ALACRON

RT860 Software User's Guide

30002-00111

Notice

Alacron, Inc.
71 Spitbrook Road, Suite 204
Nashua, NH 03060 USA

telephone: (603)891-2750
fax: (603)891-2745
electronic mail:
sales@alacron.com
support@alacron.com

Document Name: RT860 *Software User's Guide*
Alacron Part Number: 30002-00111
Document Number: 02-00111.DOC

Revision History:

1.0	01-Aug-95
1.1	22-Aug-95
1.1.1	24-Aug-95

Table of Contents

INTRODUCTION 1

Main Board 1

Processor Board ANSI Compatibility 3

System Call Handling 4

Slave Mode 4

Stand-Alone Mode 6

SLAVE MODE PROGRAMMING 9

Summary 9
Device Management 10
Read/Write Processor Board Memory 10
Memory Management 11
System Call Handling/Program Completion 11
Program Loading, Invocation and Status 11
Interrupt Management 12
Error Handling 12
Windows 3.1 Management 13

Slave Mode Programming Examples 13

Virtual and Physical Addresses 14

Dynamic Memory Allocation 14

Slave Mode Example - Passing Arguments 14

Slave Mode Example - Vector Library 15

Additional Examples 17

PROCESSOR BOARD PROGRAMMING 19

Processor Board Tools 19

Processor Board Library 20

Multiple Processor Extensions 20

Application Libraries 20

Daughtercards 21

Daughtercard Libraries 22

Debugging Processor Board Applications 22
Debugger 22
Print Statements 23
Memory Dumps 23

MULTIPLE PROCESSOR EXTENSIONS 25

Execution Model 25

Parallel Algorithms 26

Memory Contention 27

Non-Memory Intensive Algorithms 27

Cache the data 27

Strip-Mining 27

Memory Intensive Algorithms 28

Tutorial 28

Compiling/Linking/Running 28
Thread control 29

Thread control 30
Overview 30
Starting Threads 30
Querying Thread Status 32

Mutual Exclusion and Synchronization 32
Mutual Exclusion 32
C Runtime Library 37
Synchronization 37

Bit Counter 39

MMU Page Size 42

Inter-Processor Interrupts 46

Miscellaneous Functions 49
Retrieving Processor ID 49
mp_printf/mp_fprintf/mp_assert 51

Sharing Memory 51
Write-Through Cached - Snooped 51

Write-Back Cache 51
Partitioning Data 52
Making the Caches Coherent 53
Programming Hints 57

WINDOWS 3.1 PROGRAMMING 59

Features 59

Using W3SLV860 60

USER EXTENSIBILITY 63

BUILDING YOUR APPLICATION 71

RT860 Default Directory Tree Location 71

Example Code 72

Slave Mode Programs 72

Processor Board Programs 73

INTEL 860 PRIMER 75

Basic Architecture 76

Cache Issues 76

Four Megabyte Pages 78

BUS MASTERSHIP 79

ISA 79

PCI 79

VME 79

VIRTUAL ADDRESS SPACE 81

Overview 81

FT200-AT/AL860-V/FT200-V 81

FT200-V Memory Map 82

GETTING HELP 85

Troubleshooting 85

Technical Support 85

INDEX 89

Preface

Purpose of This Manual

The *RT860 Software User's Guide* provides an overview of Alacron processor board programming for application developers. The manual is intended to serve as a comprehensive introduction to the process of integrating Alacron processor boards, daughtercards and application libraries into an application.

Please refer to the *RT860 Software Reference Manual* for a detailed description of the host and processor board library functions.

Audience

This reference manual is intended for technical personnel responsible for developing application software that uses an Alacron processor board. This manual assumes familiarity with the C and FORTRAN programming languages which are used to develop the applications that execute on the Alacron processor board.

Manual Organization

The *RT860 Software User's Guide* is divided into the following sections:

Chapter 1, "Introduction" provides an overview of the RT860 Runtime Software.

Chapter 2, "Slave Mode Programming " discusses topics related to Slave Mode Programming such as device management, accessing processor board memory, memory management, system call requests, loading and calling programs and more.

Chapter 3, "Processor Board Programming" provides a brief introduction to the topics such as building the processor board application, working with application libraries and daughtercards.

Chapter 4, "Multiple Processor Extensions" contains a fairly extensive overview of programming on multiple processor boards.

Chapter 5, "Windows 3.1 Programming" provides instruction in the use of the Slave Mode library in a Windows 3.1 environment.

Chapter 6, "User Extensibility" provides instruction in the use of the Slave Mode library in a Windows 3.1 environment.

Appendix A, "Building Your Application," shows how to build a basic Slave Mode and processor board program.

Appendix B, "Intel 860 Primer," discusses topics that are specific to the Intel 860 CPU such as XR/XP differences, caching and 4 MB paging.

Appendix C, "Bus Mastership," covers the ability of processor boards to master the host bus.

Appendix D, “Virtual Address Space,” shows the allocation of the virtual address space on certain processor boards.

Appendix E, “Getting Help,” talks about obtaining technical support from Alacron.

Notation Conventions

Alacron documentation uses *italics* to introduce or define new terms. For example:

> The DIO daughtercard is intended to be used in conjunction with additional hardware, a *mezzanine card*, which handles the interface to some external device or bus.

Boldface is used to emphasize words within the text. For example:

> The user **must** take precaution against damage from static electricity when handling the processor board.

`Computer font` is used to represent text that would appear on your display screen. For example, the user may change directory into the default FT200 directory tree by typing the following:

```
cd \usr\ft200
```

Preface

Glossary

Alacron processor	The processor on an Alacron board.
Alacron processor board	An ISA, PCI or VME compatible board, with one or more processors, that is used in Image Processing, DSP or other types of applications.
Cache	Some CPU's move code and data to a private data buffer, internal to the chip, before reading them. See the chapter titled *Intel 860 Primer* for information on the i860 cache.
COFF	Common Object File Format. COFF is a file specification that describes the file structure of an Intel 860 executable file.
Daughtercard	An expansion card that plugs into the Expansion Connector on the main board. More than two daughtercards may be configured in the two expansion connectors by configuring the daughtercards in a stack. See Daughtercard Stack.
Daughtercard Library	A function library built to run on the processor board which allows an Alacron processor to control a daughtercard.
Daughtercard Processor	A processor on a daughtercard which plugs into an Alacron processor board expansion connector.
Daughtercard Stack	Certain daughtercards may be stacked, on top of the other, and plugged into a single expansion connector. In this manner, several daughtercards may be configured on a single main board. See Functionally Stackable.
Expansion Connector	Alacron processor boards have connectors which mate with expansion modules that provide additional functionality. These connectors are often called Expansion Connector 1 and 2. The boards that plug in are labeled Daughtercards 0 and 1 respectively.
Functionally Stackable	Daughtercards that are able to share an Expansion Connector are functionally stackable. More than one Functionally Stackable daughtercard may be configured in a daughtercard stack. See mechanically stackable.
Host	The Alacron processor board normally plugs into an ISA, PCI or VME computer. That computer is often referred to as the host computer or simply, the host.
Host Processor	The main CPU of the host computer. Often this is an Intel X86 or Sparc processor.
Kernel	The processor board runs an executive control program which provides support to the user application. Kernel functionality includes handling system calls, traps, fatal errors and initializing hardware resources. The Kernel interfaces with the host and user application in a variety of ways. Since the Kernel runs completely on the Alacron board, it remains unchanged when the processor board is ported to a new environment.
Main Board	A board that plugs into the host backplane. A main board usually contains a CPU, DRAM and two expansion connectors. Daughtercards are mounted on the main board.
Main Processor	The CPU on an Alacron processor board. On a multiple processor Alacron board, the

	main processor is processor 0.
Mechanically Stackable	Mechanically stackable daughtercards may not share a single Expansion Connector. Each mechanically stackable daughtercard must be connected to a private Expansion Connector. Thus, only two mechanically stackable daughtercards may be configured on a main board. The only exception is that mechanically stackable cards *may be stacked with memory daughtercards*.
RT860	Synonymous with the term *Runtime software*. See rt860.
rt860	When written in lower case, rt860 refers to a specific utility that operates stand-alone mode.
Runtime Software	The software utilities, host libraries, processor board libraries, include files and source code that facilitate the use of an Alacron processor board. In short, the entire package of software, excluding the compiler tool set and processor board libraries, that accompanies an Alacron processor board.
Slave Mode	A method in which a program executing on the host makes calls to an Alacron library in order to control the processor board. See Slave Mode Library.
Slave Mode Library	A host library that contains a set of functions that allow the host to control the Alacron processor board. The Slave Mode library contains functions which download application code to the board, read and write processor board memory, invoke functions on board-level processors and service the processor board’s system calls requests.
Stand-Alone Mode	A utility called rt860 allows a user to execute code on the processor board without having to write a Slave Mode program. Stand-Alone mode is preferred when the host processor is not required to manage other host devices or resources while invoking an application on the Alacron processor board.

Related Information

This manual assumes that you are already familiar with the information in the user’s manual for your PC or workstation. In addition, the following documents provide additional reference information:

- *AL860-AT and FT200-AT Hardware User’s Guide*
- *AL860-PCI and FT200-PCI Hardware User’s Guide*
- *FT200-V Hardware Reference Manual*
- *RT860 Software Reference Manual*
- Intel Corporation’s *i860Ô Microprocessor Family Programmer’s Reference Manual*

1

Introduction

Alacron produces data acquisition, processing, and I/O modules for a variety of application areas including Image Processing, Digital Signal Processing, Neural Networks, Compression, Data Acquisition, Graphics and more.

The Alacron architecture is based on the selection of a main board which plugs into industry standard ISA, VME or PCI backplanes. The *main board* houses a *main processor* which serves as both an administrator and as a compute engine. If additional compute power is required, additional processor(s) may be added on the main board or on an expansion module.

The main processor is also commonly known as processor zero.

Likewise, expansion modules may be used to add memory or to provide direct interfaces to a variety of industry standard interfaces: SCSI, VSB, Digital Cameras, Data Translation DT-Connect I & II, VGA displays and more.

Hand coded application libraries and optimizing compilers harness the compute power of Alacron board processors for specific needs. We currently support separate optimized libraries for Image Processing, DSP, compression, and more.

In short, the combination of processing, I/O and application libraries forms a subsystem which may be adapted to a variety of different application requirements. The modularity of the system allows the user to choose the level of price/performance that meets the needs of a given requirement.

Main Board

The Alacron main board contains a processor, DRAM and two expansion connectors. The Alacron processor board has a high-speed *local bus* which serves as the basis for the board's architecture. A basic board may be configured as shown in Figure 1: Basic Processor Board Architecture. The processor and any devices connected to the expansion ports may have direct, master mode access to DRAM. Similarly, processor(s) and devices on the host computer may also access the Alacron board's DRAM over the host bus, but host bus traffic is often minimized due to it's slow speed relative to the Alacron board local bus.

Introduction

The choice of the main processor has the biggest impact on the price and performance of the main board. AL860 series boards are configured with an Intel 860 XR processor while FT200 series boards contain an Intel 860 XP processor. Table 1: Main Board Scalability below shows the range of processor scalability in main boards.

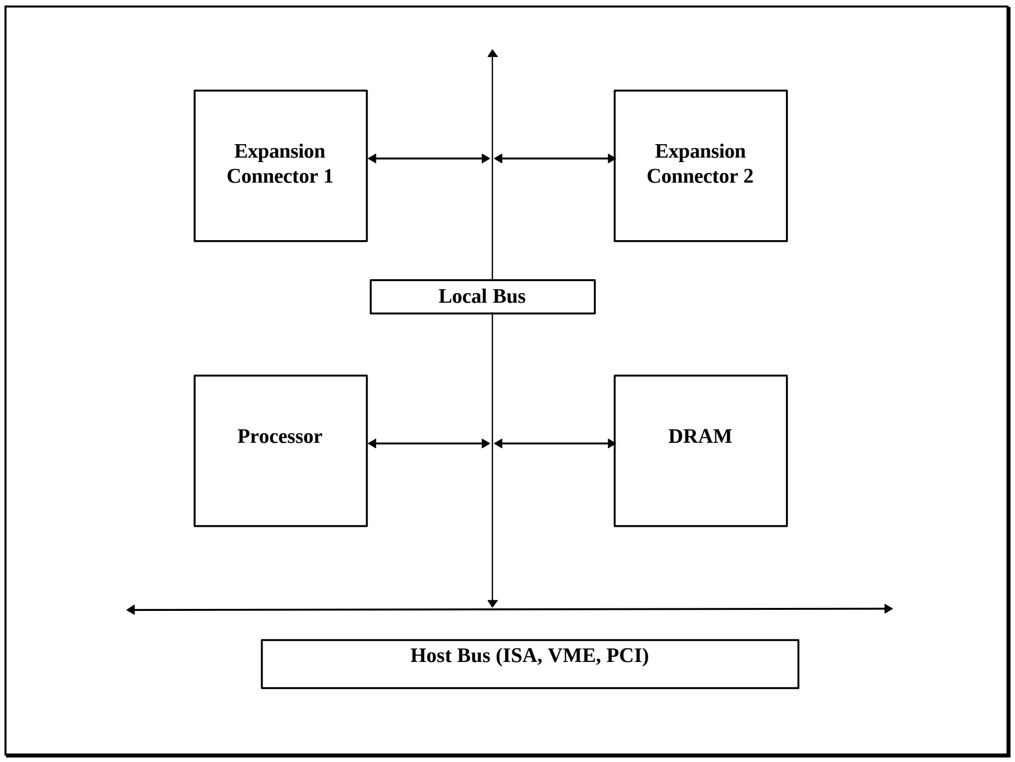

Figure 1: Basic Processor Board Architecture

Main Board	AL860 25 MHz	AL860 40 MHz	FT200 Single	FT200 Dual
Main Processor	**i860 XR**	**i860 XR**	**i860 XP**	**i860 XP**
Clock Speed	**25 MHz**	**40 MHz**	**50 MHz**	**50 MHz**
Number of Processors	**1**	**1**	**1**	**2**
MFLOP's	**50**	**80**	**100**	**200**
Code Cache	**4 KB**	**4 KB**	**16 KB**	**16 KB**
Data Cache	**8 KB**	**8 KB**	**16 KB**	**16 KB**
Maximum DRAM Size - PCI	**256 MB**	**256 MB**	**256 MB**	**256 MB**
Maximum DRAM Size - ISA & VME	**64 MB**	**64 MB**	**272 MB**	**272 MB**
Peak Memory Access Speed	**100 MB/sec**	**160 MB/sec**	**200 MB/sec**	**200 MB/sec**

Table 1: Main Board Scalability

Processor Board ANSI Compatibility

The Alacron processor board is essentially a computer in and of itself. Indeed, the user may compile and link standard ANSI C and FORTRAN source code and run it directly on the Alacron board; usually with no changes.

The primary difficulty in performing this task seamlessly regards the fact that the Alacron main board has no direct connection to a keyboard, console, disk file system or other physical devices. Standard ANSI source code contains system call specifications for input and output to these common devices. For example, the ANSI C function **printf()** writes a character string to the standard output device which is usually the display. In order to achieve ANSI compatibility, an architecture must provide a mechanism to support system calls such as this.

It is common for a compiler to rely on the operating system to handle device input and output. For example, in the case of **printf()** above, the compiler makes a system call to perform a **printf(),** but leaves the implementation of the **printf()** function to the operating system. This is also true of compilers that are used with Alacron processor boards since they are generic in nature and have no awareness of the particularities of the Alacron processor board architecture.

The operating system on the Alacron board is called the *Kernel*. It executes on the main processor and performs system call requests on behalf of the user application. All of the architecture specific operations that must be performed in order to run ANSI source code are embedded in the Kernel and are *hidden from the user application.*

Introduction

System Call Handling

When a user application, written for the processor board, performs a system call, the compiler generates a trap instruction to pass the program flow of control from user context to Kernel context. The trap instruction passes, to the Kernel, the system call arguments and a trap code, which identifies the particular system call being executed. The Kernel examines the trap code and decides if the system call can be executed locally. For example, the function **malloc(),** which allocates memory off the heap, is executed locally. Also, the **time()** system call may be implemented using the on-board Real Time Clock device. However, other system calls, such as file I/O, require host support.

The Kernel prepares to pass a system call request to the host by initializing a message area. Once the system call trap code and system call arguments are written to the message area, the processor board Kernel sends an interrupt to the host and blocks until the host messages back that the system call has been completed. A processor that performs a system call waits, in the Kernel, for the host to service it before continuing.

