IFX

Device Driver

DEVELOPER'S GUIDE

Document Number 523311002
May 26, 1989

REV.	MANUAL REVISION HISTORY	PRINT DATE
-002	IFX/68000, software release 1.06	5/26/89
-002	IFX/386, software release 1.05	5/26/89
-002	IFX/86, software release 1.05	5/26/89

Ready Systems makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Ready Systems assumes no responsibility for any errors that may appear in this document. The information in this document is subject to change without notice.

Copyright © 1989
Ready Systems
470 Potrero Avenue
P.O. Box 60217
Sunnyvale, California 94086
408/736-2600
FAX: 408/736-3400
TELEX: 711510608 (domestic)
0231510608 (international)

All rights reserved
Printed in U.S.A.

How To Use This Manual

Chapter 1 Overview

Chapter 2 Calling Conventions

Chapter 3 Managing Multitasking

Chapter 4 Handling Interrupts

Chapter 5 Device Driver Guidelines

Chapter 6 Supporting Function Codes

Chapter 7 Installing IFX Devices

Appendix A Example Device Drivers

Appendix B Sample 68000 Device Drivers

Appendix C Sample 386 Device Driver

Glossary

Index

List of Illustrations

List of Tables

List of Examples

How To Use This Manual

Purpose Of This Manual

IFX is available for several different microprocessor types. This manual contains information on how to write and install device drivers that are compatible with IFX/68000 version 1.06, IFX/386 version 1.05, and IFX/86 version 1.05.

It also includes samples of some commonly used device drivers and device managers, and examples that illustrate how to maximize your driver's performance. You can use the sample device drivers, which are included on your shipping media, to develop your own device drivers.

We recommend that you first install IFX as supplied and test it using the example application given in Appendix D of the *IFX User's Guide*. The example installs IFX standard devices, which require only minimal memory for operation.

Once you have the example application up and running, you can start developing your own device drivers following the guidelines given in this manual. We recommend you develop and test a single device at a time, so that it is easier to locate and correct any problems.

This developer's guide is part two of a two-volume manual set. Part one is the *IFX User's Guide.* It contains general information about IFX that you need to be familiar with before you can successfully write a custom device driver.

Intended Audience

This manual is for the system programmer who wants to write and install device drivers that are compatible with IFX. Knowledge of the C programming language, assembly language, and the architecture of your target microprocessor would facilitate your understanding of this manual. You should also be familiar with IFX system calls and basic concepts of real-time, multitasking programming. Consult the *IFX User's Guide* for more information about IFX.

How This Manual Is Organized

This manual is organized as follows:

- Chapter 1 defines a device driver and device manager, and their relationship to IFX.
- Chapter 2 presents general calling conventions that are applicable to all device driver types. These conventions must be observed by your device driver in order to maintain compatibility with IFX.
- Chapter 3 provides guidelines for managing multiple and simultaneous task operations.
- Chapter 4 shows how to implement interrupt handlers.
- Chapter 5 describes how to write a device driver for commonly used device types, such as disk, serial, and clock.
- Chapter 6 provides a reference of the function codes used with IFX device drivers.
- Chapter 7 explains how to install your IFX device driver and device manager.
- Appendices contain source code for sample device drivers that are included on your IFX shipping media, and examples useu to illustrate programming techniques.
- A glossary is provided to explain technical terms that may not be familiar to you. Terms defined in the glossary are set in **bold** type throughout the manual.

Where To Start

You should read Chapters 1, 2, 3, and 4 to familiarize yourself with general concepts that are applicable to all device driver types.

Chapters 5 and 6 provide necessary information for developing an IFX device driver.

To install your device driver, read Chapter 7.

Refer to the appendices for ready-to-use device driver samples and additional help.

Conventions

There are several conventions you should be aware of as you read the *IFX Device Driver Developer's Guide:*

- The term device driver is used to encompass both device drivers and device managers.
- Examples and functions are described using standard C language format. C examples frequently use a variable called *status* to receive a function return value. Because this variable is always declared as an *int*, the declaration is not repeated in each example.
- Hexadecimal numbers are represented in standard C language format. They start with *0x*. Numbers not prefixed by *0x* are decimal numbers.
- A notation, such as R[7:0], stands for register R, bits 7 through 0. Bit 0 is the least significant bit.
- Unless stated otherwise, the term *device driver* applies to both true device drivers and device managers. Only when the term *device manager* is used is a distinction made.

Related Documents

We recommend the following documents for additional information:

- Your VRTX32 user's guide describes the VRTX32 real–time multitasking kernel used with IFX.
- The *IFX User's Guide* describes how to install and use IFX, and it prepares you for developing your own device driver.
- *Getting Started With Ready Systems Software Components* provides details on the real–time software development process.
- *How to Write a Board Support Package for VRTX* provides information on writing device initialization and device interrupt handler code.
- *Interfacing a Language to Silicon Software Components* provides guidelines for writing an interface that allows a high-level language to make system calls to Ready Systems' software components.

- *Interfacing a Language to Silicon Software Components* provides guidelines for writing an interface that allows a high-level language to make system calls to Ready Systems' software components.
- *The C Programming Language* by Kernighan and Ritchie provides a definition of the C language.

