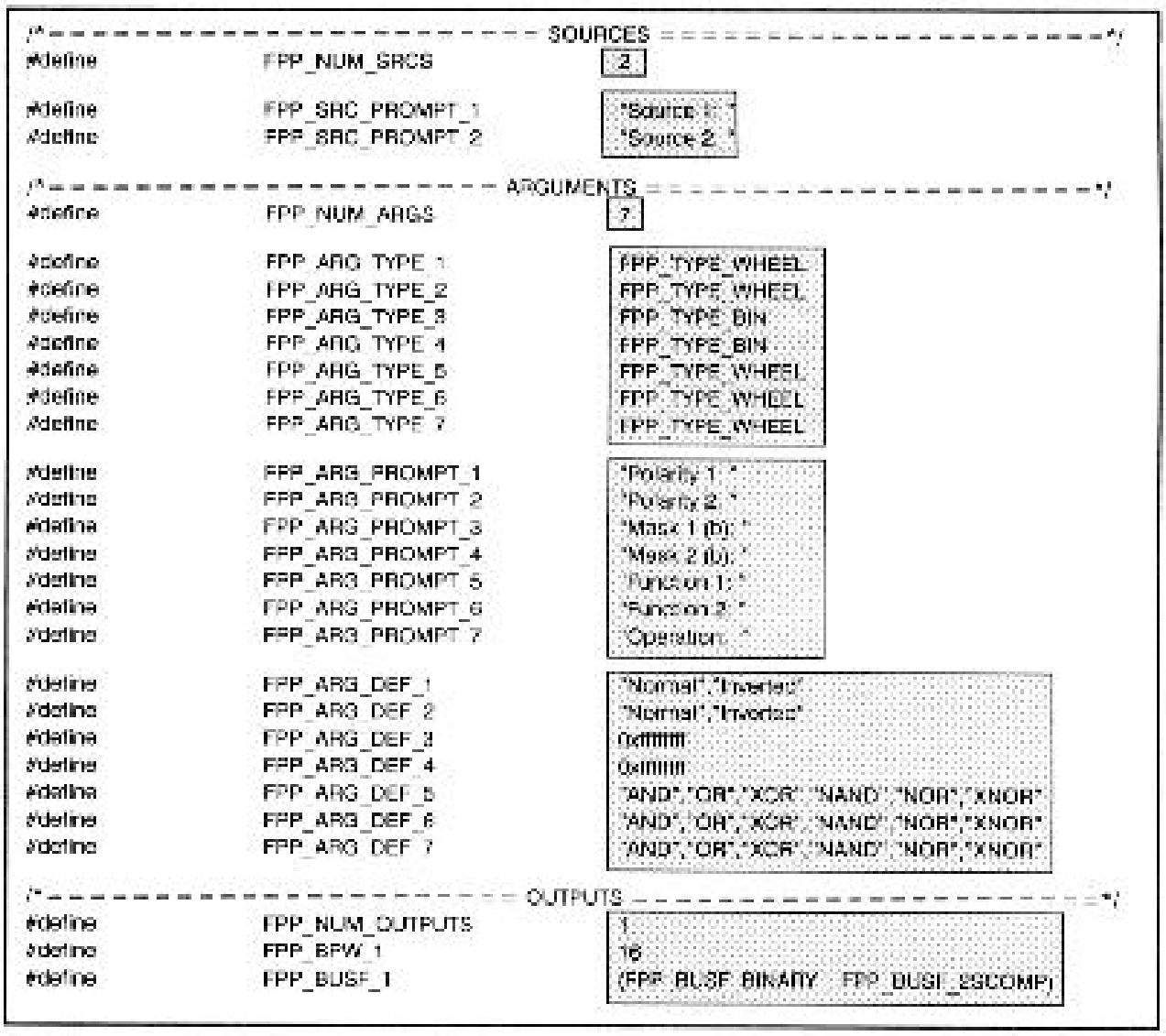

Figure 5

Header file for Loral's standard_logical_func algorithm

REFERENCES:

1. Boehm, Barry W. “Improving Software Productivity”, IEEE Computer Magazine, Sept, 1987, p. 43.

2. Williamson, Gale, “User Programmable Real-Time Processing in Modern Telemetry Systems”, Proceedings of International Telemetering Conference, Vol. XXI, 1985.

3. XL-Series Overview, Weitek Corporation, Sunnyvale, CA.

4. XL-Series C Compiler User’s Guide, Weitek Corp., Sunnyvale, CA.

5. XL-Series Programmer’s Reference Manual, Weitek Corp., Sunnyvale, CA.

6. Gurd, J. and Watson, I. “Data Driven Systems for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution”, Computer Design, June 1980, p. 91.

7. Dennis, J.B., “Data Flow Supercomputers”, IEEE Computer Magazine, November, 1980, p. 48.

8. Crawford, Michael A. and Sweitzer, Ralph F., “Bus Structured Software for a Modern PCM Decommutator”, Proceedings of International Telemetering Conference, Vol. XVII, 1981.

9. Powell, Richard L. et.al., “High Performance, Real-Time Parallel Processing Telemetry System”, Proceedings of International Telemetering Conference, Vol. XXIV, 1988.

10. Querido, Robert and Friedman, Paul J., “Distributed, Real-Time, High-Resolution Color Graphics Display System for Telemetry”, Proceedings of International Telemetering Conference, Vol. XXIV, 1988.

11. Sharp, Kirk and Thompson, Lorraine, “Adaptation of a Loral ADS 100 as a Remote Ocean Buoy Maintenance System”, Proceedings of International Telemetering Conference, Vol. XXV, 1989.