The host interrupt handler sets a flag that a processor board system call request has been received and exits. The host will not actually service the system call request until the user application running on the host makes a function call to specifically do so.

Why doesn't the host simply poll the message area in processor board memory to check for requests from the Kernel? The answer is that processor boards are designed with a single ported memory. Host accesses to processor board memory use up local bus bandwidth. With the interrupt handshake, the host detects an interrupt request by polling a flag in it's own memory. This has no affect on processor board performance.

When the host makes the function call required to detect and service system call request(s) from the processor board, it reads the trap code and system call arguments from the messaging area in processor board memory. Once this is done, the host translates the trap code to a specific system call and executes the requested system call. The function return value from the system call is written back to the message area which allows the processor board Kernel to unblock and return program control to the user context. After returning to user context, the user application continues, having received the system call return code from the Kernel.

Slave Mode

The user application running on the processor board executes system calls using the runtime context of the host program or host process which services processor board system call requests. The host program performs several important tasks that facilitate the execution of an application on the processor board:

- Opens the processor board device.

- Loads the processor board Kernel software and the executable user application into processor board memory. Initializes the Kernel data areas. Starts the processor board Kernel.
- Requests that the Kernel start executing the user application.
- Services user application system calls, and waits for user application completion.
- Closes the processor board device.

Alacron has developed a library with separate functions for each of these tasks. This library is called the *Slave Mode library*. The host program that makes calls to the Slave Mode library is called a *Slave Mode program*.

> *The historical terms* **host** *and* **slave** *are used in this discussion, although the locus of activity is often on the main board not the host. In fact, the main board is often "*host*" to other devices.*

Every Slave Mode program must call the function **alopen** to open the processor board device. Processor boards are assigned a unique unit number starting at zero. The argument passed to **alopen** is the specific unit number of the board that the Slave Mode program wishes to open.

Afterward, the Slave Mode program must select a specific board by calling the **aldev** function. The argument passed to **aldev** is, once again, the device unit number. Subsequent Slave Mode functions will operate on the device specified by **aldev**.

The Kernel and user application are loaded by calling the Slave Mode function **almapload**. The almapload function takes two arguments: the file name of the user application (an executable file compiled and linked for the processor board) and the amount of space to allocate on the processor board for runtime stack growth.

Before starting a function on the processor board, the Slave Mode program must first make a call to function **aladdr**, passing it a string argument specifying the name of a processor board function. Function **aladdr** will check the symbol table of the user program loaded by **almapload** and return the function entry point, i.e. the start address of the function.

That address may be passed to the Slave Mode function **alcall** which sends a request to the processor board Kernel to start executing code at that address.

The function **alwait** waits for the processor board application to complete and services all system call requests that are generated along the way. Function **alwait** is one of a group of functions that the host can call to service processor board system call requests.

Last, the function **alclose**, which is passed a device unit number, is called to close the processor board device.

An example of a basic Slave Mode program is shown below. The program executes the user application userapp on the processor board. In other words, userapp is a file on the host file system that has been built, using a cross compiler, assembler and linker, to execute on the processor board. The user application is started in function main.

In order to keep the example simple, no error checking is done. In a real Slave Mode program, it is important to check the return values of functions to ascertain whether errors were detected.

```
int dev;
unsigned long addr;

dev = 0;

/* open device 0 */
alopen(dev);

/* select device zero */
aldev(dev);

/* Load the processor board Kernel and the user application
userapp.  The stack size is 64 KB; the default */
almapload (“userapp”, (long) 0x10000);

/* get the address of function main in program userapp */
addr = aladdr(“_main”);

/* Processor board starts executing function main.  Main()
 * takes no arguments. */
alcall(addr, 0);

/* service system calls and wait for completion of userapp */
alwait();

/* close the processor board device */
alclose(dev);
```

Stand-Alone Mode

The short program above is quite powerful. For instance, consider the *hello world* program below:

```
main()
{
printf (“hello world\n”);
}
```

It is quite straightforward to compile and link this program on the host and run it. However, the program could be compiled, linked and run on the Alacron board just as easily using the Alacron board ANSI C compiler and the Slave Mode program above.

As a matter of fact, a large class of users require little more of a Slave Mode program than the one above; specifically those users that do not require host and Alacron board processing to occur in parallel. This restriction is borne out of the fact that the call to function **alwait** in the above program ties up the host processor until the Alacron board is done.

One notable limitation is that the program above hard-codes the file name *userapp*. However, this could be quite easily resolved by using the C language argv and argc arguments to allow the user to specify the processor board executable file name on the command line. Once taken off the argument stack, the file name could be passed to the **almapload** function. In so doing, we would create a utility that very simply runs function **main()** of the executable file whose name the user specifies on the command line.

As a matter of fact, what is being described is the core functionality of the Alacron utility **rt860**. Bells and whistles aside, **rt860** is essentially the Slave Mode program that is shown above with the enhancement of allowing the user to specify the processor board executable file on the command line. For example, here is how one would execute the function main in file userapp using **rt860** on an MSDOS machine:

```
C:> rt860 userapp
```

If userapp consisted of the hello world program, then the string **`hello world`** would be printed to the console, much as one would expect. However, the program would have executed *on the Alacron processor board*.

This method of using the Alacron board is called Stand-Alone Mode. Users of Stand-Alone Mode use the **rt860** utility to run their user application instead of writing their own Slave Mode program. The **rt860** utility takes the place of the Slave Mode program. Whereas Slave Mode users must write both a Slave Mode program and a processor board application, Stand-Alone users only need to develop a processor board program.

The next chapter covers Slave Mode in much greater detail. In general, you will need to write a Slave Mode program if you wish to have your application control host devices besides the Alacron board. Generally speaking, you will also need to use Slave Mode if your Alacron board is hidden underneath a GUI that runs on the host. If so, proceed to the next chapter.

The main exception to these generalities is if you decide to use User Extensibility. However, User Extensibility is an advanced topic which will be covered later.

If you plan to use Stand-Alone mode, you may choose to skip the next chapter on Slave Mode programming and proceed to the section on processor board programming. However, an understanding of how Slave Mode works is fundamental to working with the Alacron architecture.

2

Slave Mode Programming

This section describes the details of Slave Processor Mode Programming. In addition to defining Slave Processor Mode, and summarizing the support programs and libraries, several short examples are provided.

The Slave Mode program may control multiple processor boards by using the **aldev** function, shown below, to select a particular unit. Most of the functions in the Slave Mode library execute a command on one board only, but a few, like **alwaitany** and **mpwaitany**, execute on all *open* processor boards.

Many of the Slave Mode functions have two flavors which are distinguished by the prefix of the function name. Generally speaking, functions that begin with the letters *al* perform an operation with respect to the main processor, i.e. processor zero. Functions that begin with the letters *mp*, take a processor number as an argument and may be executed on any processor on the main board. For example, the function **alcall** is used to start a function on processor zero. Alternatively, function **mpcall** may be used to start a function on any processor.

AL860-AT processor boards do not support any of the Slave Mode functions with the mp prefix.

Please refer to the section on *Building Your Application* for information on building a Slave Mode program.

Summary

Slave Mode library functions perform a variety of functions which may be grouped by functionality. The list of functions is shown below:

Slave Mode Programming

Device Management

Every Slave Mode program will use functions **aldev**, **alopen** and **alclose**. These functions take as an argument the unit number of the processor board. On certain hosts, the **alopen** function performs a software initialization only and will pass even if the device is not present. To check for the presence of a device, use the **alsize** function which will return a size of zero or an error for a board that is not physically present.

aldev	select a processor board device
alclose	closes the processor board device
alopen	open processor board device
alreset	reset processor board
alsize	return size of processor board memory

Read/Write Processor Board Memory

This set of functions allows the host to transfer data to and from processor board memory. The host processor (Intel X86, Sparc, etc, ...) uses programmed I/O to perform the transfer.

PCI processor boards have a DMA controller which allows the processor board to perform transfers to/from PCI memory. VME processor boards are also able to master the VME bus, however, ISA processor boards are unable to write to host memory, only the host may initiate the operation. The functions below are all host initiated. For information on processor board memory access to the host bus, please refer to the appendix entitled *Bus Mastership.*

If the processors on your Alacron board have on-chip internal caches, be sure to use *uncached* memory references when accessing a buffer that is shared between the board processor and an external device, such as the host or an expansion connector device or daughtercard.

algetb	read a byte from processor board memory
algetba	read a byte array from processor board memory
algetw	read a 16-bit word from processor board memory
algetwa	read a 16-bit word array from processor board memory
algetl	read a 32-bit longword from processor board memory
algetla	read a 32-bit longword array from processor board memory
alsetb	write a byte to processor board memory
alsetba	write a byte array to processor board memory
alsetw	write a 16-bit word to processor board memory
alsetwa	write a 16-bit word array to processor board memory
alsetl	write a 32-bit longword to processor board memory
alsetla	write a 32-bit longword array to processor board memory

Slave Mode Programming

Memory Management

The processor board CPU runs with it's Memory Management Unit (MMU) enabled. This functionality performs virtual to physical memory translation, page level memory protection, access mode management and more. Refer to the primer on your processor for more information.

The **aladdr** function is used in nearly every Slave Mode program to look-up global symbol names in the processor board program's symbol table. If the host wishes to call a function, it uses **aladdr** to obtain the virtual address, in processor board space, for that function. If that virtual address needs to be translated to a processor board physical address, then either **VtoP** or **mp_vtop** are used.

aladdr	provides the address of a processor board program global symbol
AllocPage	allocate physical memory
alinvalidate_caches	force reload of symbol table from processor board memory
ClrMMU	initializes and clears the directory page table
MapVtoP	creates a virtual to physical mapping
mp_vtop	perform virtual to physical translation on specified processor
VtoP	perform virtual to physical translation on main processor

System Call Handling/Program Completion

A program running on the processor board will block if it performs a system call that requires host support. The host must make a function call to one of the Slave Mode functions below in order to service processor board system calls. These functions also provide the Slave Mode program with a way to detect that the last function started by **alcall** or **mpcall** has terminated.

The differences between these functions are somewhat subtle and the short descriptions below may be misleading. Please refer to the reference entries on these functions in the *RT860 Software Reference Manual.*

alintstat	handle system call requests from main processor and return
alwait	wait for main processor to complete and handle system calls
alwaitany	wait for main processor on any board to complete and handle system calls
mpintstat	handle system call requests from specified processor and return
mpwait	wait for any processor to complete and handle system calls
mpwaitany	wait for any processor on any board to complete and handle system calls

Program Loading, Invocation and Status

Nearly all programs will use the function **almapload** to load the processor board Kernel and the user application into processor board memory. Functions are started on the main board processors by calling the **alcall** and **mpcall** functions. When a function completes on the processor board, it's return value is retrieved by the **algetiresult, algetdresult, algetfresult, mpgetiresult, mpgetdresult**

and mpgetfresult functions listed below. The appropriate function to use depends on whether the processor board function returns an integer, float or double. When functions terminate abnormally, status may be retrieved by **algetstatus** and **mpgetstatus**.

alcall	starts a function on the main processor
alload	load processor board program with MMU disabled
almapload	loads a processor board executable file and the Kernel
mpcall	starts a function on the specified processor
algetdresult	get the double return value from a function
algetfresult	get the float return value from a function
algetiresult	get the integer return value from a function
mpgetdresult	get the double return value from a specified processor
mpgetfresult	get the float return value from a specified processor
mpgetiresult	get the integer return value from a specified processor
algetstatus	get the termination status of a processor board function
alprtexitstatus	print the status returned by algetstatus
mpgetstatus	get the termination status of a specified processor
mpprtexitstatus	print the status returned by mpgetstatus

Interrupt Management

The functions alinterrupt and mpinterrupt allow the host to interrupt a processor on the main board. The interrupt receiver should have installed an interrupt handler. See the chapter on *Processor Board Programming* for more information.

alclrint	clear pending interrupts from the main processor
alinterrupt	interrupt the main processor
mpclrint	clear pending interrupts from the specified processor
mpinterrupt	interrupt the specified processor

Error Handling

The Slave Mode library provides some basic error message generation facilities which prioritize messages based on an error level. The functions below allow the Slave Mode program to set and get the current error level and to generate level sensitive error output.

algeterrorlevel	return the current error level
alseterrorlevel	set the current error level
errmsg	write an error message if error exceeds current error level

Slave Mode Programming

Windows 3.1 Management

MSDOS compatible releases also contain an MSWindows 3.1 Dynamically Linked Library (DLL) for use with Windows applications. All Slave Mode functions are supported. The functions below allow the Windows application to handle output to standard output and standard error that is generated from the processor board.

alputs	Windows 3.1: write a string to a window
W3Slv860Setup	Windows 3.1: specify a window as standard output and standard error

Slave Mode Programming Examples

In Slave Processor Mode, a program is loaded onto a processor board, and individual functions within the processor board program are called from a host program running on the host processor (Intel X86, Sparc, etc, ...).

The Slave Mode program makes calls to library functions to perform the following actions:

- Load processor board application
- Call library functions to determine processor board code and data addresses
- Optionally call processor board functions for dynamic memory allocation (malloc and free). Note, this may also be done by the processor board program.
- Optionally call library functions to load data into processor board memory. Data may also be transferred to processor board memory from an external data source.
- Invoke processor board functions function(s)
- Wait for completion of processor board function(s)
- Optionally call library functions to read data out of the processor board memory. Data may also be transferred out to an external device.
- Handle multiple processors and multiple processor boards

System call requests generated from the processor board application require the attention of the Slave Mode program. Processor board system calls are serviced by Slave Mode program function calls (**alwait**, **alwaitany**, **alintstat, mpwait, mpwaitany and mpintstat**). If the processor board program makes system calls, it will suspend execution until the request is serviced by the Slave Mode program. Thus it is necessary that the Slave Mode program call one of the above functions periodically (if not frequently) in order to poll for system requests.

It is quite possible to write a processor board program that does not perform any system call requests. In this case, the host may *not* need to call *any* Slave Mode function calls after having started the processor board program using the **alcall** (or **mpcall**) function.

Slave Mode Programming

Virtual and Physical Addresses

Processor board programs are run with the memory management unit (MMU) enabled. This provides for the assignment of program addressing of code and data (virtual addresses) to different hardware addresses (physical). The Slave Mode library requires the proper use of virtual and physical addresses. In general, any address that is to be used by the processor board application program should be a virtual address, and any address that is used to transfer data to and from the Slave Mode program must be a physical address.

The library function **aladdr** is used to look up the virtual addresses of global data and code variables. These addresses may always be converted from virtual to physical addresses by calling **VtoP**.

The **alcall** function's first parameter (**mpcall's** second) is the address of a function or subroutine entry point that the on-board processor is to execute. Since the processor board Kernel receives this address, it must be a virtual address.

The **alget** and **alset** family of functions are used to transfer data to and from the processor board. They must be called with physical addresses.

The Vector Library example code later in this section illustrates the correct use of virtual and physical addresses.

Dynamic Memory Allocation

Processor board programs may allocate memory either by defining data structures or by performing heap allocation system calls. Using Dynamic Memory Allocation techniques, arrays for data may be allocated and freed as needed, providing optimal utilization of available processor board memory. The heap management routines are implemented as processor board functions.

An alternative is to allow the host to perform processor board heap allocations in the Slave Mode program and pass addresses of allocated buffers to the processor board as function arguments. As a general rule, *any function that is callable on the processor board is also callable from a Slave Mode program.* The user is free to call heap management functions (**malloc** and **free)**, for instance, from Slave Mode. This technique is illustrated in the example below.

Slave Mode Example - Passing Arguments

The following short program loads the executable ***test*** to processor board memory and calls the function ***function()*** with two long arguments, 100 and 200. The function on the processor board returns a 32-bit integer argument which is retrieved using **algetiresult**.

Slave Mode Programming

```
#include <allib.h>
#define I860PROGRAM "test"
#define DEV         0
#define STACKSIZE   100000

main ()
{
      alopen (DEV);
      aldev(DEV);
      if (almapload (I860PROGRAM, STACKSIZE) != 0)
            errexit ("can't load %s\n", I860PROGRAM);

      /*--- call i860 function "_function" with two
             arguments, wait for completion, and fetch
             an integer result ---*/

      alcall (aladdr ("_function"), 2, 100L, 200L);
      alwait (DEV);
      printf ("result %ld\n", algetiresult ());
      alclose (DEV);
}
```

Slave Mode Example - Vector Library

The Vector Library is a DSP application library hand optimized for the Alacron processor board. It contains a variety of 1-D and 2-D floating point operations, one of which is the dot product function or **dotpr().** The program below shows how to call function **dotpr** from a Slave Mode program.