Questions/Suggestions

If you have questions about IFX that are not answered by this manual, contact Ready Systems Application Engineering Department. To give us suggestions about this manual, use the reader comment card at the back of the manual. If the card is missing, send your suggestions to Ready Systems Technical Publications Department. Contact us at this address:

Ready Systems
470 Potrero Avenue
P.O. Box 60217
Sunnyvale, California 94086
408/736-2600
FAX:408/736-3400
TELEX: 711510608 (domestic)
0231510608 (international)

Chapter 1
Overview

◈ READY
SYSTEMS

1.1 Introduction

IFX provides several layers of software that your application can use to interact with hardware devices. These layers provide your application with the ability to access a device at different levels of complexity. Your application can work on a high-level with very little knowledge of the underlying device. Or, it can work at a low-level requiring some knowledge of device characteristics.

Each of these layers represents a different way of interacting with a device; thus, they are called **virtual devices**. Figure 1–1 illustrates the different IFX virtual device layers. A **device driver** is the function that IFX calls to handle I/O operations on a virtual device. A **device manager** is a particular type of device driver that translates high-level I/O requests, such as *open file*, to much simpler operations, such as *read disk sector*.

While IFX is generic and can be used with any kind of device, the device driver is specific to a particular device. The following built-in device drivers and device managers are included with IFX.

- Console terminal driver
- Pipe driver
- Clock driver
- Null driver
- MS-DOS File Manager
- Disk Buffer Manager
- Line-Edit Manager
- Circular Buffer Manager

These device drivers and managers are provided in binary form in the IFX component. Source code for additional device drivers is also provided on your shipping media and in Appendices B and C. The source code includes comments and installation instructions.

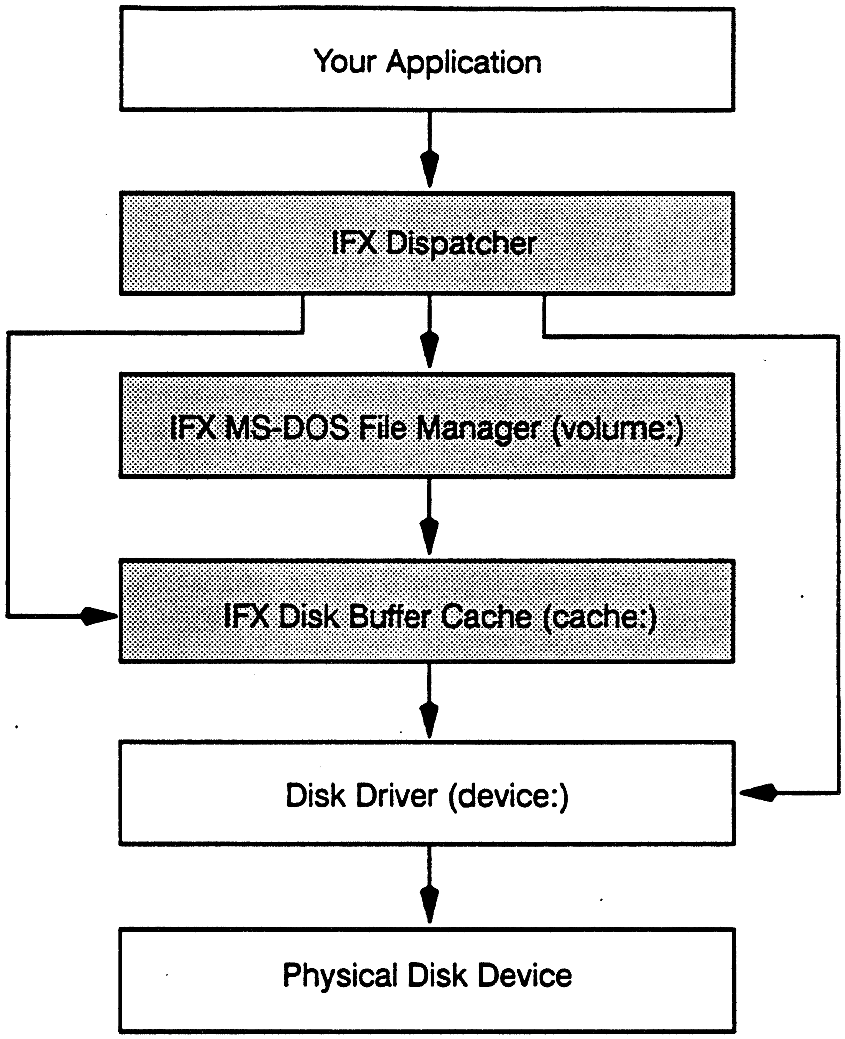


Figure 1–1 Virtual Disk Device Layers

Chapter 2
Calling Conventions

READY SYSTEMS

2.1 Introduction

All device drivers use a particular set of calling conventions to interact with a device. This chapter explains the guidelines for interfacing IFX device drivers and managers with your device. More detailed information for each device type, and how to install devices can be found in the remaining chapters of this manual.

IFX and the device driver follow the conventions described below.

- The device driver should only be called by IFX. Do not call the device driver directly from an application task or other user code.
- The device driver runs in the context of a task, as opposed to an interrupt handler. However, serial device drivers are devided into two parts: one part runs in a context of a task, and the other part is an ISR.
- IFX calls the device driver with interrupts enabled. If the device driver needs to disable interrupts, it should restore the interrupt status before returning.
- For the M68000 processor family, the device driver executes in supervisor mode. In particular, on the 68020 and 68030, the device driver uses the regular supervisor stack, as opposed to the interrupt stack.
- For the 80386, the device driver executes at privilege level 0.
- The device driver is allowed to call VRTX32 services, such as *sc_pend*, that suspend the caller.
- We do not recommend that the device driver call any IFX services. In particular, be aware of the potential for deadlock if the device driver calls an IFX service that involves another device.
- The device driver should return when it completes its operation or detects an error.