As part of the Alacron Vector Library, a precompiled processor board program is provided called **progxple**, for ISA and PCI systems (little endian), or **progxpbe,** for VME (big endian) systems. The executable files progxple and progxpbe have a copy of each vector library routine along with the dynamic allocation routines **malloc** and **free**. The following example also shows how data may be dynamically allocated and loaded. The function **fetch_some_data** is presumed to exist and provides a source of data to operate on.

The /usr/ft200 path name used in the example below is specific to the FT200-AT and FT200-V processor boards. If you are using a different processor board, substitute the correct full path name of progxple or progxpbe. If you have installed the vector library, the file should reside in the lib directory under the RT860 software directory tree.

Slave Mode Programming

```
#if defined(LE)
#define I860PROGRAM "/usr/ft200/lib/progxple"
#else
#define I860PROGRAM "/usr/ft200/lib/progxple"
#endif
#define STACKSIZE   100000
#define DEV         0
typedef unsigned long ADDR;
#define N                 1024
float buf[N];

ADDR i860malloc (n)
long n;
{
      alcall (aladdr ("_malloc"), 1, n);
      alwait ();
      return algetiresult ();
}

main ()
{
      ADDR A_buf1;
      ADDR A_buf2;
      ADDR A_result;
      ADDR A_dotpr;
      float result;

      if (alopen (DEV) != 0)
            errexit ("can't open device %d\n", DEV);

      if (almapload (I860PROGRAM, STACKSIZE) != 0)
            errexit ("can't load %s\n", I860PROGRAM);

      /*--- allocate data/arrays ---*/

      A_buf1 = i860malloc ((long)N*sizeof (float));
      A_buf2 = i860malloc ((long)N*sizeof (float));
      A_result = i860malloc ((long)sizeof (float));

      /*--- fetch some data ---*/

      fetch_some_data (buf, N);
      alsetla (VtoP (A_buf1), buf, N);
      fetch_some_data (buf, N);
      alsetla (VtoP (A_buf2), buf, N);

      /*--- call dot product ---*/

      A_dotpr = aladdr ("_dotpr");
      alcall (A_dotpr, 6, A_buf1, 1L, A_buf2, 1L, A_result,
            (long) N);
      alwait ();
      algetl (VtoP (A_result), &result, 1);
      }
```

Slave Mode Programming

Additional Examples

The RT860 software distribution provides a series of programming examples that further illustrate various ways in which Slave Processor Mode applications may be structured. For information on running these programs please refer to the section on *Building Your Application*.

3

Processor Board Programming

While Slave Mode offers a variety of functionalities, it's fundamental purpose is to facilitate the execution of an application on the processor board. This section discusses a variety of issues that relate to processor board programming.

Processor Board Tools

The processor board has it's own set of tools which are different from the host's. For example, a PC user might use Microsoft Visual C to build a program that runs on the Intel X86 processor. However, the code that runs on the processor board is built with tools that are designed especially for the Alacron board processor.

For example, the AL860 and FT200 series boards contain at least one Intel 860 processor. In order to build programs for these processor boards, the user must use an i860 assembler or an i860 C or FORTRAN compiler. Alacron uses the Portland Group tool set for this purpose. The PG tools product includes ANSI C and FORTRAN compilers, an assembler, a linker, a debugger and a profiler. The utilities usually run on the host system. In this case they are called *cross tools* because they run on one type of system, but generate code for another, in this case the i860.

The Portland Group compiler is straightforward to use. The following command compiles and links `main.c`, creating the Intel 860 executable `main`.

On an MSDOS system:

```
C:> pgcc main.c -o main
```

On a Unix system

```
# pgcc main.c -o main
```

There is no difference in how the command is entered between Unix and MSDOS. This is because the compiler has an identical command line interface across all platforms. Additionally, aside from the host interface logic that interacts with the host bus, the processor boards themselves do not change. The processor board software is more or less uniform across Alacron's entire product line.

Please refer to the Portland Group tools manuals for detailed information on how to use the compiler tool set.

Processor Board Programming

Processor Board Library

Alacron supplies a library of functions that provides architecture specific functionality. Standard ANSI C functions are supported by the compiler and Kernel without having to link in any special libraries. However, calls to non-ANSI functions require the application to be built with the processor board library. On i860 processor boards, this library is called i860lib.a. This library is often referred to as the *Runtime Library*.

The Runtime Library contains a set of functions which are specific to the Alacron processor board architecture. For example, the **rtcioctl** function manages the on-board Real Time clock. The function **atioctl** allows the processor board program to install host and peer processor interrupt handlers. Other functions provide memory management unit (MMU) support.

The functions **malloc** and **free** are standard ANSI functions which are implemented using the on-chip cache. Alacron has added the functions **umalloc** and **ufree** which are uncached. Alacron has also added functions to perform virtual memory management. The functions **vtop** performs virtual to physical address translation. The function **mapvtop** creates a user specified virtual to physical memory mapping. All of these additional functions are located in i860lib.a.

For information on building applications with the processor board Runtime Library, please refer to the section called *Building Your Application.* All Runtime Library functions are also covered in detail in the *RT860 Software Reference Manual.*

Multiple Processor Extensions

Multiple processor programs have requirements for additional functionality that support parallelism. The Runtime Library contains all of these Multiple Processor Extensions. A comprehensive overview of the multiprocessing on Alacron processor boards is provided in the section entitled *Multiple Processor Extensions*.

Application Libraries

The developer usually is concerned with achieving as much efficiency as possible with the available hardware. Alacron offers libraries of functions which are specialized according to application niches. All told, the libraries contain several hundred functions which are callable from standard C or FORTRAN. Each function has been hand optimized to perform as efficiently as possible on the Alacron architecture. Application libraries are specially written for different types of chips on the main board to take advantage of their low level architectural features. The libraries are as follows:

Real Time Image Processing Libraries (RIPL) - contains many integer operations on 8/16/32 bit data, convolutions, histograms, binarization, FFT's, filters, etc, ...

Vector Numerical Library(VLIB) - contains 1D and 2D 32-bit floating point operations, real and complex functions, FFT's, convolutions, matrix operations, mathematical functions, etc, ...

C Images Library - higher level image processing functions, object location, centroids, morphology, rendering, and much more.

Compression Operations - JPEG, CCIT G4

Once an application makes a call to a library function, the library must be added to the link line at build time. The example below links the VLIB with user application main.c on an MSDOS system:

```
C:> pgcc main.c -o main -Mquad -O4 -I$AL860\..\include vlibxple.a
i860lib.a -lm
```

The environment variable `AL860` should be defined to point to the RT860 software host bin directory. The `-I` option to the compiler specifies a directory, referenced off of symbol `$AL860`, to be searched for include files. The Vector Library itself is listed as `vlibxple.a`. In your build file you may need to specify a full path name for all libraries on the link line. The inclusion of `i860lib.a` references the Alacron processor board library which contains support for the application libraries and provides support for certain processor board features. The `-lm` option instructs the compiler to link the compiler's default math library which the vector library requires. The `-Mquad` option is used for data alignment. Option `-O4` turns on full compiler optimization.

The application libraries have literally hundreds of functions. Please refer to your RIPL, VLIB, or C Images manuals for more information.

Daughtercards

The Alacron architecture is unique in it's ability to combine compute modules and IO modules. The main board, which plugs into the host bus, has at least one processor which can both perform an administrative function and double as a compute engine. As processing needs increase, a second processor may be added to the main board. The main board may have a maximum of two processors, but additional processing modules may be added using the expansion connectors.

Each Alacron main board has two expansion connectors which accept multiple expansion cards or *daughtercards*. The daughtercards may contain additional processors and/or logic that connects to an external device or industry standard interface. Most daughtercards may be stacked in a single expansion connector providing the main board with access to several devices. The following is a subset of currently available daughtercards:

- Digital Interface (Digital Cameras)
- Video Display Adapter (VGA Displays)
- SCSI
- VSB
- Generic Digital Input/Output (Third Party Expansion)
- Data Translation DT Connect I and II interface
- High Resolution Graphics Card (Multisync Displays)

- Imaging Technologies VisionBus

The daughtercards are controlled by processors on the main board. Most of the daughtercards are equipped with high speed DMA controllers so the main processor is uninvolved in the movement of data between the external device and processor board memory.

Daughtercard Libraries

In order to control a daughtercard, the processor board application needs to link with the daughtercard library for each daughtercard that is being used. The daughtercard library is a set of functions that have been built with the processor board compiler.

Alacron daughtercard libraries follow a standard format. The library functionality is organized around a basic set of routines:

- open the device
- close the device
- read data (optional)
- write data (optional)
- Input Output Control

The open routine establishes a connection with the daughtercard device and places it in it's default mode. The close routine is called after the program is done using the daughtercard. The read and write routines are not always present. They receive and send data to the daughtercard. The Input Output Control function, ioctl, is used to perform device specific control operations. There are usually many different ioctl commands, each performing a specific device function.

There are different open, close, read, write and ioctl routines for each daughtercard. Usually the daughtercard name forms the prefix of the function. For example, on the DI daughtercard, the open routine is called **diopen**.

Please refer to the manual on your daughtercard for information on how to call daughtercard library functions. Most daughtercard software distributions contain example code which may be examined.

Debugging Processor Board Applications

Debugger

The best way to debug Stand-Alone programs is to use the symbolic debugger. On processor boards with an i860, this utility is called PGDBG. Please refer to the manuals on PGDBG for more information.

Print Statements

Aside from using a debugger, the application developer's most important debugging tools are the **printf** and **mp_printf** (multiple processor safe **printf**) system calls. These functions may be called from anywhere in a processor board application except from within an interrupt handler.

One of the problems with using these print functions is that host operating systems often buffer output to standard output. If you are using print statements as program trace points, for example, to determine how far an application progressed before crashing, it is important for a print statement to actually be output before continuing. The ANSI system `fflush` may be used, in this case, to flush all buffered output to the standard output device. The example below shows how `fflush` is used.

```
#include <stdio.h>

main()
{

printf (“Trace Point %d\n”,1);
fflush(stdout)

/* undebugged code here */


printf (“Trace Point %d\n”,2);
fflush(stdout)
```

If the program crashes in the undebugged section of code, the print of `Trace Point 1` will still appear because fflush flushes buffered output before the program continues. This indicates that the program progressed to at least that point. The absence of the print statement `Trace Point 2` would indicate that the program never progressed beyond the section of undebugged code.

If the `fflush` calls were omitted, the print statements would be buffered and printed to standard output by the host operating system after an indeterminate interval.

Memory Dumps

The use of a print statement in the processor board application requires a Slave Mode program to call a Slave Mode function that handles processor board system calls. In certain applications, this is impractical. For example, some Host Programs may be busy managing other devices.

It may make sense to try to debug a processor board program as a Stand-Alone program before mating it with a Slave Mode program. However, this is also not appropriate for all situations.

One frequently used technique is to reserve a shared buffer space in processor board memory where the Alacron board processor can write debugging information. The host may access this memory later using Slave Mode function calls to read memory. This is often how a processor board Kernel is debugged.

4

Multiple Processor Extensions

The processor board runtime library has been enhanced to provide support for multiple processor boards. The following section is an overview of these functions and shows how they work. Important issues such as cache coherency are discussed. Several programming examples show how to write code using the multiple processor extension functions.

Multiple Processor Extensions are not available for AL860 series processor boards and many are unsupported on all single processor FT200 boards. Multiple Processor Extension Functions are located in the processor board Runtime Library. See the "RT860 Software Reference Manual" for more information on individual functions.

Execution Model

The RT860 runtime software Multiple Processor Extensions provide a light-weight thread execution model for a tightly coupled shared memory multiprocessor hardware environment. In this model, all processors run single-tasked and share all text (code) and data. A single processor board program file is loaded on to the FT200.

When the processor board is reset, processor 0 (the main processor) begins execution, while all other processors on the selected FT200 board are placed into an "idle" state. Processor 0 uses thread control functions (**mp_thread_start**, **mp_thread_query**) to initiate program execution on other processors. The code executed by all processors is shared from the same text (executable code) segment. Each processor accesses the same data area, however each processor has a separate execution stack.

RT860 provides two sets of inter-processor communication and synchronization functions . The Semaphore functions (**mp_sem_allocate, mp_sem_free, mp_sem_acquire, mp_sem_release**) implement a simple binary semaphore, which may be used to provide mutually exclusive access to shared resources, as well as simple program synchronization.

The inter-processor interrupt functions **mp_catch_int** and **mp_gen_int** may be used to asynchronously interrupt or signal from one processor to another.

Support for cache mode control is provided through an enhanced version of the **mapvtop** library function. This function may be used to change the cache policy by modifying page table entries. The function mp_4meg_addr may be used to obtain an address with the desired cache mode.

Parallel Algorithms

This section describes some of the issues that are relevant when designing parallel algorithms for the FT200. The first area discussed is a brief overview of parallel algorithm models. It is by no means a complete discussion of the issues of parallel algorithm analysis; the reader is encouraged to explore the literature for more detailed analysis. The second area of discussion is that of memory contention issues as they pertain to the FT200.

A general model for parallel algorithms is to split up the computational work required to complete a given task. This is done usually in one of two ways; the workload is divided in time, or divided in space.

Division in time works as follows. Units of work arrive in time sequence (for example, images acquired from a video camera are captured, and processed sequentially). To exploit parallelism in this environment, one might simply assign each unit of incoming data to successive processors (the first frame goes to Processor 0 , the second to Processor 1, etc.). Using this model, the throughput of computation would tend to increase linearly with the number of processors available (some will argue that there is a theoretical basis to discount a true linear increase in performance; we don't disagree, but in the context of this discussion, our goal is to get a dual processor FT200 to run as close to twice as fast as possible). This approach tends to be simple and straightforward to implement, but has the drawback that it may not effectively address the issue of memory contention (which has a significant impact on total performance). In addition, while the throughput may increase dramatically, the response time (time from receipt of a unit of data to time results are available) does not improve.

Division in space works by taking a unit of data and dividing it up into 2 or more pieces, assigning each piece to a separate processor. In our video capture example, an image may be received, the top half processed by one processor and the bottom half processed by the second processor. Using this model, again the throughput of the computation increases linearly with the number of processors. Response time is also improved, since each unit of work is being worked on simultaneously by more than one processor. Division in space algorithms tend to be more difficult to code, since care must be taken dealing with cache coherency issues. And often times an algorithm doesn't simply partition, i.e. while much of the work may be allocated to separate processors, the final result may require a merging step that combines in some way the results of the separate computations.

Depending on the specific algorithm being implemented, some analysis is required to choose between dividing the algorithm in time or space.

Memory Contention

Regardless of the parallel computation model chosen, the problem of memory contention becomes an area requiring detailed analysis.

Multiple Processor Extensions

The FT200 has two 50 MHz i860 XP processors, sharing a common 200 MB/sec bandwidth memory. Access to the memory is serialized, that is, only one processor may access the memory at a time. However the i860 XP CPU has large on chip cache memories (16 KB for code, and 16 KB for data).

When developing MP applications for the FT200, one must analyze the algorithms being implemented, and structure them to take best advantage of the FT200 processor-memory architecture.

Given the serialized access to the memory, algorithms that are memory bandwidth limited will not benefit from multiprocessing. The issue then becomes how to structure an algorithm so that it is less memory intensive and more compute intensive.

Non-Memory Intensive Algorithms

Some algorithms are by their very nature more compute intensive. Examples are the FFT, matrix multiply, matrix inversion, convolutions, etc. In these cases, no special memory contention considerations are required in order to gain an acceptable improvement in performance. In other words, if the algorithm is compute limited, not I/O limited, then the programmer's attention would be better spent limiting the number of compute cycles in the algorithm and not on limiting the number of memory accesses.

Cache the data

A simple rule for developing parallel algorithms on the FT200 is to construct the algorithm so that as much data as possible is brought into the i860 XP internal cache. Once the data is in cache, it may be accessed without any external memory cycles being run. In many cases, a parallel algorithm may be designed to work on units of data that do fit into cache.

Strip-Mining

Strip-Mining is the approach of getting the data into cache as applied to vector data. The input vector data is copied into cache using a very fast memory to memory copy function that fully utilizes the available memory bandwidth. Vector results are written to cache, and copied out when complete.

On the FT200, the goal is to minimize memory contention. The best way to do this is to access memory only when necessary, and then when memory is required, use as quickly as possible. Strip-Mining as a rule does this well.

Using Strip-Mining will result in parallel execution where computation by one processor is overlapped with the fetching and storing of input and output data by the other processor,

Memory Intensive Algorithms

Even the most memory intensive algorithm may benefit slightly from parallel execution. For example, the dot product applied to uncached data is memory limited. The theoretical performance of 50 Mflops on a single i860 XP processor results from memory bandwidth limitations. In practice, 50 Mflops is not achievable given a single processor, as there is function entry and exit code, and loop wrapper code that may result in processor stalls. A typical implementation might achieve a 44 Mflop throughput on dot products. While a dual processor implementation can't do anything about the 50 Mflop memory bandwidth imposed limit, a small improvement of 15 to 20 percent might be achievable.

Tutorial

This remainder of this section is a tutorial explaining the functions and use of the Multiprocessor extensions to the FT200 Runtime Software. The reader is referred to the *RT860 Software Reference Manual* for a function by function specification of the processor board library functions.

The FT200 main board is a two processor system. The main processor is processor 0. The other processor is processor 1.

If you are using an Intel 860 processor board, please refer to the section *Intel 860 Primer* for an overview of caching on the Intel 860.

The programming examples below often refer to the directory \usr\ft200 or /usr/ft200 which is the default RT860 installation directory for the FT200-AT and FT200-V processor boards respectively. If you are using the FT200-PCI, the default software installation directory is \usr\alacron. In this case, substitute \usr\alacron wherever you see a reference to \usr\ft200 or /usr/ft200 throughout this manual.

Compiling/Linking/Running

Example code contained in this tutorial is compiled and linked using the following compiler commands:

For FT200-AT processor board on an MSDOS system.

```
pgcc -c -Mreentrant -I\usr\ft200\include mpex1.c
pgcc -o mpex1 mpex1.o \usr\ft200\lib\i860lib.a
```

For FT200-AT or FT200-V processor boards on a Unix system:

```
pgcc -c -Mreentrant -I/usr/ft200/include mpex1.c
pgcc -o mpex1 mpex1.o /usr/ft200/lib/i860lib.a
```

For FT200-PCI processor boards on an MSDOS system:

```
pgcc -c -Mreentrant -I\usr\alacron\include mpex1.c
pgcc -o mpex1 mpex1.o \usr\alacron\lib\i860lib.a
```

The **-Mreentrant** flag is required to specify that all "leaf" functions are to be reentrant (a leaf function is one that doesn't call any other functions). Without this flag, the C compiler allocates static data for auto variables and for saving registers. In this case, multiple processors attempting to execute the same leaf function would get in each others way. With this flag, as with non-leaf functions, the stack is used.

Multiple Processor Extensions

Except as noted, example code is executed using rt860 with the following command:

```
rt860 -T mpex1
```

The -T flag specifies to rt860 that the data and stack address spaces should be mapped as cached write-through. Using this flag alleviates problems arising from cache coherency issues. This comes at some expense in execution speed. However it is considerably easier for programming, and is assumed in much of the example code of this tutorial. Some examples are intended to be run without -T and use Write-Back caches. These are noted.

Where maximum speed is required, cache issues must be examined in detail when the program structure is being evaluated. This section describes some of the techniques available to developers for multiprocessing application development using a write-back cache policy for maximum speed.

Thread control

Many ANSI C standard Runtime library functions are implemented as part of the Alacron Runtime environment. As a general rule, C Runtime library functions are not reentrant. That is to say, it is not safe to call a C system call on multiple processors concurrently. The function mp_printf() has been implemented as a multiprocessor safe version of the standard function printf(). This function is present in the standard Alacron library i860lib.a.

Other functions, such as malloc(), calloc(), realloc() and free(), are implemented in two separate versions. One version is for single processor use while the other is safe for multiple processor use. The single processor version is more efficient and should be used in single processor environments. This version of the memory allocation functions is found in the default Runtime library i860lib.a.

The multiple processor safe version of the memory allocation functions is found in the libcr.a library which, by default, is located in \usr\<target>\lib or /usr/<target>/lib. If libcr.a is placed ahead of i860lib.a on the link line, then the multiple processor memory allocation routines will be linked in. Both the single and multiple processor versions of these functions are semantically equivalent.

The directory <target> is a generalized way to refer to the location of RT860 Runtime software.

The only difference is that before calling the multiple processor memory allocation routines, the application must perform a call to the imalloc() function. This call must be performed on processor 0, once and only once, before a thread is started on processor 1, and before any of the multiple processor memory allocation routines are called. See the function description of the imalloc() routine for more information.

Multiple Processor Extensions

Thread control

Overview

The Multiprocessor extensions to **RT860** and Slave Mode programming provide the means to run a separate execution thread on each processor of a multiprocessing Alacron board. The execution model used by **RT860** is a light-weight thread execution model assuming a tightly coupled shared memory multiprocessor hardware environment. A single thread executes on each physical processor.

Each processor executes from the same code, and accesses the same data. The TEXT, DATA and BSS sections of the program are shared between processors. Each execution thread does however have a separate stack. The processors are not required to execute the precise same code; when a thread is initiated, the code to be executed may be from any function entry point in the program as a whole.

At program start time, processor 0 begins execution, while all other processors are placed into an "idle" state. Processor 0 may then use the thread control functions (**mp_thread_start**, **mp_thread_query**) to initiate program execution on other processors.

Once started, all threads execute autonomously and concurrently.

Starting Threads

Threads are started (initially by Processor 0) by invoking the **mp_thread_start** function. The calling sequence for **mp_thread_start** is as follows:

```
int mp_thread_start (int proc, void (*func)(void), void *param);
```

When called, this function attempts to start a thread of execution on the processor specified by *proc*. If successful, the remote processor will begin executing the function whose address is given by *func*, called with an initial parameter of *param*. **mp_thread_start** will return the value 0 if successful, -1 on failure.

The call will fail for one of two reasons; if *func* is not a valid text address, or if processor *proc* is already executing a thread.

The argument parameter *param* is an arbitrary pointer that would generally be either a simple integer argument, or the address of a set of arguments. Some care is required when specifying *param* as an address such that cache coherency issues are addressed. Refer to the section on Cache Issues for further details.

Example 1 demonstrates the use of mp_thread_start().

```
/*-----------------------------------------------------------------
|
|            mpex1.c - demonstrate thread start and query
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex1.c
|                  pgcc -o mpex1 mpex1.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex1.c
|                  pgcc -o mpex1 mpex1.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex1.c
|                  pgcc -o mpex1 mpex1.o \usr\alacron\i860lib.a
|
|            Run:
|                  rt860 -T mpex1
|
|-----------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

#define P1   1

void remote_function (void *param);

main ()
{
      int res;

      /*--- start thread ---*/

      res = mp_thread_start (P1, remote_function, (void*) 2);
      mp_assert ((res == 0));

      /*--- wait for completion ---*/

      mp_thread_query (P1, 1);

      mp_printf ("done\n");

}

void remote_function (void *param)
{
      /*--- Processor 1: print something ---*/

      mp_printf ("processor %d received argument %d\n", mp_proc_id (),
            (int) param);
}
```

In this example, **mp_thread_start** is called to start execution of the function **remote_function** on Processor 1. A single integer argument is passed, in this case the value 2. The return value of **mp_thread_start** is examined to see that no error occurred. P0 then calls **mp_thread_query** to wait for the completion of P1 execution. Meanwhile, P1 begins execution of function **remote_function**, calls **mp_printf** to print the Processor's ID, and the integer argument.

Note that P0 was free to perform any functions of it's choice following the **mp_thread_start** call and before the call to **mp_thread_query**.

Querying Thread Status

Once a thread is started on a remote processor, the initiating processor may query the state of the remote processor. The function **mp_thread_query** is used to determine the status of the executing thread. The calling sequence for **mp_thread_query** is as follows:

```
int mp_thread_query (int proc, int wait);
```

The argument *proc* identifies the remote processor to be queried. If the argument *wait* is non zero, **mp_thread_query** will not return until the currently executing thread completes. If the argument *wait* is zero, **mp_thread_query** will return immediately with the processor status. If the returned value is -1, then an invalid processor argument was specified. If the return value is 0, the remote processor is idle, indicating that any previously executing threads have completed. If the return value is 1, then the processor is currently executing a thread. If the return value is 2, then the thread has completed successfully.

Mutual Exclusion and Synchronization

As in any multiprocessing environment, the need exists to control the access to program resources. In particular, access to all shared data must be performed in a mutually exclusive way. In general, access to read only data need not be exclusive, however any time data is being updated, a mutual-exclusion mechanism is required.

Mutual Exclusion

A classic example of the problem with mutual exclusion is shown in the following code.

```
/*-----------------------------------------------------------------
|
|             mpex2.c - demonstrate the need for mutex
|
|             Compile on an FT200-AT or FT200-V on a Unix system:
|                   pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex2.c
|                   pgcc -o mpex2 mpex2.o /usr/ft200/i860lib.a
|
|             Compile on an FT200-AT on an MSDOS system:
|                   pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex2.c
|                   pgcc -o mpex2 mpex2.o \usr\ft200\i860lib.a
|
|             Compile on an FT200-PCI on an MSDOS system:
|                   pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex2.c
|                   pgcc -o mpex2 mpex2.o \usr\alacron\i860lib.a
|
|             Run:
|                   rt860 -T mpex2
|
|---------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

#define P1 1

int add1 (int n);

int count = 0;

/*-----------------------------------------------------------------
|
|             incr - increment the global variable
|
|---------------------------------------------------------------*/

void incr ()
{
      count = add1 (count);
}


/*----------------------------------------------------------------
|
|             add1 - perform the integer increment function
|
|---------------------------------------------------------------*/

int add1 (int n)
{
      return n + 1;
}

/*-----------------------------------------------------------------
|
|             loop - function to invoke 'incr' a few times
|
|---------------------------------------------------------------*/

void loop (void *param)
{
        int i;
        int n = (int) param;

        for (i = 0; i < n; i ++)
                incr ();
}

main ()
{
        int res;

        /*--- start thread on P1 ---*/

        res = mp_thread_start (P1, loop, (void*) 100);
        mp_assert (res == 0);

        /*--- perform P0's share of work ---*/

        loop ((void*) 100);

        /*--- wait for P1 to complete ---*/

        mp_thread_query (P1, 1);

        /*--- display results ---*/

        mp_printf ("count %d\n", count);
}
```

The code in the above example should result in the value of count being 200 at the completion of execution. In a multiprocessing environment the value is indeterminate, though less than 200. This is due to the fact that one processor might fetch the value of **count** after the other processor has fetched it, but before it has been incremented and put back.

While this example might seem simplistic, it demonstrates a very real problem, that of the need to provide mutually exclusive access to shared data. The data might be the pointers in a queue or linked list data structure, or the character counts in a linear buffer.

The primitive operations provided by RT860 are the **mp_sem_acquire**, and **mp_sem_release** functions which act on binary semaphores. A binary semaphore is an integer variable, whose value may take on 0 or 1. When the semaphore value is 1, it is said to be available; when the semaphore value is 0 it is not available. When the function **mp_sem_acquire** is called, it returns when the semaphore becomes available, changing it to unavailable. If the semaphore is not available, implying that another processor has acquired it, the processor must wait until it becomes available, at which time it is marked unavailable and the function returns.

The function **mp_sem_release** is used to release a semaphore and return it's value to available, allowing other processors to acquire it.

The calling sequences for **mp_sem_acquire** and **mp_sem_release** is as follows:

```
int mp_sem_acquire (void *sem, int wait);
```

The routine will attempt to acquire the semaphore associated with *sem*. If *wait* is zero, **mp_sem_acquire** will not return until the semaphore is acquired. If *wait* is non-zero, **mp_sem_acquire** will loop through the acquisition code up to *wait* times to acquire the semaphore, return 1 if the semaphore was acquired, and will return 0 if the semaphore is unavailable.

```
int mp_sem_release (void *sem)
```

mp_sem_release will release a semaphore, making it available to be acquired by other processors.

Two other routines are required when using semaphores; the **mp_sem_allocate**, and **mp_sem_free** functions are used to create instances of semaphores.

```
void *mp_sem_allocate ()
```

The **mp_sem_allocate** function returns a semaphore descriptor to be used with **mp_sem_acquire** and **mp_sem_release**. The initial semaphore value is set to available. If not successful, **mp_sem_allocate** returns NULL.

```
void mp_sem_free (void **sem);
```

The **mp_sem_free** function deallocates the semaphore whose address is given by *sem*.

In the current implementation of **RT860**, there are a limited number of semaphores that may be allocated at one time (128 at the time of writing).

The next code example shows the use of binary semaphores to "fix" the concurrent update example of the previous section:

```
/*-----------------------------------------------------------------
|
|            mpex3.c - demonstrate proper use of mutex
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex3.c
|                  pgcc -o mpex3 mpex3.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex3.c
|                  pgcc -o mpex3 mpex3.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex3.c
|                  pgcc -o mpex3 mpex3.o \usr\alacron\i860lib.a
|
|            Run:
|                  rt860 -T mpex3
|
|----------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

#define P1 1

int add1 (int n);

int count = 0;
void *mutex;

/*----------------------------------------------------------------
|
|            incr - increment the global variable
|
|----------------------------------------------------------------*/

void incr ()
{
      mp_sem_acquire (mutex, 0);
      count = add1 (count);
      mp_sem_release (mutex);
}

/*-----------------------------------------------------------------
|
|            add1 - perform the integer increment function
|
|----------------------------------------------------------------*/

int add1 (int n)
{
      return n + 1;
}

/*-----------------------------------------------------------------
|
|            loop - function to invoke 'incr' a few times
|
|----------------------------------------------------------------*/
```

```
void loop (void *param)
{
      int i;
      int n = (int) param;

      for (i = 0; i < n; i ++)
            incr ();
}

main ()
{
      int res;

      /*--- create mutex semaphore ---*/

      mutex = mp_sem_allocate ();
      mp_assert (mutex != NULL);

      /*--- start thread on P1 ---*/

      res = mp_thread_start (P1, loop, (void*) 100);
      mp_assert (res == 0);

      /*--- perform P0's share of work ---*/

      loop ((void*) 100);

      /*--- wait for P1 to complete ---*/

      mp_thread_query (P1, 1);

      /*--- display results ---*/

      mp_printf ("count %d\n", count);
      mp_sem_free (&mutex);
}
```

In this example, the expected value (200) is produced, since in the function **incr**, where the variable **count** is modified, the semaphore **mutex** is acquired prior to performing the update. Using this technique, the fetch, increment, store of **count** is guaranteed to execute to completion before a second processor attempts to execute the same code.

C Runtime Library

As demonstrated, the requirement for a mutual exclusion primitive derives from the need to protect shared data. Unfortunately, many routines in the ANSI C Runtime Library access static data, and therefore will not function properly in a multiprocessing environment without a mutual-exclusion mechanism.

Code should make use of semaphores to permit the use of all C runtime library functions in a multiprocessor environment.

Synchronization

The binary semaphore may be used to sequence the execution of code amongst processors. The following example demonstrates this use:

```
/*----------------------------------------------------------------
|
|            mpex4.c - demonstrate co-routines using semaphore
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex4.c
|                  pgcc -o mpex4 mpex4.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex4.c
|                  pgcc -o mpex4 mpex4.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex4.c
|                  pgcc -o mpex4 mpex4.o \usr\alacron\i860lib.a

|            Run:
|                  rt860 -T mpex4
|
|---------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

#define P1 1

void *sem1;
void *sem2;
int count;

/*----------------------------------------------------------------
|
|            remote_function - P1's code
|
|---------------------------------------------------------------*/

void remote_function (void *param)
{
      for (;;)
      {
            /*--- release P0 ---*/

            mp_sem_release (sem2);

            /*--- wait to be released by P0 ---*/

            mp_sem_acquire (sem1, 0);

            /*--- print current value of count ---*/

            mp_printf ("count is %d\n", count);
      }
}

main ()
{
      int res;

      /*--- create semaphores ---*/

      sem1 = mp_sem_allocate ();
```

```
        sem2 = mp_sem_allocate ();

        mp_assert ((sem1 != NULL) && (sem2 != NULL));

        /*--- set them both to unavailable ---*/

        mp_sem_acquire (sem1, 0);
        mp_sem_acquire (sem2, 0);

        /*--- start thread on P1 ---*/

        res = mp_thread_start (P1, remote_function, (void*) 0);
        mp_assert (res == 0);


        /*--- execute sequenced execution by using semaphores ---*/

        for (;;)
        {
              /*--- wait to be released by P1 ---*/

              mp_sem_acquire (sem2, 0);


              /*--- increment count, terminate if done ---*/

              if (++count > 10)
                    break;

              /*--- release P1 ---*/

              mp_sem_release (sem1);
        }

}
```

This example demonstrates the classic *co-routines* model, where each processor "takes turns" executing. In this case, processor 0 increments **count**, "wakes up" processor 1 and "sleeps" on **sem2**. Meanwhile processor 1 is sleeping on **sem1**, when awakened, displays the value of **count**, wakes up processor 0 and sleeps on **sem1** again. The process repeats.