There are also rules that govern the use of parameters, registers, and status codes. These conventions are described in the following sections.

2.1.1 Parameters

The device driver gets its input parameters on the stack. The leftmost parameter is nearest to the top of the stack. The device driver should not remove the parameters from the stack when it returns. That is the responsibility of IFX.

- The first (leftmost) parameter to the device driver is an integer **function code** that tells the device driver what to do. The various function codes are listed in the appropriate section for each device type. IFX may call the device driver with other function codes and control operation codes that are *not* mentioned in this chapter. Refer to the discussion on Status Codes in Section 2.1.3.
- The second parameter to the device driver is a pointer to its **Device Control Block**, defined as the IFXDCB structure. Refer to Section 2.2 for more information.
- The third parameter to the device driver is a pointer to a **parameter list** containing other parameters needed for the specified function. Refer to the appropriate section for each device type for information on the parameter list.

The device driver returns all other outputs indirectly, through input parameters that are actually pointers to where to store the result. This corresponds to *call by reference* in high-level language terminology.

NOTE

You should read your C compiler manual for paramete-passing conventions. Conventions may vary from compiler to compiler, and even from version to version of the same compiler.

To summarize, IFX calls the device driver in C language as follows:

```
status = device_driver(func_code, &IFXDCB, &param_list);
```

Figure 2–1 illustrates parameter conventions.

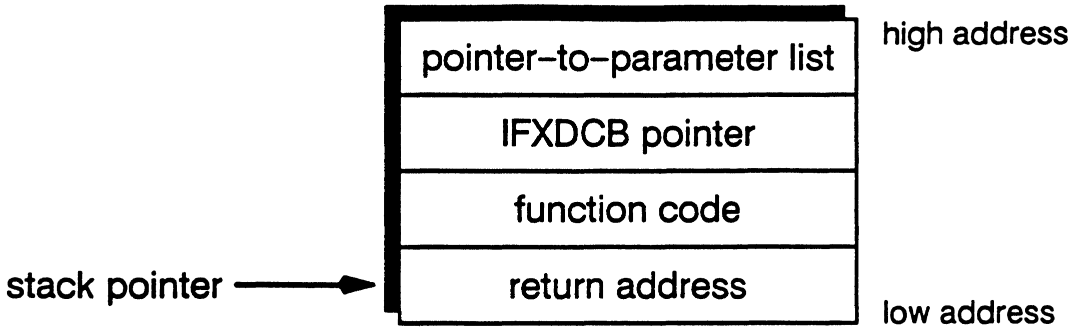


Figure 2–1 Stack Format After Entering Device Driver

All of the C device driver examples assume that your compiler pushes parameters onto the stack from right to left, and that the size of each parameter is that shown in Table 2–1. If this is not the case for your compiler (in particular the Alcyon and Intermetrics compilers), you may have to modify your parameter types so that the correct size value is pushed onto the stack.

Table 2–1 Standard Parameter Sizes

	int	long	pointer
68000	32 bits	32 bits	32 bits
80386	32 bits	32 bits	64 bits (32-bit offset, 16-bit selector, 16-bit filler)
8086	16 bits	32 bits	32 bits (16-bit offset, 16-bit segment)

2.1.2 Register Conventions

The device driver is allowed to destroy selected registers. If it uses any others, it should save and restore them.

- For M68000 processors, the device driver is allowed to destroy registers D1, A0, and A1.
- For the 80386 processor, the device driver is allowed to destroy registers EDX, EDI, and ES.
- For iAPX86 processors, the device driver is allowed to destroy registers BX, DX, DI, and ES.

Upon entry to the device driver:

- For M68000 processors, A5 contains the address of IFX workspace.
- For the 80386 processor, DS contains the selector of IFX workspace.
- For iAPX86 processors, DS contains the segment of IFX workspace.

This fact can be used to your advantage when using the direct-calling convention. Consult the *IFX User's Guide* for more information on IFX workspace.

2.1.3 Status Codes

A **status code** is an integer value returned by each IFX function that indicates the disposition of the requested operation. Your device driver should do the following:

- Return either zero (RET_OK), if successful, or an error status. If the device driver doesn't support a particular feature or understand the function code, it should return the error code IFXENOTIMP.
- Return the status code in register D0 for M68000 family processors, EAX for the 80386, and AX for iAPX86 family processors.

2.1.4 Writing a Device Driver in C for IFX/68000

The calling conventions for IFX device drivers are directly compatible with the code generated by the following 68000 C compilers:

- Ready Systems RTC
- Microtec
- Oasys/Green Hills
- Sun-3

For these compilers, you can directly install a C device driver. For other compilers, especially those with different calling conventions, you will need to write an assembly language routine that converts the IFX calling conventions to those used by your compiler.

For example, suppose your C compiler user's guide states that a function is allowed to destroy registers D0, D1, D2, A0, A1, and A2, with return values placed in register D7.

The code in Example 2–1 does the conversion:

```
MOVEM.L D2/D7/A2,-(SP)       * save registers not saved by compiler
MOVE.L  24(SP),-(SP)         * copy parameter list pointer
MOVE.L  24(SP),-(SP)         * copy IFXDCB pointer
MOVE.L  24(SP),-(SP)         * copy function code
JSR     C_portion            * call C portion of driver
LEA     12(SP),SP            * clean up stack
MOVE.L  D7,D0                * move status code to correct register
MOVEM.L (SP)+,D2/D7/A2       * restore registers
RTS                          * return to IFX
```

Example 2–1 68000 Assembly Language Conversion Routine

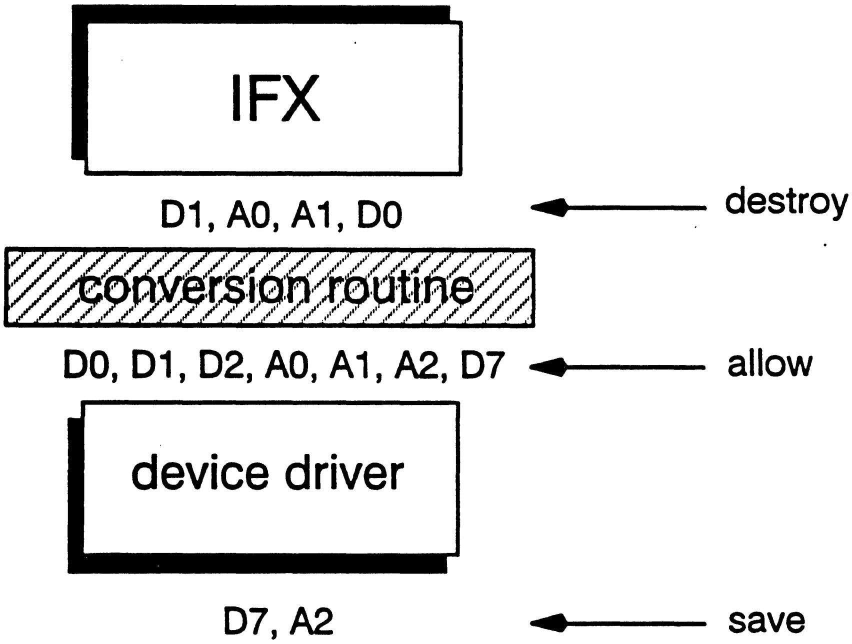


Figure 2–2 68000 Conversion Routine

2.1.5 Writing a Device Driver in C for IFX/386

For IFX/386, there must be an assembly portion for changing the data segment register (DS), since the IFX data segment is different than the driver data segment. The assembly routine in Example 2-2 should be defined as a FAR procedure. The C portion should be defined as "near."

```
push   ebp                 ; Save stack frame register
move   ebp,esp             ; Prepare stack frame register
push   es                  ; Save register to be used
push   ds                  ; Save IFX data segment register
mov    ax,NEW_DS           ; Restore DS to be the driver data segment register
mov    ds,ax
les    eax,[ebp+018h]      ; Copy parameter list pointer (onto the stack)
push   es
push   eax
les    eax,[ebp+010h]      ; Copy IFXDCB pointer (onto the stack)
push   es
push   eax
mov    eax,[ebp+0Ch]       ; Copy function code (onto the stack)
push   eax
call   C_portion           ; Call C portion of the driver
add    esp,014h            ; Clean up stack
pop    ds                  ; Restore IFX data segment register
pop    es                  ; Restore registers
pop    ebp
DB     OCBH                ; RETF - far return to the caller.
```

Example 2–2 80386 Assembly Language Conversion Routine

2.2 Device Control Block

The **Device Control Block**, or IFXDCB, is a data structure that is associated with each IFX device. There is one IFXDCB per device. The IFXDCB is allocated by IFX when the device is installed with *ifx_install* or *ifx_mount*. The IFXDCB is released by IFX when the device is removed with *ifx_remove*. A pointer to the IFXDCB is passed to a device driver as its second parameter.

```
/* Device Control Block */

typedef struct IFXDCB {
   long          *reserved1;     /* reserved for use by Ready Systems */
   unsigned char  device_type;   /* null, RAM disk, disk, volume, etc. */
   char           reserved2[11]; /* reserved for use by Ready Systems */
   struct IFXDCB *mounted_on;    /* address of underlying device's IFXDCB */
   char          *device_driver; /* address of device driver code */
   char          *dt;            /* device-type specific information */
   long           reserved3[3];  /* reserved for use by Ready Systems */
   IFXDDCB       *my_driver;     /* address of driver's IFXDDCB */
   long           reserved4[5];  /* reserved for use by Ready Systems */
} IFXDCB;
```

The IFXDCB contains a number of fields, some of which are undocumented and reserved for use by IFX, and others that are documented and of interest to a device driver. Your device driver should not refer to the undocumented fields, as they are subject to change.