A similar structure may be used for implementing a "master-slave" model, where P1 waits on a semaphore for P0 to give instructions and release P1. P1 and P0 execute concurrently, P0 then waits on another semaphore for P1 to complete, whereupon P1 returns to an idle state. This sounds quite similar to the functionality provided by the **mp_thread_start**, and **mp_thread_query** functions, but may be used to implement a more elaborate scheme for interaction between the processors. As it turns out, **mp_thread_start** and **mp_thread_query** use semaphores from within the Kernel in exactly the manner described above.

Bit Counter

The FT200-AT board is equipped with a 32 bit down counter that is clocked at 1.0 MHz. This hardware feature gives the programmer the ability to make very fine grain measurements of code execution speed, accurate to 1.0 usec.

Multiple Processor Extensions

On boards other than the FT200-AT, the Real Time Clock functions shown below are implemented using RTC interrupts. The granularity of measurement is much less than 1 Mhz in these cases.

The 32 bit counter is shared with the RTC (real-time clock) functionality. The counter is loaded from a software setable preset value. The counter counts down, and when the value zero is reached, if so enabled, an interrupt to the i860 is generated. The counter is automatically reloaded from the preset value, and counting continues. The frequency of interrupts is determined by the preset value.

For the purposes of profiling, a preset value of all one's (0xFFFFFFFF) may be used, while leaving the interrupts disabled. At a 1 MHz rate, the counter will roll over every 71.58 minutes. For most uses, simply reading the counter before and after the code to be profiled should suffice. In those cases where the 71.58 minute roll over may cause ambiguous results, the RTC interrupt functions may be use to eliminate any ambiguity.

Two functions are required to access the 32 bit counter. The first is **rtcioctl**, which is used to set the counter preset and turn on the counter. The second is **mp_clock_val**, which returns an instantaneous 32 bit counter value.

The following example shows the use of the 32 bit counter to make a simple timing measurement.

```
/*-----------------------------------------------------------------
|
|            mpex6.c - demonstrate 32 bit counter
|
|            Compile:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex6.c
|                  pgcc -o mpex6 mpex6.o /usr/ft200/i860lib.a
|
|            Run:
|                  rt860 -T mpex6
|
|----------------------------------------------------------------*/


#include <stdio.h>
#include <i860lib.h>

#define N                 1000000

#define CLK_PERIOD  1.0e-6

void dotpr (float *a, float *b, int n, float *res);

main ()
{
      double tstart;
      double tend;
      double telapsed;
      float nops;
      float *p1;
      int i;
      float result;

      /*--- initialize 32 bit counter ---*/

      rtcioctl (RTC_SETFREQ, 0xFFFFFFFF);


      /*--- allocate and initialize an array of N floats ---*/

      p1 = (float*) malloc (N*sizeof (float));
      for (i = 0; i < N; i ++)
            p1[i] = ((float) (rand () & 0x7FFF)) / 32768.0;


      /*--- mark start time ---*/

      tstart = mp_clock_val ();

      /*--- execute benchmark ---*/

      dotpr (p1, p1, N, &result);

      /*--- mark stop time ---*/

      tend = mp_clock_val ();

      /*--- compute elapsed time ---*/

      telapsed = (tstart - tend) * CLK_PERIOD;
```

```
        /*--- display results ---*/

        mp_printf ("Elapsed: %.6f sec\n", telapsed);
        nops = N*2;
        mp_printf ("%.2f Mflops\n", (nops / telapsed) / 1.0e6);
}

void dotpr (float *a, float *b, int n, float *res)
{
        float acc = 0.0;
        int i;

        for (i = 0; i < n; i ++)
                acc += a[i]*b[i];
        *res = acc;
}
```

Note, the use of double precision floating point variables for storing the counter values. This is done as the mantissa of a single precision variable is not sufficient to hold the full 32 bit integer counter value. Note also that the counter does count down; delta's are computed by subtracting the ending count from the starting count.

If required, the number of times the counter has rolled over may be established by enabling RTC interrupts, and calling **rtcioctl** with the RTC_GETCOUNT command. The following code fragment demonstrates a means of measuring times in excess of 71.58 minutes:

```
...

rtcioctl (RTC_SETFREQ, 0xFFFFFFFF);
rtcioctl (RTC_ENABLE);

rtcioctl (RTC_GETCOUNT, &start_rollover_count);
start_count = mp_clock_val ();

...

rtcioctl (RTC_GETCOUNT, &stop_rollover_count);
stop_count = mp_clock_val ();

elapsed_time = (stop_rollover_count - start_rollover_count) *
71.582788 + (start_count - stop_count) * 1.0e-6;
```

MMU Page Size

If you are using an Intel 860 FT200 processor board, your code could be improved by using 4 MB paging instead of the default of 4 KB. Please refer to the section *Intel 860 Primer* for a discussion of 4 MB paging.

The FT200 Runtime software provides access to 4 MB pages through address aliasing. With address aliasing, the same physical address may be accessed at differing virtual addresses, each using a different cache and page size.

The library function **mp_4meg_addr** is defined as follows:

Multiple Processor Extensions

`void *mp_4meg_addr (void *address, unsigned long int prop);`

The **mp_4meg_addr** returning an address allowing a program to access data at *address* using 4 MB pages with the cache policy requested by *prop*. *prop* may be one of the following (defined in <i860lib.h>):

• MP_KADDR_WBC	Write-back cached
• MP_KADDR_WTC	Write-through cached
• MP_KADDR_UNC	Uncached

The following code example demonstrates the speed up a program may experience as a result of using 4 MB pages:

```
/*-----------------------------------------------------------------
|
|            mpex5.c - demonstrate 4 MB pages
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex5.c
|                  pgcc -o mpex5 mpex5.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex5.c
|                  pgcc -o mpex5 mpex5.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex5.c
|                  pgcc -o mpex5 mpex5.o \usr\alacron\i860lib.a
|
|            Run:
|                  rt860 -T mpex5
|
|---------------------------------------------------------------*/

#include <stdio.h>
#include <malloc.h>
#include <i860lib.h>

#define N    1024
float *p4K;
float *p4M;

void transpose (float *a, int n);

main ()
{
      int i;
      double tstart;
      double tend;
      double telapsed4K;
      double telapsed4M;
      float result;

      /*--- initialize timer ---*/

      rtcioctl (RTC_SETFREQ, 0xFFFFFFFF);

      /*--- allocate an N x N array of floats ---*/

      p4K = (float*) malloc (N*N*sizeof (float));
      mp_assert (p4K != NULL);

      /* compute the corresponding 4 Mb, Write-Back cached address */

      p4M = (float*) mp_4meg_addr (p4K, MP_KADDR_WBC);
      mp_assert (p4M != NULL);


      /* get start time, call transpose 10 times, using 4 Kb pages */

      tstart = mp_clock_val ();
      for (i = 0; i < 10; i ++)
            transpose (p4K, N);
```

```
        /*--- get stop time, print results ---*/

        tend = mp_clock_val ();
        telapsed4K = (tstart - tend) * 1.0e-6;
        mp_printf ("4 KB pages: Elapsed: %.6f sec\n", telapsed4K);


        /* get start time, call transpose 10 times, using 4 MB pages */

        tstart = mp_clock_val ();
        for (i = 0; i < 10; i ++)
                transpose (p4M, N);

        /*--- get stop time, print results ---*/

        tend = mp_clock_val ();
        telapsed4M = (tstart - tend) * 1.0e-6;
        mp_printf ("4 MB pages: Elapsed: %.6f sec\n", telapsed4M);

        /*--- compute percentage improvement ---*/

        mp_printf ("Improved by %.2f percent\n",
                ((telapsed4K - telapsed4M) / telapsed4K) * 100.0);
}


/*-----------------------------------------------------------------
|
|               transpose - transpose square matrix in place
|
|---------------------------------------------------------------*/

void transpose (float *a, int n)
{
        float *p1;
        float *p2;
        float tmp;
        int i;
        int j;

        for (i = 0; i < n; i ++)
        {
                p1 = p2 = a;
                for (j = i; j < n; j ++)
                {
                        tmp = *p1, *p1 = *p2, *p2 = tmp;
                        p1 ++;
                        p2 += n;
                }

                a = a + n + 1;
        }
}
```

In this example, a large square array, *p4K*, is allocated using **malloc**. An aliased 4 MB write-back paged address is computed by calling **mp_4meg_addr**. A simple benchmark is run, in this case the transposing of rows and columns of the array. Using the high resolution timer of the FT200 (see previous section), times are measured for the 4 KB page case, and for the 4 MB page case. In this example, an improvement in elapsed time of 35 percent is measured.

Inter-Processor Interrupts

Inter-Processor Interrupts are supported by the Runtime Software Multiprocessor Extensions. It is possible for any processor to generate an interrupt on any other processor for whatever application specific purposes are required.

Two functions are provided to accomplish this. The **mp_catch_int** function is used to register a function that is invoked when an interrupt is received by a particular processor. The **mp_gen_int** function is called to cause an interrupt to be generated on a processor.

The functions are described as follows:

```
void mp_catch_int (int proc, void (*fctn) (int));
```

mp_catch_int specifies that the function *fctn* be invoked by processor *proc* when an inter-processor interrupt is received by *proc*. Note, **mp_catch_int** may be called by any processor for the purpose of registering an interrupt handler for any other processor (including itself). The function *fctn* is called with a single argument, whose value specifies which processor generated the interrupt.

```
void mp_gen_int (int proc)
```

mp_gen_int causes an interrupt to be generated on processor *proc*.

The following example demonstrates the use of inter-processor interrupts.

```
/*-----------------------------------------------------------------
|
|            mpex7.c - demonstrate inter-processor interrupts
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex7.c
|                  pgcc -o mpex7 mpex7.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex7.c
|                  pgcc -o mpex7 mpex7.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex7.c
|                  pgcc -o mpex7 mpex7.o \usr\alacron\i860lib.a
|
|            Run:
|                  rt860 -T mpex7
|
|----------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

typedef struct {
      volatile int iflag;
      volatile int procid;
      } int_info_t;

volatile int_info_t info;

#define P0   0
#define P1   1

void *sem;

void remote_function (void *param);

main ()
{
      int res;

      /*--- create a semaphore, mark as unavailable ---*/

      sem = mp_sem_allocate ();
      mp_assert (sem != NULL);
      mp_sem_acquire (sem, 0);

      /*--- start thread on P1 ---*/

      res = mp_thread_start (P1, remote_function, (void*) 0);
      mp_assert (res == 0);

      /*--- wait to be released by P1 ---*/

      mp_sem_acquire (sem, 0);

      /*--- send interrupt to P1 ---*/

      mp_printf ("P0 sending interrupt\n");
      mp_gen_int (P1);
```

```
	mp_thread_query (P1, 1);
}


/*-----------------------------------------------------------------
|
|		remote_int_handler - P1's interrupt handler
|
|---------------------------------------------------------------*/

void remote_int_handler (int procid)
{
	info.iflag = 1;
	info.procid = procid;
}

/*-----------------------------------------------------------------
|
|		deref - derefence a pointer to an integer
|
|-----------------------------------------------------------------*/

int deref (int *pv)
{
	return *pv;
}

void remote_function (void *param)
{

	info.iflag = 0;
	info.procid = -1;

	/*--- setup  to catch interrupt ---*/

	mp_catch_int (P1, remote_int_handler);

	/*--- release P0 ---*/

	mp_sem_release (sem);

	/*--- wait on interrupted flag ---*/

	while (deref (&info.iflag) == 0)
		;

	/*--- print out result ---*/

	mp_printf ("P1 received interrupt from processor %d\n", info.procid);
}
```

In this example, P0 starts a thread on P1. P1 initializes a global data area, registers it's interrupt handler and waits on the value of a memory variable in the global data area.

Semaphore *sem1* is used to keep P0 from proceeding until P1 has completed it's initialization. When P1 has completed initialization, P0 generates an interrupt to

P1. P1's interrupt handler modifies the flag in global memory. P1 detects this flag change, output's a message and terminates.

Miscellaneous Functions

Several useful functions are provided in the Runtime Multiprocessor Extensions and are described below.

Retrieving Processor ID

mp_proc_id returns an integer value unique to the processor making the call. For example, **mp_proc_id** returns 0 for Processor 0, and 1 for Processor 1. *mp_proc_id* is useful when an application requires different code paths for each processor.

This example demonstrates the use of **mp_proc_id**.

```
/*-----------------------------------------------------------------
|
|            mpex8.c - demonstrate mp_proc_id
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex8.c
|                  pgcc -o mpex8 mpex8.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex8.c
|                  pgcc -o mpex8 mpex8.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex8.c
|                  pgcc -o mpex8 mpex8.o \usr\alacron\i860lib.a
|
|            Run:
|                  rt860 -T mpex8
|
|---------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

#define P0   0
#define P1   1

void remote_function (void *param);

main ()
{
      int res;

      /*--- start thread ---*/

      res = mp_thread_start (P1, remote_function, (void*) 0);
      mp_assert (res == 0);

      /*--- P0 call's same function ---*/

      remote_function ((void*) 0);

      /*--- wait for P1 to complete ---*/

      mp_thread_query (P1, 1);
}


void remote_function (void *param)
{
      /*--- display something clever ---*/

      mp_printf ("hello world from processor %d\n", mp_proc_id
());
}
```

Multiple Processor Extensions

mp_printf/mp_fprintf/mp_assert

As described in the section on semaphores and the need for mutual exclusion, many C runtime library functions may not be used without using semaphores. The functions **mp_printf** and **mp_fprintf** are replacement functions for **printf** and **fprintf** that may be used concurrently by all processors.

mp_assert is a replacement for **assert** and invokes **mp_fprintf** rather than **fprintf**.

Sharing Memory

Cache issues are quite relevant when developing processor board programs. This is particularly true for multiprocessor applications.

Simply stated, the issue is one of maintaining cache coherency. That is, each processor in a multiprocessor system must operate on data that reflects the "true" or "current" value of that data. With the i860 cache sub-system, it is possible for each i860 to have a copy of a particular data item in it's individual cache, each copy with a different value. In general this will prove to be problematic for most applications.

The section on Write-Through or Snooped Caching only applies to Intel 860 processor boards.

Write-Through Cached - Snooped

The FT200 provides a very simple mechanism for alleviating the problem. The i860 XP's may be configured to operate with the *write-through* cache policy, and support hardware of the FT200 implements *snooping*. When configured for write-through, all writes to cached locations are written to physical memory as well as the cached location. With snooping, all other processors observe the write to physical memory and if the physical address is currently cached, invalidates the appropriate cache line. Using this scheme, cache coherency is guaranteed between all processors.

There is however a downside to using this technique; the i860 processor will typically run faster when cache is configured for Write-Back, rather than Write-Through. With Write-Back, writes to cached addresses don't run any memory cycles to physical memory until the cache line is replaced (due to cache flushing, or replacement resulting from the addressing behavior of the program)

The rt860 command line argument -T is used to cause the data and stack addresses to be cached as Write-Through and Snooped.

Write-Back Cache

To obtain maximum performance out of the i860, the data must be configured for Write-Back caching. There are several programming techniques that are useful in dealing with the cache coherency issues.

They are all based on the premise that data is not shared in any general sense. A program will "partition" the data, and assign it to the set of available processors. Each processor will operate on it's piece of data, and only at the completion of the algorithm will it become necessary to deal with the cache coherency issues.

Partitioning Data

When partitioning data for multiprocessors, the intention is to break the data up into sections that when accessed (assuming a random pattern of reading and writing), the particular cache state of each processor has no effect on the outcome of any other processors.

The simplest way to ensure that this is the case is to observe that on the i860 XP the cache lines are 32 bytes in length. If data is partitioned on 32 byte boundaries, there is no way that any two processors would cache the same addresses and create problems.