The documented fields are:

device_driver This field is set by IFX to point to the device driver code. Device drivers should ignore this field. Device managers use it to call the underlying device driver.

device_type This must be set by the driver to the device type code (IFXD . . .) when the driver is called with the IFXINSTALL opcode. The initial value is IFXDUNKNOWN (zero).

dt This is a general-purpose field that is reserved for use by the driver. The field is the size of a pointer (4 bytes for 68000 and 8086, 6 bytes for 80386). The field typically points to global variables used by the driver. The initial value is NULL (zero).

mounted_on For device drivers, this field is always zero and should be ignored. For device managers, this field is set by IFX to point to the IFXDCB of the underlying device.

my_driver This field is set by IFX to point to the associated Device Driver Control Block, or IFXDDCB for this device. Most device drivers can safely ignore it, but it is occasionally useful for reentrant drivers that need to keep track of variables that are shared among all devices serviced by the same driver. Refer to Section 2.3 for more information.

2.3 Device Driver Control Block

The **Device Driver Control Block**, or IFXDDCB, is a data structure that is associated with each device driver. There is one IFXDDCB per device driver. The IFXDDCB is allocated by IFX when the device driver is installed with *ifx_driver*. The IFXDDCB is released by IFX when the device driver is removed with *ifx_rmdriver*. A pointer to the IFXDDCB is stored in field my_driver of the IFXDCB.

```
/* Device Driver Control Block */

typedef struct IFXDDCB {
   long           reserved1[5]; /* reserved for use by Ready Systems */
   char          *ddt;          /* driver specific information */
   long           reserved2[2]; /* reserved for use by Ready Systems */
} IFXDDCB;
```

The IFXDDCB contains a number of fields, all of which are undocumented and reserved for use by IFX, except for one that is of interest to a device driver. Your device driver should not refer to the undocumented fields, as they are subject to change. The documented field is:

ddt This is a general-purpose field that is reserved for use by the driver. The field is the size of a pointer (4 bytes for 68000 and 8086, 6 bytes for 80386). The field typically points to global variables used by the driver. The initial value is NULL (zero).

Chapter 3
Managing Multitasking

3.1 Introduction

IFX assumes that the device can perform an unlimited number of I/O operations concurrently. For this reason, IFX does not lock the device during I/O. If the device is *not* capable of concurrent I/O operations, then the device driver is responsible for servicing requests in the appropriate order.

This chapter discusses how to manage concurrent I/O operations by using mailboxes, semaphores, and high-level device managers. These locking mechanisms can control concurrent reads and writes, the order in which tasks pend, as well as preemptive device scheduling.

3.2 Locking Mechanisms

The driver can prevent simultaneous I/O operations by using an appropriate VRTX32 locking mechanism, such as a mailbox or semaphore.

IFX/68000 also provides a very fast semaphore that device drivers can use for locking without incurring the overhead of VRTX32 mailbox or semaphore calls. It is implemented in the IFXSEMA structure contained in the *ifxvisi.h* file. (Consult the *IFX User's Guide* for information on definition files included with your shipping media.) Your shipping media contains assembly language routines that use IFXSEMA. These routines are contained in a file with the filename root of *semavisi* (for example, *semavisi.a68* for the 68000 family of processors).

> **CAUTION**
>
> Avoid using VRTX32 *sc_fpend* and *sc_fpost*, VRTX32 *sc_lock* and *sc_unlock*, or interrupt disabling to prevent simultaneous I/O operations. These mechanisms may have potential race conditions (*sc_fpend* and *sc_fpost*), may disable scheduling too long (*sc_lock* and *sc_unlock*), or may disable interrupts too long (interrupt disabling).

3.2.1 Simple Locking

For most devices, a simple semaphore is adequate for managing multitasking. Example 3–1 provides disk driver code that shows how to use a semaphore to prevent multiple simultaneous I/O operations.

```
int disk_driver(opcode, dcb_ptr, plist)
int opcode;
IFXDCB *dcb_ptr;
PL *plist;
{
    static int sema;  /* holds semaphore ID number */
    int err;
    switch (opcode) {
    case IFXFINSTALL:
        sema = sc_screate(1, 0, &err);
        /* do other initialization here */
        break;
    case IFXFREADS:
        sc_spend(sema, 0L, &err);
        /* do read here */
        sc_spost(sema, &err);
        break;
    case IFXFWRITES:
        sc_spend(sema, 0L, &err);
        /* do write here */
        sc_spost(sema, &err);
        break;
    case IFXFREMOVE:
        sc_sdelete(sema, &err);
        break;
    }
    return err;
}
```

Example 3–1 Locking with Semaphores

3.2.2 Concurrent Reads and Writes

There are some devices that can do both a read and a write simultaneously, but no more than one read or write at once. An example of this device type is a serial communication line. By using two semaphores, one for reading and the other for writing, the driver can enforce this rule.

```
int serial_driver(opcode, dcb_ptr, plist)
int opcode;
IFXDCB *dcb_ptr;
PL *plist;
{
    static int read_sema, write_sema;
    int err;
    switch (opcode) {
    case IFXFINSTALL:
        read_sema = sc_screate(1, 0, &err);
        write_sema = sc_screate(1, 0, &err);
        /* do other initialization here */
        break;
    case IFXFREAD:
        sc_spend(read_sema, 0L, &err);
        /* do read here */
        sc_spost(read_sema, &err);
        break;
    case IFXFWRITE:
        sc_spend(write_sema, 0L, &err);
        /* do write here */
        sc_spost(write_sema, &err);
        break;
    case IFXFREMOVE:
        sc_sdelete(read_sema, &err);
        sc_sdelete(write_sema, &err);
        break;
    }
    return err;
}
```

Example 3–2 Controlling Concurrent Reads and Writes with Semaphores

3.3 Order of Processing Requests

If you use either a mailbox or the IFX fast semaphore, then I/O requests that arrive while another I/O operation is active pend in task-priority order. If you use a VRTX32 semaphore, then you can choose whether tasks pend in priority order or first-in first-out order (a parameter to *sc_screate* selects the order). However, you may desire more sophisticated control over the order in which requests are processed.