There are several ways of creating data that is 32 byte aligned.

1. Use compiler directives to force variables to be 32 byte aligned. Unfortunately, at the time of writing, no compiler flags exist to do this (this will hopefully not be the to case for too long). It is possible using assembly language to declare data that is properly aligned.

For example, the C data declaration:

```
float var[1024];
```

may be written in assembler with:

```
             .data
             .align 32
_var:: .blkf  1024
```

2. Another technique is to declare more data than is required, and adjust the addresses to be 32 byte aligned.

```
float var[1028];
float *pvar;

pvar = (float*) align32 ((void*)var);

/*--- only use pvar from now on ---*/

void *align32 (void *p)
{
        while (((long) p) & 0x1f)
                (char*) p ++;
        return p;
}
```

3. Dynamic allocation using **malloc** is another way to force alignment to 32 bytes.

```
float *pvar;

pvar = (float*) align32 (malloc (1028*sizeof (float)));
```

4. The **umalloc** library function may be used. In this case the address returned is already 32 byte aligned, however it is (by definition) uncached. The **mp_4meg_addr** function is used to alias the address to a write-through cached address (which additionally carries the benefit of 4 MB page access). Note, the exact size required is requested.

```
        float *pvar;

        pvar = (float*) mp_4meg_addr (umalloc (1024*sizeof
(float)),                       MP_KADDR_WTC);
```

Making the Caches Coherent

At the completion of execution by a remote processor, it is often the case that the results of the computation are required by another processor. However, with Write-Back caching, some (or all) of the computation results may be resident in the remote processors cache. A mechanism is required to force the results out of the cache into physical memory so that it may be accessed by another processor.

Any of the following techniques may be used to update physical memory.

- Flush the cache - The RT860 library function **flush** may be invoked by the remote processor, causing all cached data to be written out to physical memory.
- Copy results - Copy results to uncached or Write-Through/Snooped memory. The application may designate memory for results that is different than the working storage required by the remote processor. Access to the results area will be configured for Write-Through cached (using **mp_4meg_addr**). Upon completion, the results are copied by the remote processor to the designated results area where it is accessible by other processors.

The following example demonstrates the use of a results area.

```
/*-----------------------------------------------------------------
|
|            mpex9.c - Demonstrate use of Write-Through
|            and Write-Back data regions.
|
|            Compile on an FT200-AT or FT200-V on a Unix system:
|                  pgcc -c -Mreentrant -O4 -I/usr/ft200/include mpex9.c
|                  pgcc -o mpex9 mpex9.o /usr/ft200/i860lib.a
|
|            Compile on an FT200-AT on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\ft200\include mpex9.c
|                  pgcc -o mpex9 mpex9.o \usr\ft200\i860lib.a
|
|            Compile on an FT200-PCI on an MSDOS system:
|                  pgcc -c -Mreentrant -O4 -I\usr\alacron\include mpex9.c
|                  pgcc -o mpex9 mpex9.o \usr\alacron\i860lib.a
|
|
|            Run:
|
|                  rt860 mpex9
|
|---------------------------------------------------------------*/

#include <stdio.h>
#include <i860lib.h>

#define N    102400
#define P1 1

/*--- Declare results data structure ---*/

typedef struct {
      float low;
      float high;
      int lowidx;
      int highidx;
      } result_t;

/*--- Function Proto's ---*/

void dostats (float *p, int n, result_t *pres);
void remote_function (void *param);
extern void *umalloc (int);

/*--- Data Declarations ---*/

float *pdata0;                           /* data set for P0 */
float *pdata1;                           /* data set for P1 */

result_t *pres0;                 /* results for P0 */
result_t *pres1;                 /* results for P1 */

void remote_function (void *param)
{
      dostats (pdata1, N, pres1);
}

main ()
{
      int res;
```

```
        int i;

        /*--- allocate results, 32 byte aligned, Write-Through ---*/

        pres0 = (result_t *) mp_4meg_addr (umalloc (sizeof (result_t)),
MP_KADDR_WTC);
        mp_assert (pres0 != NULL);

        pres1 = (result_t *) mp_4meg_addr (umalloc (sizeof (result_t)),
MP_KADDR_WTC);
        mp_assert (pres1 != NULL);

        /*--- allocate data, 32 byte aligned, Write-Back ---*/

        pdata0 = (float *) mp_4meg_addr (umalloc (N*sizeof (float)),
MP_KADDR_WBC);
        mp_assert (pdata0 != NULL);

        pdata1 = (float *) mp_4meg_addr (umalloc (N*sizeof (float)),
MP_KADDR_WBC);
        mp_assert (pdata1 != NULL);

        /*--- initialize data ---*/

        for (i = 0; i < N; i ++)
        {
                pdata0[i] = (float) (rand () & 0xffff);
                pdata1[i] = (float) (rand () & 0xffff);
        }

        /*--- flush so that P1 can see data ---*/

        flush ();

        /*--- start 'remote_function' on P1 ---*/

        res = mp_thread_start (P1, remote_function, (void*) 0);
        mp_assert (res == 0);


        /*--- P0 does it share of work ---*/

        dostats (pdata1, N, pres1);

        /*--- P0 waits for P1 to complete ---*/

        mp_thread_query (P1, 1);

        /*--- P0 prints all results ... since results are in write-through
                cached memory, no special action is required by P1 to make
                cache coherent ---*/

        mp_printf ("data 0: low %8f high %8f lowidx %6d highidx %6d\n",
                pres0->low, pres0->high, pres0->lowidx, pres0->highidx);

        mp_printf ("data 1: low %8f high %8f lowidx %6d highidx %6d\n",
                pres1->low, pres1->high, pres1->lowidx, pres1->highidx);

}

/*----------------------------------------------------------------------
```

```
|
|               dostats - accumulate statistics
|
|               Inputs:
|                     float *p            address of data
|                     int n               number of data elements
|                     result_t *pres      address to put results
|
|               dostats traverses it's input array,
|               determines the minimum and maximum
|               values and indexes in the array.
|
|---------------------------------------------------------------------*/

void dostats (float *p, int n, result_t *pres)
{
      int i;
      int lowidx = 0;
      int highidx = 0;
      float low = p[0];
      float high = p[0];

      for (i = 1; i < n; i ++)
      {
            if (p[i] < low)
            {
                  low = p[i];
                  lowidx = i;
                  continue;
            }
            if (p[i] > high)
            {
                  high = p[i];
                  highidx = i;
            }
      }

      /*--- write results ---*/

      pres->low = low;
      pres->high = high;
      pres->lowidx = lowidx;
      pres->highidx = highidx;
}
```

In this example, Write-Back data regions are allocated (pdata0 and pdata1), and Write-Through results areas are allocated (pres0 and pres1). The data areas are initialized by P0, the cache flushed and the function **dostats** is invoked on the data. The results are displayed by P0. Note that the state of P1's cache has no effect on the Write-Through results area.

The trade-offs between flushing the cache, and copying to a results area are essentially one of speed. If the results are greater than the cache size (16 KB), then flushing the cache will likely be faster than copying to a results area. However, if the size of results are significantly smaller than 16 KB, then copying to a results area will likely be faster.

Programming Hints

Cache coherency is THE major issue.

Run the program uncached, or write-through cached (rt860 -U or rt860 -T). If it works uncached, or write-through but fails with write-back, then there is likely a cache coherency problem. Be sure that cache lines are not split, and that the remote processors caches are flushed.

Don't be fooled by the C compiler on polled variables.

Different C compilers, and even the same C compiler with different optimization levels, seem to treat "volatile" in unpredictable ways. The following code fragment should work but doesn't always:

```
volatile int pollvar;

fctn ()
{
      pollvar = 1;
      while (pollvar)
             ;
}
```

It is recommended that code of the following form be used, which generally gets the desired results.

If you are using the Portland Group compiler, don't enable function inlining.

```
int pollvar;

fctn ()
{
      pollvar = 1;
      while (deref (&pollvar))
             ;
}

int deref (int *pv)
{
      return *pv;
}
```

5

Windows 3.1 Programming

This release contains files to be used in developing Windows 3 mode applications that utilize Alacron processor boards. The standard Slave Mode programming and DOS Driver Library are replaced with a single Windows DLL (Dynamic Link Library). Using this library, the application developer has full access to the Slave Mode functionality of one or more processor boards from within a Windows 3 application.

The user is referred to the previous section of this manual that discusses Slave Mode programming.

The advantages to using a DLL in this application are two-fold. First, since the driver library requires interrupts, the only way Windows 3 guarantees that hardware interrupts will work is by putting the interrupt service routine in a fixed, preloaded code segment in a DLL. Secondly, by separating the library code from the application code, the user will not need to re-link application code for updates, and the DLL may be accessed from small and medium model programs (unlike the non-Windows Slave Mode library that requires only large model applications).

When compiling a slave mode program that will use the DLL, use the -D compile flag to define the symbols for the C preprocessor: MSC, MSDOS and W3. See the section below for the location in the software tree that contains an example slave mode program using the Windows DLL.

Features

Single Library/DLL

The library/DLL W3SLV860.LIB and W3SLV860.DLL incorporate all the library functions provided the Slave Mode Library (SLV860.LIB) and the MSDOS Driver Library (CDOSP860.LIB).

Support for Debugging

The W3SLV860 DLL provides the function **alputs** that displays text on a user selected scrolling window. This provides debug output capability for the W3SLV860 DLL.

Mapping of stdio/stderr writes

Processor board program writes to file descriptors 1 and 2 (standard output and standard error) are processed by passing the desired output data to the **alputs** function. In this manner, **printf** calls made by the FT200 are routed to the **alputs** window.

Windows 3.1 Programming

Files

```
LIB\W3SLV860.LIB
BIN\W3SLV860.DLL
SRC\EXAMPLES\W3LIB
```

The file names in this section are assumed to be installed in the RT860 runtime software installation directory.

LIB\W3SLV860.LIB

This is the library (actually a Windows Import Library that is logically associated to the DLL) that the Windows 3 application links to in order to gain access to the Slave Mode and Driver library functions.

BIN\W3SLV860.DLL

This is the W3SLV860 DLL. It is loaded by Windows when ever an application requires access to a Slave Mode or Driver Library function. This library is installed in the BIN directory with the assumption that this directory is in the DOS search path (PATH). Windows looks in the directories that PATH specifies in order to find W3SLV860.DLL.

EXAMPLES\W3LIB

This directory contains sample source for a simple Windows Application that uses the Alacron board. The user is referred to the README file in this directory for specific information on the example.

New Functions

Refer to the RT860 Software Reference Manual for information on the **W3Slv860Setup** and **alputs** functions.

Using W3SLV860

Using the W3SLV860 library and DLL is quite straight forward. The following summarizes program changes required.

1. Call **W3Slv860Setup ()**

Following the creation of a main window or a debugging output window (at the programmers discretion), a call to **W3Slv860Setup ()** is required to configure the Slave Mode and Driver Library.

```
#include <stdlib.h>

#if _MSC_VER >= 700
W3Slv860Setup (hWndMain, _environ);
#else
W3Slv860Setup (hWndMain,  environ);
#endif
```

2. Call Slave Mode program functions as required.

The standard Slave Mode functions should be called in whatever manner the application requires. The following code fragment is common:

```
if (alopen (device) != SUCCESS)
{
      printf ("ERROR: can't open device %d\n", device);
      return;
}
aldev (device);
if (almapload (I860PROGRAM, STACKSIZE) != SUCCESS)
{
      printf ("ERROR: can't load %s\n", I860PROGRAM);
      return;
}
alcall (aladdr ("_main"), 0);
alwait ();
```

3. Linking Libraries

The Slave Mode import library must be included in the link specification, along with standard windows libraries. The following link specification is used in the accompanying example:

For MSC Version 6.0

```
demo.obj+
ctrl.obj+
pf.obj
demo.exe /w /align:16
demo.map/map/li/nod/noe
msdos\lib\w3slv860.lib+
\windev\lib\llibcew.lib+
\windev\lib\libw.lib
demo.def
```

For MSC Version 7.0

```
demo.obj+
ctrl.obj+
pf.obj
demo.exe /w /align:16
demo.map/map/li/nod/noe
msdos\lib\w3slv860.lib+
\windev\lib\oldnames.lib+
\windev\lib\llibcew.lib+
\windev\lib\libw.lib
demo.def
```

Be sure to provide the full path name of the msdos\lib\w3slv860.lib library when linking.

6

User Extensibility

This section describes the User Extensibility feature of **RT860**. Using this facility, a programmer may conveniently extend the system call interface available to the processor board application to access hardware and software that runs on the host processor. Using User Extensibility, processor board application might access a graphics library running on an Intel X86 host to manipulate a VGA display adapter, or perhaps access an interface library to a data acquisition board installed in the host computer.

ANSI system calls are each assigned a unique trap code. This code is used by the host to identify the specific system call that was generated on the processor board. When servicing system call requests, the host essentially uses a *switch* statement with a *case* for each trap code that is supported. If the trap code is not handled by the switch statement, then an appropriate error is generated.

The Runtime Library function call **usertrap** allows a processor board program to generate a system call with a user defined trap code. User Extensibility is the processor of adding entries to the switch statement so that the host program can handle any user defined request. In effect, this *extends* the system call handling capability of the Slave Mode library or the **rt860** utility. *This is the best way for the processor board to make a remote procedure call to the host.* **RT860** has no support for the installation of a user installed host interrupt handler which will handle processor board interrupts.

Using the User Extensibility facility of **RT860** requires that a new **rt860** be built incorporating application supplied code. Generally the programmer will modify the file **userstub.c** which contains the hooks to extend the system call interface. **userstub.c** is recompiled and relinked with the following libraries:

For UNIX Hosts:

$AL860/lib/rt860.a
$AL860/lib/slv860.a

For MSDOS Hosts:

lib\rt860.lib
lib\slv860.lib
lib\cdosp860.lib

User Extensibility

USAGE

For UNIX Hosts:

```
cc [flags] userfile.c $AL860/lib/rt860.a $AL860/lib/slv860.a
[libraries]
```

For MSDOS Hosts:

```
cl [flags] userfile.c LIB\RT860.LIB
$AL860\LIB\SLV860.LIB LIB\CDOSP860.LIB
```

```
void userinit (char *fname);
void usersys (int fctncode, long sysargs[]);
main (int argc, char *argv[], char *argp[]);
void setresult (long result);
```

DESCRIPTION

User Extensibility provides a means for extending the system call functionality of **RT860** with user specific functions. Under normal operation, a program running on the processor board will invoke a system function by executing a *trap* instruction with a function code and up to five 32 bit arguments. The trap allows the processor board to transfer control to the Kernel. The processor board Kernel receives and examines the function code and, when necessary, communicates the function code along with the arguments to the host program (**rt860**). The function code is examined by **rt860** and appropriate action is taken.

A processor board application may invoke a non-standard system call by choosing an unused trap code and passing it to the function usertrap. Function **usertrap** is a standard processor board (i860lib.a) function.

If the function code received by the host does not correspond to one of the standard runtime library functions, the function **usersys** is called with the function code and arguments. Function **usersys** is a user defined function which handles user defined system calls. This mechanism may be viewed as providing remote procedure call functionality. By providing application specific code in the function **usersys**, the developer can extend the functionality of **RT860**.

The user must provide two functions that are linked with routines in the **rt860.a** or **rt860.lib** library: **userinit** and **usersys**. **userinit** is called at program start time to parse command line arguments, specify details of the processor board executable file to be loaded, and specify command line arguments and environment list for the processor board program. The function **usersys** is called during program execution to execute user system functions.

The function **setresult** may be called from the user provided code to specify the 32 bit integer value that will be returned to **usertrap**.