For example, for a disk you may want tasks to pend in one of the following orders:

- **Shortest seek time first.** Choose the I/O request whose starting sector number is nearest to the sector that was just transferred. This tends to reduce the amount of disk head movement.

- **Shortest transfer count first.** Choose the I/O request that has the smallest transfer count. This tends to give preference to small I/O requests over large I/O requests.

- **Elevator.** Choose I/O requests in increasing order by starting sector number, until there are no more requests. Then choose I/O requests in decreasing order by ending sector number, until there are no more requests. Continue alternating between increasing order and decreasing order. This is called the **elevator algorithm** because it resembles a building elevator that moves in one direction until there are no more passengers to pick up, then switches direction.

It is not possible to implement any of the above order sequences using a mailbox or semaphore alone. However, by adding data structures to keep track of I/O requests that have been received but not started, these orderings and many others can be implemented. The advanced request-ordering example in Appendix A, shows how to do this.

For most applications, we recommend that your driver process requests in either task-priority or first-in first-out order. Experience shows that if the average queue length of waiting requests is small (one or two), then the order of processing requests makes no difference. If the average queue length is large, then the order of processing requests is significant. Yet this indicates that there may be a severe mismatch in performance between the application and the device. Perhaps a faster device is needed rather than clever scheduling.

3.4 Preemptive Device Scheduling

There are considerations for a device driver that completes one I/O operation before it begins the next one. If a low-priority task is in the middle of a long data transfer and a high-priority task calls the driver, then the high-priority task must wait for the low-priority task to finish before it can begin using the device. An advanced driver could implement preemptive scheduling of the device. This type of driver could suspend the low-priority task's I/O operation, then do the high-priority task's I/O, and finally resume the low-priority task's I/O.

For most applications, we do not recommend such elaborate device scheduling. Implementation of preemptive scheduling is beyond the scope of this manual. Instead, we suggest that you use separate physical devices for critical and noncritical tasks, rather than share a device. For example, critical tasks could use a small hard disk to store incoming data. Noncritical tasks could use a large optical disk to archive this data. Then, periodically, you could copy files from the small disk to the large disk. As a result, there would be no contention for disk resources.

3.5 Locking by High-Level Device Managers

It may appear that a disk device driver is never called simultaneously by more than one task. This is because the higher-level disk managers, such as the disk buffer cache manager and volume manager, use their own semaphores to enforce mutual exclusion. Thus, there is little chance for the disk device driver to be called simultaneously from more than one task, but it is not entirely impossible. For example, if an application task opens the disk device by name, and issues *ifx_reads* or *ifx_writes* calls to the device, then it will bypass the semaphores of the Disk Buffer Cache Manager and the Volume Manager. The result is that the disk device driver can be called simultaneously by more than one task. Therefore, we recommend that your driver include the semaphores to handle this case, even if it appears unnecessary. (See Figure 1–1 for an illustration of virtual device layers.)

3.6 Global Variables

When a driver returns to its caller, any information that it has put in variables on the stack or in registers will be lost. Therefore, if the driver needs to keep information around for a longer period, such as across calls to the driver, it has three choices:

1. Keep the information in global variables. This is the simplest solution, but it has the disadvantage of making the driver nonreentrant (the driver can only service one device). If there were more than one device of the same class (for example, two disk drives), then there would be no way for the driver to know which device, and which global variables, it should use.

2. If the global storage required is only a few bytes, the driver can simply store the information directly in the dt field of the IFXDCB.

3. If the global storage required is greater than the size of a pointer, then the driver should place a pointer to a structure containing the information into the dt field. This pointer should be initialized by the driver when it is called with opcode IFXFINSTALL to point to global variables used by the driver.

3.7 Reentrancy

Since IFX may reenter the device driver before it exits from a previous call, the device driver *must* be reentrant. A fully **reentrant function** can be used by more than one task at the same time. The reentrant function can be interrupted at any point and resume processing later at that same point without loss of data. Reentrant functions do not store any data to the data segment (global memory).

To illustrate the need for reentrant code, suppose a task places a value in a global variable. When the task is preempted, a second task might use the same function and overwrite the variable with its own value. When the first task resumes execution, the original value of the variable has been destroyed.

There are many advantages to making your device driver reentrant, such as greater versatility and better structure. A **reentrant device driver** can service more than one device attached to a single controller board. And if there are several controller boards of the same kind, the device driver can even service more than one controller board.

It is easy to achieve reentrancy. A device driver should only use registers or the stack for local variables, and avoid using global variables, self-modifying code, and other nonreentrant programming practices. If global variables must be used, they should be protected by a locking mechanism around the critical region of code.

3.8 Dependent Devices

IFX assumes that each device is independent of all other devices. However, this is not true if two devices share a common controller board that is not capable of simultaneous I/O operations on both devices. For example, a typical disk controller board can handle up to four disk drives, but can only service one disk drive at a time. If you have dependent devices, add a locking mechanism in each device driver around the critical region of code.