The function **userinit** is called with a pointer to a structure of type *USER_INIT*, defined in **userinit.h**. The values of elements of the *USER_INIT* structure may be altered from their defaults in the function **userinit**. At the very least, the *u_i860fname* element should be set. The following summarizes the contents of **userinit.h**.

```
typedef struct {
      int   u_argc;                  /* arg count */
      char  **u_argv;                /* arg list */
      char  **u_argp;                /* environment list */
      char  *u_i860fname;            /* file name */
      int u_ctrlmask;          /* control flags */
      int u_rt860dev;          /* device */
      long  u_stacksize;       /* stack size */
      char  *u_kernel;         /* name of Kernel */
      int u_trapenable;        /* enable/disable traps */

/* The fields below are not supported on the AL860-AT */
      int u_rtcfreq;                 /* requested RTC freq */
      long  u_cache;                 /* caching for stack & data */
      int u_syscall_verbose;   /* system call traces */
      int u_kernel_verbose;    /* Kernel traces */
      int u_version_dump;      /* dump versions */
      int u_unmap_zero;        /* don't map page zero */
} USER_INIT;


#define User USER_INIT

#define INITKERNEL             (1 << 0)
#define LOADKERNEL             (1 << 1)
#define PRELOAD                (1 << 2)
#define RUNUNMAPPED            (1 << 3)
#define RUNMAPPED              (1 << 4)
#define PRTMAP                 (1 << 5)

/* The fields below are not supported on the AL860-AT */
#define ENABLE_FTE             (1 << 0)
#define ENABLE_TI              (1 << 1)
#define ENABLE_FZ              (1 << 2)
#define ENABLE_ALIGN           (1 << 8)
```

u_argc	command line argument count passed to the processor board program main. Default value is *argc* from invocation of host control program
u_argv	command argument vector, array of pointers to character strings, passed to processor board program main. Default is *argv* from invocation of host control program
u_argp	environment vector, array of pointers to character strings, passed to processor board program main. Default is *argp* from invocation of host control program
u_i860fname	name of processor board executable file to be loaded. If not found in the current directory, the directories specified in the environment variable **PATH** is searched to find the specified file for loading. Default is **a.out**. The name u_*i860fname* is maintained for historical reasons.
u_ctrlmask	specifies operation of control program. The following defined constants may be

	used in combination to specify the exact operation of the control program. By default, u_ctrlmask is set to INITKERNEL+RUNMAPPED. INITKERNEL — initialize processor board Kernel and prepare for execution LOADKERNEL — load i860 Kernel code and data PRELOAD — preload i860 executable file RUNMAPPED — run i860 file in mapped mode PRTMAP — print contents of preload table, and processor board Kernel variables
u_rt860dev	specifies the processor board unit to be loaded and run. Default is unit 0
u_stacksize	specifies the size of the stack allocation in bytes. Default is 65536 bytes.
u_kernel	specifies the file containing the processor board Kernel program. By default, lib/mpkernel is used for FT200 series boards. For AL860 series boards, the Kernel is slvkernel or rtkernel..
u_trapenable	specifies trap enable bits used by the i860 program. The following defined constants may be used together to specify trap enables. Default for *u_trapenable* is ENABLE_FTE | ENABLE_ALIGN | ENABLE_FZ: ENABLE_FTE — enable all floating point traps ENABLE_TI — enable traps on inexact FP results ENABLE_FZ — enable flush zero ENABLE_ALIGN — enable handling of data alignment traps
u_rtcfreq	The *u_rtcfreq* is used to enable and specify the Real Time Clock (RTC) functions of the processor board. With the default *u_rtcfreq* value of zero, all system calls requesting timing information (time, times) trap and are interpreted by the host program. By setting *u_rtcfreq* to a non-zero value, the RTC functions of the processor board are enabled, resulting in a periodic interrupt at the closest frequency available to that requested. System calls for time are then totally executed by the on-board processor. By using the processor board library functions for RTC access, a user function may be called on each RTC interrupt.
u_cache	Sets the caching characteristics for task data. Valid values include one of the following. The tasks' text are always Write-Back cached. 0: Write-Back caching enabled for data, bss and stack PTE_WT: Write-Through caching enabled for data bss and stack. *Do not use this flag on an AL860 processor board.* PTE_CD: caching is disabled for data bss and stack
u_syscall_verbose	If non-zero, this element will cause all system call invocations to the host to print out the request while handling the request
u_kernel_verbose	If non-zero, displays various Kernel configuration information, and more verbose output upon system calls

u_version_dump	If non-zero, Kernel and host library versions are displayed
u_unmap_zero	If non-zero, virtual addresses 0x00000000 through 0x00000fff are not mapped into the programs address space. Dereferencing null pointers will generate page faults. The default is to map in address zero.

The following is a template for an i860 and **usersys** routine for implementing a generic user extension:

Processor Board Code

```
/*

      file: i860.c

      Compiling Instructions:

            pgcc -o i860 i860.c /usr/<target>/lib/i860lib.a

*/

#define FCTN1             1000
#define FCTN2             1001

extern int usertrap (int, ...);
int fctn1 (int, char*);
int fctn2 (float*, float*);

i860program ()
{
      int i;
      float f1;
      float f2;
      int ret;

      ret = fctn1 (i, "testing 1 2 3");
      ret = fctn2 (&f1, &f2);
}

int fctn1 (int i, char* s)
{
      return usertrap (FCTN1, i, s);
}


int fctn2 (float* pf1, float* pf2)
{
      return usertrap (FCTN2, pf1, pf2);
}
```

Host Code

```
/*
      file: host.c

UNIX (SYSV386) Compiling Instructions:

cc -DUNIX -DSYSV386 -o host host.c sysv386/lib/rt860.a
             sysv386/lib/slv860.a

MSDOS (using Microsoft C) Compiling Instructions:

cl -DMSDOS -DMSC -AL -o host host.c MSDOS\lib\rt860.lib MSDOS\lib\slv860.lib
MSDOS\lib\cdosp860.lib
*/


#include <userinit.h>

userinit (USER_INIT *pinit)
{
      static char **argv[4] = {
             "these", "are", "my", "arguments"};

      pinit->u_argc = 4;
      pinit->u_argv = argv;
      pinit->u_i860fname = "i860";
      pinit->u_stacksize = 100000L;
      /* disable FP traps */
      pinit->u_trapenable = ENABLE_ALIGN;
}
```

```
#define FCTN1             1000
#define FCTN2             1001

void usersys (int fctncode, long sysargs[])
{
      char buf[100];
      float fv2;
      float fv2;
      long ivalue;

      switch (fctncode)
      {
      case FCTN1:
            algetba (buf, VtoP (sysargs[1], sizeof (buf));
            setresult ((long) fctn1 ((int) sysargs[0], buf));
            break;

      case FCTN2:
            algetba (&fv1, VtoP (sysargs[0]), sizeof (fv1));
            algetba (&fv2, VtoP (sysargs[1]), sizeof (fv2));
            setresult ((long) fctn2 (&fv1, &fv2));
            break;

      default:
            printf ("unimplemented function code: %d\n",
fctncode);
            break;
      }
}

int fctn1 (int i, char* s)
{
      printf ("fctn1 (%d, %s)\n", i, s);
      return 1;
}

int fctn2 (float *pf1, float *pf2)
{
      printf ("fctn2 (%.2f, %.2f)\n", *pf1, *pf2);
      return 2;
}
```

FILES

lib/rt860.a	UNIX rt860 extension library
lib/slv860.a	UNIX Slave Mode Library

LIB\RT860.LIB	DOS rt860 extension library
LIB\SLV860.LIBDOS	Slave Mode Library
LIB\CDOSP860.LIB	DOS Device Driver Interface Library

A

Building Your Application

There are many different host operating systems, compilers and environments covered in this manual. The simplest way to understand how a program is built is to study an example which matches your system and copy it.

RT860 Default Directory Tree Location

The Alacron RT860 runtime software is installed into a different default directory for each processor board.. The defaults for the various processor boards are shown below. Bear in mind that during the software installation process, the user is given the option of installing the RT860 software into a directory other than the default

Main Board	Default Directory Location for RT860
AL860-AT	\USR\AL860
AL860-V	/usr/ft200
AL860-PCI	\USR\ALACRON
FT200-AT	\USR\FT200
FT200-V	/usr/ft200
FT200-PCI	\USR\FT200

The directory delineators used above are \ for MSDOS style and / for Unix style. The processor boards are supported on a variety of Operating Systems so use the directory delineator that is appropriate for your environment. The default directories are the same across operating systems.

Building Your Application

Example Code

Alacron has provided example stand-alone processor board programs and Slave Mode programs for MSDOS using Watcom and Microsoft C, Windows 3.1, Unix and more. Please investigate the various examples in the src\examples (MSDOS style) or src/examples (Unix style) directory underneath your RT860 runtime software directory tree. Most of the example programs have build files which assume the use of Microsoft nmake or Unix make. In certain cases a batch file is provided. Please examine the appropriate build file to see how an application is built. The make files for each target have different file suffixes. Build files are show below:

make.sysv386	Unix 386 make file
make.sun5x	Solaris 2/x86 make file
make.c8	Microsoft C Version 8
make.c8w	Microsoft C Version 8, Windows
make.wc	Watcom C
make.sunos	SunOS/Solaris 1 make file
make.solar2	Solaris 2 make file
make.lynx	LynxOS make file
make.i860	i860 make files for use with Unix make
nmake.860	i860 make files for use with Microsoft NMAKE
DOSC8P.BAT	MSDOS batch file for MSVC rev. 1.5 (AL860/FT200-PCI only)
DOSWCP.BAT	MSDOS batch file for Wattcom rev. 10 (AL860/FT200-PCI only)

Slave Mode Programs

Slave mode programs, that execute on the host (**not** the processor board program), should be compiled using the following compile switches. These defines tell the C language preprocessor information about the particular environment that you are using. This enables the same include files to support many different hosts and configurations. Slave Mode programs generally only need to include the file allib.h which may be found in the include directory underneath the Alacron RT860 software tree.

MSDOS using Microsoft C Version 8	-DMSDOS -DMSC
MSDOS using Watcom C	-DMSDOS -DWATCOM
MSDOS using Microsoft C, Windows	-DMSDOS -DMSC -DW3
SYSV386	-DUNIX -DSYSV386
Solaris 2/x86	-DUNIX -DSUN5X
SunOS/Solaris 1	-Dsunos
Solaris 2	-Dsolar2

OS/2	-DOS2
LynxOS	-DLYNX

Slave mode programs should link with the appropriate libraries. Slave mode libraries are contained in the directories shown below. Note that each host operating system has its own directory so that library files with the same name may be kept distinct.

Host Operating System	**Directory of Host Libraries**
Unix System V/386	`sysv386/lib`
Solaris 2, Intel X86	`sun5x/lib`
OS/2	`OS2\lib`
LynxOS	`lynx/lib`
SunOS/Solaris 1	`sunos/lib`
Solaris 2, Sparc	`solar2/lib`
MSWindows 3.1	`MSDOS\lib`
MSDOS	`MSDOS\lib`

MSDOS and OS/2 Slave Mode applications should link with libraries **CDOSP860.LIB** and **SLV860.LIB**. MSWidows 3.1 Slave Mode programs should be linked with **W3SLV860.LIB**. Note that the DLL, **W3SLV860.DLL**, must be referenced in the PATH environment variable. Slave Mode programs running under Unix and all other operating systems should link with file **slv860.a**.

Please refer to your host compiler documentation for information on how to compile a program for your host computer.

Note: MSDOS Slave Mode programs must use large model.

Processor Board Programs

Programs that run on the processor board must be built using the processor board compiler. For Intel 860 main boards, this is the Portland Group C or FORTRAN compiler. A C program is compiled as follows:

```
pgcc main.c -o main
```

If your program uses any of the processor board support functions, then you will need to link in the library i860lib.a. Library *i860lib.a* should always appear last on the link line. It is located in the lib directory which is assumed to be under the RT860 directory tree. The path name of the RT860 tree is specified as \usr\<target> in the examples below. The `-I` compiler flag specifies the directory where the compiler will find the processor board Runtime Library include file, `i860lib.h` and any application library or daughtercard library include files.

On a Unix System:

```
pgcc main.c -I/usr<target>/include -o main /usr/<target>/lib/i860lib.a
```

On an MSDOS or OS/2 System:

```
pgcc main.c -I\usr\<target>\include -o main \usr\<target>\lib\i860lib.a
```

To link other processor board libraries, such as VLIB, RIPL, SIPL, or the daughtercard libraries, please see the appropriate manual for build instructions.

B

Intel 860 Primer

This section offers some basic information on the Intel 860 chip. If you are using an i860 Alacron processor board, a familiarity with the i860 architecture will greatly facilitate the process of application development. This section is by no means a comprehensive reference. For a detailed description of the i860, it's instruction set, registers, and architecture, please refer to the Intel Corporation's *i860Ô Microprocessor Family Programmer's Reference Manual.*

The Intel 860 microprocessor architecture was first implemented in the i860 XR chip. The central processing unit (CPU) contains integer, floating-point, graphics, memory management, and cache management functionality. The next generation is the i860 XP which is upwardly compatible, and in addition contains feature enhancements.

The Alacron AL860 processor board contains an XR chip, while the XP chip is on the Alacron FT200 processor board. Below is a brief list of distinctions between the XP and the XR:

- The i860 XR has a 4 KB code cache and an 8 KB data cache. The XP has a 16 KB code cache and a 16 KB data cache.
- The AL860 contains a single XR chip at 25 or 40 Mhz.
- The FT200 contains one or two XP chips running at 50 Mhz.
- The XP has additional instructions, such as the quad-pipelined load (128 bits), which enhance performance.
- Both chips have page level cache control support for write back and uncached memory accesses. The XP chip has multiprocessor cache support via it's snooped/write-through feature. Additional information on caching is found below.
- Both chips support 4 KB pages. The XP chip supports 4 MB pages. Additional information on 4 MB paging is found below.
- The XR has a 2-way set associative cache while the XP is 4-way.

Basic Architecture

The i860 has a 64-bit external data bus, a 128-bit internal data bus and a 64-bit internal instruction bus. Floating point and graphics programs are able to use these three buses simultaneously.

There are two classes of instructions: core instructions and floating point/graphics instructions. Instructions are executed on the core integer, floating point and graphics units respectively.

The integer core processing unit uses reduced instruction set (RISC) technology. Each instruction requires one CPU clock. The integer core executes integer instructions and operating system code. The integer unit performs memory loads and stores, transfers between registers, integer arithmetic, register shifts, logical operations, controlled branches, control register manipulation, cache mode configuration, and interrupt management.

The floating point unit contains a separate adder and multiplier. The adder performs the following floating point operations: addition, subtraction, comparison and conversions. The multiplier is able to execute floating point and integer multiplies as well as perform floating point reciprocal operations. Both 32 and 64-bit IEEE floating point formats are supported as native data types.

The graphics unit supports 64-bit integer logic that may be employed in 3-D graphics drawing algorithms. The unit has native pixel types of 8, 16 and 32 bits. Red, green and blue color values may be computed within a pixel.

The CPU contains support for the on-chip data cache and a number of parallel processing techniques. Among other features, the chip supports simultaneous integer and floating point operations, parallel multiplier and adder units (an add and a multiply can execute in one instruction), automatic increment on register indexes, and pipelined floating point operations.

The i860 register set consists of 32 general purpose registers, each 32 bits wide and 32 floating point registers. The floating point registers may be optionally accessed as 32, 64 or 128 bit registers.

In dual instruction mode, the processor can execute one instruction from each class, i.e. integer core and floating point. Using dual instruction mode and dual operation instructions in the floating point unit, the processor can execute three instructions *in a single cycle*.

The i860's address space is a linear 32-bit space that addresses 4 GB of memory. The on-chip memory management unit (MMU) provides page level cache mode and virtual memory management. The MMU page tables, a database of page level MMU information, is memory resident at an address stored by software in an on-chip register.

Cache Issues

The cache is a memory buffer, internal to the chip, where code and data are moved before being accessed by the processor. Data is cached in units of *lines* which are 32-bytes in size. When the processor references any data in a line that is not in cache, the whole line is cached. This occurs even if the code is referencing a single byte within a line. Cache lines are aligned on 32-byte address boundaries.

The Intel 860 processor provides the means to control the caching policy of data and text memory on a per page basis. As implemented in RT860, support is provided for all caching policies. The XR processor supports write-back and uncached modes. The XP supports these as well as write-back mode. The caching policies are explained below:

Uncached	When memory is uncached, the i860 caches are never used. Uncached memory never poses any cache coherency problems, between multiple i860's or the Host
Write-Through or Snooped	With Write-Through/Snooped, writes by each i860 update the cache, and update physical memory. On the FT200, snooping is implemented, so that if one i860 writes to a location that is currently cached by another, the hardware detects this and invalidates the cache row of the second processor. In this way cache coherency is maintained between all i860's. This method is significantly faster than uncached. It is also attractive for code that "polls" on fixed memory locations, since memory bus bandwidth is not consumed during the polling (physical memory is only accessed following the cache line invalidation caused by snooping). Unfortunately, memory coherency is not maintained between i860's and the Host. Using uncached memory, or flushing the cache is required in this case.
Write-Back	In Write-Back, data written by the i860 to currently cached locations is only written to physical memory on cache row replacement, or as the result of an explicit flush. Write-Back provides the fastest program execution but requires special attention be paid to cache coherency issues

Support is provided for all cache policies. Text (code) is always loaded as write-back (although as text is generally read-only the cache policy is irrelevant). Data/stack/bss/heap may be initially set to Write-Through-Snooped, or Write-Back.