3.9 Direct Memory Access

Many intelligent device controllers support **Direct Memory Access (DMA)**. DMA is a technique whereby an external device can transfer large amounts of information to and from the processor's memory without the processor having to be involved. This offers improved performance since the device can access memory in bursts at full bus speed while the processor also continues to independently execute at almost full speed.

Memory that can be accessed by both the processor and via DMA is called **dual-ported memory**. Dual-ported memory is addressed in two different ways depending on who is using it. The address used by the processor is called the **internal address**. The address used by DMA is called the **external address**. These two addresses often differ by a fixed constant.

All address parameters passed to a device driver by IFX are internal addresses. It is the device driver's responsibility to convert between internal addresses and external addresses as necessary.

For example, the MVME133 68020 processor board includes 1 megabyte of memory. This memory, as seen by the processor, is always addressed as 0x00000000 to 0x000FFFFF (internal address). As seen from the VMEbus, however, the memory can be located on any 1-megabyte address boundary (external address). The actual address boundary is selected by option jumpers on the MVME133 board. Therefore, if the external address range was jumpered as 0x00200000 to 0x002FFFFF, then the device driver would have to add 0x00200000 to any internal address in order to convert it to an external address.

Chapter 4
Handling Interrupts

READY SYSTEMS

4.1 Introduction

IFX does not have special features related to interrupts, yet most device drivers generate and wait for interrupts as part of their normal operation.

When IFX calls a device driver, it expects the driver to perform the operation and then return to IFX with a status code. If the driver happens to use interrupts, this is of no concern to IFX and should be completely hidden from it. An IFX driver is simply a subroutine that is called at task level. Since the driver executes at task level, it is free to use VRTX32 features such as mailboxes, queues, semaphores, and event flags to manage interrupts.

This chapter shows the recommended ways for a device driver to manage interrupts using VRTX32 interrupt handlers.

This information is not applicable to serial drivers that are used with the Circular Buffer Manager. Refer to Chapter 5 for information on handling serial interrupts.

4.2 Mailboxes

The simplest VRTX32 feature for managing interrupts is the mailbox. A **mailbox** is a pointer-sized variable that you define, located in user read/write memory. Mailboxes coordinate data transfer between a device driver and the interrupt handler. Mailboxes are fast and easy to use, so in the absence of other considerations, we recommend that you use a mailbox to handle interrupts.

We assume your device generates an interrupt when it completes an operation. Therefore, the steps below should be followed in the driver to use a mailbox:

1. Initialize mailbox to zero.

2. Start the I/O operation.

3. Issue the *sc_pend* call to wait for an interrupt. An optional timeout may be used if there is a chance that the interrupt won't happen.

4. If the *sc_pend* returns a timeout error code, then cancel the I/O and return an error code.

5. Clean up after the I/O operation is done.

The interrupt service routine should look like this:

1. Save register D0, AX, or EAX, depending on the processor.

2. Issue UI_ENTER for 80386 and 8086, or 68000 and 68010, if interrupt stack switching is enabled.

3. Save other registers used by interrupt service routine.

4. Check the device to make sure that it caused the interrupt. If not, go to step 7.

5. If device returns a small amount of data at the end of the I/O operation, the interrupt service routine should acquire this data from the device and save it in a global variable. Large amounts of data (more than a few bytes) should be transferred at task level to minimize the interrupt service routine execution time.

6. Issue the *sc_post* call, passing it the address of the mailbox and any nonzero message. This marks the task pending for the mailbox as ready to run.

7. Restore registers other than D0, AX, or EAX.

8. Issue the UI_EXIT call. This schedules the task to run.

Example 4–1 shows a device driver that uses a mailbox with a corresponding 68000 interrupt service routine (shown in Example 4–2). The device is capable of reading a large number of bytes through direct memory access (DMA), and generating a single interrupt when the transfer is done.

```
char *mailbox;

void device_driver(opcode, dcb_ptr, plist)
int opcode;
IFXDCB *dcb_ptr;
struct  {
    char *buffer;
    long desired_count;
    long *actual_count;
} *plist;
{
    int err;
    switch (opcode) {
    ...
    case IFXFREAD:
        mailbox = 0;
        /* set up DMA controller here, using
           ´plist->buffer´ and ´plist->desired_count´
        */
        /* start read operation here */
        sc_pend(&mailbox, 0L, &err);
        /* now the read is done */
        *plist->actual_count = plist->desired_count;
        break;
    ...
    }
    return err;
}
```

Example 4-1 Device Driver

```
interrupt_service_routine:
        MOVE.L   D0,-(SP)           * save D0
        MOVEQ.L  #UIFENTER,D0       * issue UI_ENTER call
        TRAP     #0
        MOVEM.L  D1/A0,-(SP)        * save other registers
        BTST.B   #DONE,IO_port      * check if I/O is done
        BEQ.S    not_done           * if not, ignore interrupt
        MOVEA.L  #mailbox,A0        * load mailbox address
        MOVEQ.L  #1,D1              * load message to post
        MOVEQ.L  #SCFPOST,D0        * issue SC_POST call
        TRAP     #0
not_done:
        MOVEM.L  (SP)+,D1/A0        * restore other registers
        MOVEQ.L  #UIFEXIT,D0        * issue UI_EXIT call
        TRAP     #0                 * this also restores D0
```

Example 4–2 Interrupt Service Routine

4.3 Event Flags

Another VRTX32 feature suitable for managing interrupts is the event flag group. An **event flag group** is a global, long-word (32-bit) structure in VRTX32 Workspace. Each of the 32 bits in the event flag group is an **event flag**. The main advantage of event flags over mailboxes is that you can wait for more than one event. With a mailbox, you can wait for only one event. A detailed example of event flag use can be found in Appendix A.