Further support is provided through an enhanced version of the **mapvtop** library function. This function may be used to change the cache policy by modifying page table entries.

And lastly, RT860 implements virtual address aliasing, whereby the same physical addresses may be accessed at different virtual addresses. Three such regions are mapped into the i860 address space, one each for Uncached, Write-Through/Snooped, and Write-Back. Using library functions, the an i860 application may convert any address to one whose access will result in the application of the desired cache policy.

Four Megabyte Pages

The i860 XP processor's MMU provides both 4 KB and 4 MB page sizes. 4 MB pages are desirable as they reduce the frequency of TLB (translation lookaside buffer) cache misses, and improve (often significantly) program execution time.

The i860 XP processor has a feature called 4 MB pages. By default, all memory is mapped (using the Memory Management Unit of the processor) in 4 KB page units. The Translation Lookaside Buffer (TLB) is a small cache that holds second level page table entries, each of which describes a page-size page. As a program executes, the entries in the TLB are constantly being updated with those required to access the data or code being addressed. As an example, when accessing a 2 MB array of data, as many as 512 TLB entries will be required. The TLB size of the I860 XP is 16 entry, 4-way interleaved, or 256 entries total. Clearly, the TLB entries will be updated continually during access of the 2 MB array.

This pattern of behavior is referred to as TLB cache thrashing. However, by using 4 MB pages, the 2 MB array can be accessed with a single TLB entry, eliminating entirely any TLB cache thrashing.

The effect on performance can be quite substantial, as much as 20 to 30 percent increase in throughput has been measured on applications such as video imaging, graphics frame buffer manipulation and large FFT's.

The FT200 Runtime software provides access to 4 MB pages through address aliasing. With address aliasing, the same physical address may be accessed at differing virtual addresses, each using a different cache policy and page size.

C

Bus Mastership

ISA

The AL860-AT and FT200-AT processor boards are not able to master the ISA and initiate reads or writes to/from host DRAM. All transfers between processor board memory and host DRAM are implemented using programmed I/O via the host CPU. The AL860-AT and FT200-AT registers in I/O space allow the host CPU to read or write any physical memory location in the processor board memory space.

PCI

PCI processor boards may access addresses on the PCI bus using programmed I/O or by initiating transfers on the auxiliary DMA processor. The DMA library functions are documented in the *RT860 Software Reference Manual.* Please refer to those functions for more information.

VME

The FT200-V is able to perform 8, 16 or 32-bit programmed I/O. VME master accesses to nearly any A32 address, and 16 or 32-bit accesses to any A16 address. VME memory is mapped into the processor board physical address space as shown in the table below:

Processor Board Physical Addresses		VME Addresses		Description
Start Address	End Address	Start Address	End Address	
0x18000000	0x1EFFFFFF			Mapped VME Space
0x20000000	0xFEFFFFFF	0x2000000	0xFEFFFFFF	VME A32 Space
0xFFFE0000	0xFFFEFFFF	0x0000	0xFFFF	VME A16/D32 Space
0xFFFF0000	0xFFFFFFFF	0x0000	0xFFFF	VME A16/D16 Space

Bus Mastership

A processor board program may write or read any A16 address either in D16 or D32 mode by using the appropriate mapped window in processor board space. The top 16 MB of A32 space is not accessible by the main processor. The low 512 MB of A32 space are accessed in 128 MB blocks by setting up a mapping between processor board address 0x18000000 and the desired VME address block. The rest of A32 space is accessed starting at processor board physical address 0x20000000.

By default, the main processor executes with virtual memory management enabled. This means that the processor board physical addresses shown in the table must be mapped to main processor virtual addresses before a program can write to them. This must be done using the function mapvtop() which is documented in the *RT860 Software Reference Manual*.

Select an unused virtual region of the desired size and map it to the appropriate physical region shown above. Once the mapping is in place, the main processor can perform standard 8, 16 and 32-bit memory accesses straight to VME memory.

For example, the following code fragment sets up a 1 GB mapping at virtual address 0x20000000. For simplicity, the region will be mapped to VME address 0x20000000 which starts at processor board physical address 0x20000000.

```
/* map virtual to physical; 256 4 MB pages; uncached, writable */
mapvtop (0x20000000, 0x20000000, 256, 0x92);

unsigned long *addr = 0x20000000;
*addr = 0xF00DCAFE;
```

This code sets up a mapping between main processor virtual address 0x20000000 and VME address 0x20000000. The size of the mapping is 1 GB. The assignment statement to the variable addr, allows the main processor to perform a 32-bit write directly to the VME bus.

The user must be careful not to try to perform 64-bit or 128-bit accesses to the VME bus. This can be done inadvertently. For example, assuming the mapping above:

```
double *p = (double *) 0x20001000;
p = (double) 2.0;
```

Since a double is 8 bytes, this assignment is not legal. Also, you may not do the following:

```
char *p = (char *) 0x20002000;
memcpy(source_buffer, p, size);
```

The memcpy function on the processor board will use 64-bit or 128-bit transfers whenever possible. On the i860, if a buffer is misaligned, it may do smaller transfers until it gets to a double or quad boundary where it is able to use more efficient instructions such as fst.d. The memcpy function was intended to be used to copy memory as fast as possible from one processor board resident buffer to another.

D

Virtual Address Space

Overview

It is frequently the case (particularly in the VME environment), that the processor board application requires access to physical memory resources that are not available to the program given the default memory management unit mapping. The function **mapvtop()** is used to perform a mapping of virtual to physical addresses. A typical example is the requirement to map a VME address into the on-board processor program address space. While the required physical address is known (generally fixed in hardware), an available virtual address must be found.

The **prtmap** program is available to help evaluate the virtual to physical mapping of the i860 processor. Following the execution of a program (using **rt860** or a slave mode application), invoking **prtmap** with the -C and -D options will display the application mapping. From this listing one may ascertain what address regions are in use.

FT200-AT/AL860-V/FT200-V

The following address regions are ***not*** available for user applications

Virtual Address	**Size**	**Used for**
0x00000000	272 Mbyte	i860 program data
0x60000000	256 Mbyte	i860 processor B stack
0x70000000	256 Mbyte	i860 processor A stack
0x90000000	128 Mbyte	VSB BUSRAM mapping
0xC0000000	256 Mbyte	write back cached DRAM address aliasing
0xD0000000	256 Mbyte	write through cached DRAM address aliasing
0xE0000000	256 Mbyte	uncached DRAM address aliasing

Virtual Address Space

0xF0000000	256 Mbyte	i860 program text, hardware register mapping

The following virtual addresses *are* available:

Virtual Address	**Size**
0x20000000	1 GB
0x80000000	256 MB
0xA0000000	512 MB

FT200-V Memory Map

A task's data, bss and heap are mapped starting at virtual address 0000:1000 up until the end of available memory. In physical address space, task memory ends at the start of the stack.

Processor 0's stack starts at virtual address (8000:0000 - stacksize) through 7FFF:FFFF. The default stack is 64 Kilobytes. In this case, sixteen 4 KB pages are allocated starting at virtual address 7FFF:0000.

Processor 1's stack starts at virtual address (7000:0000 - stacksize) through 6FFF:FFFF. The default stack is 64 KB. In this case, sixteen 4 KB pages are allocated starting at virtual address 6FFF:0000.

User text (executable code) is stored starting at virtual address F040:0000. The number of 4 KB virtual pages allocated is determined by the size of the user program.

The table below shows virtual and physical address ranges for the FT200-V. Addresses whose location is not fixed are marked as ????:????.

Virtual Address Space

FT200-V Memory Usage Table

Virtual	Physical	Description
0000:0000 - 0000:0FFF	????:???? - ????:????	writes to null pointers generate a page fault
0000:0FFF - ????:????	????:???? - ????:????	allocated per processor
????:???? - ????:????	????:???? - ????:????	page allocated for semaphores[1]
????:???? - ????:????	????:???? - ????:????	Kernel entry point[2]
B000:0000 - C0FF:FFFF	0000:0000 - 10FF:FFFF	272 MB memory space mapped in Write-Back Cache mode
C100:0000 - D1FF:FFFF	0000:0000 - 10FF:FFFF	272 MB memory space mapped in Write-Through Cache mode
D200:0000 - E2FF:FFFF	0000:0000 - 10FF:FFFF	272 MB memory space mapped Uncached mode
FFF4:0000 - FFF4:0FFF	FFF4:0000 - FFF4:0FFF	Flush area for cache flushing
FFFD:0000 - FFFD:0FFF	FFFC:0000 - FFFC:0FFF	VIC addressing
FFFD:1000 - FFFD:3FFF	FFFD:0000 - FFFD:3FFF	VAC addressing

[1] Mapped one-to-one with the physical address of an allocated page. For example, on a 16 MB FT200-V, one four kilobyte page is mapped at the address 00FE.9000.

[2] The address is based upon where the kernel is located in physical memory space. One four kilobyte page.

Virtual Address Space

Virtual	Physical	Description
FFFD:4000 - FFFD:4FFF	1200:0000 - 1200:0FFF	Control/Status port addressing
FFFD:5000 - FFFD:5FFF	1218:0000 - 1218:0FFF	Slave Select 0 Address Register
FFFD:6000 - FFFD:6FFF	121C:0000 - 121C:0FFF	Slave Select 1 Address Register
FFFD:8000 - FFFE:7FFF	????:???? - ????:????	Kernel text (executable code)
FFFF:3000 - FFFF:3FFF	1100:0000 - 1100:0FFF	EXT-1 IO address area
FFFF:4000 - FFFF:4FFF	1180:0000 - 1180:0FFF	EXT-2 IO address area
FFFF:5000 - FFFF:5FFF	FFFA:0000 - FFFA:0FFF	IOSEL5 region, used in interrupt acknowledgment cycles
FFFF:6000 - FFFF:6FFF	1210:0000 - 1210:0FFF	Slave Select 0 Mask Register
FFFF:7000 - FFFF:7FFF	1214:0000 - 1214:0FFF	Slave Select 1 Mask Register
FFFF:8000 - FFFF:EFFF	????:???? - ????:????	Kernel Data/BSS/Stack
FFFF:F000 - FFFF:FFFF	????:???? - ????:????	Kernel trap entry point

E

Getting Help

Troubleshooting

If you have turned to this section, you are probably in the middle of application development and are seeing unexpected behavior in hardware and/or software. Alacron offers a variety of technical support services that are designed to assist our user community, but before you call us, there are a few simple things that you should check.

Check the release notes for any relevant explanations. All known bugs and potential problems have been noted for you.

We recommend a systematic approach to help pinpoint your problem. You first want to try to determine if the problem is related to the Alacron boards that you are using or not. Try running the diagnostics and a short “hello world” program. You want to be sure that your processor board is working properly and that the RT860 software is installed and configured correctly.

If you do have a hardware problem, you may return your board(s) directly to Alacron for servicing. See the section below for information on contacting Alacron Technical Support.

If you believe that your problem is in software, check the reference section in this manual and the programming examples supplied with the RT860 software to insure that you are calling the processor board libraries correctly. Check the return status from functions and any input arguments. Simplify the program as much as possible to isolate the failing condition. You can do this by removing any extraneous code that doesn’t directly contribute to the failure (don’t forget to save your original work).

Technical Support

Alacron offers technical support to any licensed user during normal business hours (EST). We will offer assistance on all aspects of board installation and operation. If you wish to speak with a Technical Support Representative on the telephone, please call the number shown below and ask for *Technical Support*. In situations which involve a great deal of detail, it may be more convenient for you to contact us by fax or electronic mail.

If you have a relatively short piece of code that exhibits a problem, fax it to us and our Technical Support Representative will type it in and try to reproduce your error. We can serve you faster if you send us sample code over the Internet. Sample code is easiest to review if any extraneous operations that aren't directly related to the problem are deleted. Note: we may not be able to run your code if we cannot replicate your hardware environment.

Alacron, Inc.
71 Spitbrook Road, Suite 204
Nashua, NH 03060 USA

telephone:	(603)891-2750
fax:	(603)891-2745

electronic mail:
sales@alacron.com
support@alacron.com

Before You Call

Please help us serve you better by having the following information ready:

- The serial numbers and hardware revisions of all of your boards. This information is written on the invoice that is shipped with our products. Also, each board has it's serial number and revision written on it either in pen or on a bar code.
- The version of RT860 Runtime and other software that you are using.
- The host operating system.
- The type of system that you are running on.

Returning Hardware For Repair

If you are convinced that your hardware is in need of repair, call Alacron and request a Return Materials Authorization (RMA) number. Address the shipment to Alacron at the address above, *Attention: RMA* ######### where the "#" marks represent the RMA number. The RMA number is our means of matching a returned shipment to our problem report database. When the technician starts to work on your boards, she'll have access to all of the information that you have given us.

When you call for an RMA, please have the following information ready:

- Serial numbers of all boards that will be shipped back.
- If you are returning an AL860 or FT200, please note the number of processors (1 or 2), the processor speed, and the size of memory.
- Provide an unambiguous description of the problem. If your board isn't failing our standard diagnostics, please provide us with code that will cause the failure or give us an exact description of the conditions that will elicit the problem. Note if the problem is sensitive to some environmental condition. This information will be logged in the RMA report and will be reviewed by the technician responsible for fixing your board.

Getting Help

Reporting Bugs

We continually strive to improve our product quality in order to best guarantee the success of your project. Occasionally, however, problems will surface in the field that were not encountered during our extensive testing. If you experience a hardware or software anomaly, please contact us immediately for assistance. Often, if a fix is not available immediately, we can devise a work-around that allows you to move forward.

It is important that we are able to reproduce your error in an isolated test case. Create a stand-alone code module that is isolated from your application and demonstrates the flaw. Describe the error within the code module and email the file to us at **support@alacron.com**. We will compile and run the module, and track down the anomaly in question. If you do not have Internet access, copy the code to a disk and mail it to us, or if small enough, fax a listing of the code module to us at **603-891-2745**.

When describing the software problem, please include the revision of all associated software. This includes the libraries, the Portland Group Tools, the host C compiler, and any Alacron development software. Also indicate the Alacron processor board (FT200/AL860) in use and any attached daughter cards.

For documentation errors, photocopy the page(s) of the manual in question and clearly mark the sections of interest. Along the top of the page(s), write the name and revision level of the manual. Then fax the page(s) to us at **603-891-2745**.

Getting Help

Index

/, 71
\, 71
AL860 variable, 21
Alacron, ii, **86**
alcall, 5
aldev, 5
almapload, 5
alopen, 5
alwait, 5
Building
  FT200-AT, 28
  FT200-PCI, 28
C Images, 21
cache, 75
cache coherency, 51
cache mode support, 26
cache modes, 77
CCIT, 21
DI, 21
DIO, 21
DMA, 22, 79
DT Connect, 21
electronic mail, 87
examples, 72
host, 5
host libraries
  location, 73
HRG, 21
i860
  clock speeds, 75
interprocessor communication, 25
interprocessor interrupts, 25
interrupts, 46
ioctl, 22
JPEG, 21
Kernel, 4
loading programs, 65
Main Board, 1
main processor, 25
MMU, 14
mp_4meg_addr, 42
mp_fprintf, 51
mp_printf, 51
mp_proc_id, 49
mp_sem_acquire, 34
mp_sem_release, 34
mp_thread_query, 32
mp_thread_start, 30
Mreentrant compiler flag, 28
Page Size
  4 KB, 42
  4 MB, 42
Portland Group Tools, 19
Real Time Clock, 39
Reentrancy
  C Runtime Library, 37
RIPL, 20
RMA, 86
rt860, 6, 64
  cache modes, 29
Scalability, 2
SCSI, 21
semaphores, 25, 34
setresult, 64
slave, 5
Slave Mode, 4
  compile line defines, 72
Slave Mode Library, 10
System Calls, 3
Technical Support, 85
Timing functions, 39
TLB, 78
userinit.h, 65
usersys, 64
usertrap, 64
VDA, 21
virtual memory, 14
VisionBus, 22
VLIB, 20
volatile
  C language keyword, 57
VSB, 21
W3SLV860, 60