4.4 Using UI_ENTER and UI_EXIT

The first instruction of an interrupt service routine is UI_ENTER, and the last is UI_EXIT. The UI_ENTER and UI_EXIT pair serves three purposes:

- Prevents preemption in the middle of an interrupt service routine
- Ensures correct functioning of the task rescheduling mechanism
- Implements the optional interrupt stack switching mechanism for processors that do not have built-in stack switching

To be safe we recommend that you always use UI_EXIT, and always use UI_ENTER except for 68000 family processors.

It is possible to achieve a slight performance improvement by not using UI_ENTER and UI_EXIT in an interrupt service routine, if certain strict conditions are met. The rules are summarized below:

- If the interrupt service routine does not make any system calls and cannot possibly be interrupted by another interrupt service routine which contains UI_ENTER or UI_EXIT, then both UI_ENTER and UI_EXIT may be omitted.
- For 68000 and 68010 processors, UI_ENTER may be omitted if interrupt stack switching is not enabled.
- If the interrupt service routine handles non-maskable interrupts, or for 68000 family processors, handles interrupts at or above component disable level, then the interrupt service routine *must not* use UI_ENTER or UI_EXIT, and must not make system calls.

If you do not follow these rules, any of the following things can happen:

- Interrupt service routines may be preempted
- Low priority tasks may run instead of high priority tasks
- Interrupt stack switching may not occur even when it is enabled
- System may crash

You should use caution in omitting UI_ENTER and UI_EXIT. For additional information about UI_ENTER and UI_EXIT, consult your VRTX32 user's guide.

4.5 Writing an Interrupt Service Routine in C

It is possible to write an IFX interrupt service routine in C, but you must first write a small assembly language program. The calling conventions of C compilers are incompatible with processor conventions for interrupt service routines. The assembly language code converts between the two conventions. Even C compilers that have an option to produce interrupt code, still need this assembly language program because they do not generate the required UI_ENTER and UI_EXIT system calls (refer to Section 4.4).

C code for an interrupt service routine looks like this:

```
void c_code()
{
   /* body of interrupt service routine goes here */
}
```

If the C compiler supports run-time stack overflow detection, this feature must be disabled since an interrupt service routine can be called in the context of any task stack.

The remainder of this section contains assembly language examples for each of the processors supported by IFX and for representative C compilers. For other compilers, consult your compiler documentation for calling conventions.

Processor: **68000 or 68010**
Compiler: **Ready Systems RTC, Oasys/Green Hills, Microtec**

```
        XREF      .c_code
isr:
        MOVE.L    D0,-(SP)         * save D0
        MOVEQ.L   #UIFENTER,D0     * issue UI_ENTER call
        TRAP      #0
        MOVEM.L   D1/A0/A1,-(SP)   * save registers not saved by compiler
        JSR       .c_code          * call C function
        MOVEM.L   (SP)+,D1/A0/A1   * restore other registers
        MOVEQ.L   #UIFEXIT,D0      * issue UI_EXIT call
        TRAP      #0
```

Processor: **68020 or 68030**
Compiler: **Ready Systems RTC, Oasys/Green Hills, Microtec**

```
        XREF      .c_code
isr:
        MOVE.L    D0,-(SP)         * save D0
        MOVEM.L   D1/A0/A1,-(SP)   * save registers not saved by compiler
        JSR       .c_code          * call C function
        MOVEM.L   (SP)+,D1/A0/A1   * restore other registers
        MOVEQ.L   #UIFEXIT,D0      * issue UI_EXIT call
        TRAP      #0
```

Processor: **68020 or 68030**
Compiler: **Sun–3**

```
isr:
        movel      d0,sp@-          | save D0
        moveml     #0x0302,sp@-     | save registers not saved by compiler
        jsr        _c_code          | call C function
        moveml     sp@+,#0x40C0     | restore other registers
        moveql     #UIFEXIT,d0      | issue UI_EXIT call
        trap       #0
```

Processor: **80386**
Compiler: **Metaware**

```
        EXTRN      _c_code:FAR
isr:
        PUSH       EAX              ; save EAX
        MOV        EAX,UIFENTER     ; issue UI_ENTER call
        INT        0FFH
        PUSH       EBX              ; save registers not saved by compiler
        PUSH       ECX
        PUSH       EDX
        PUSH       ESI
        PUSH       EDI
        PUSH       GS
        PUSH       FS
        PUSH       ES
        PUSH       DS
        MOV        AX,DGROUP        ; set up DS selector register
        MOV        DS,AX
        CALL       _c_code          ; call C function
        POP        DS               ; restore other registers
        POP        ES
        POP        FS
        POP        GS
        POP        EDI
        POP        ESI
        POP        EDX
        POP        ECX
        POP        EBX
        MOV        EAX,UIFEXIT      ; issue UI_EXIT call
        INT        0FFH
```

Processor: **8086**
Compiler: **Microsoft**

```
        EXTRN   _c_code:FAR
isr:
        PUSH    AX              ; save AX
        MOV     AX,UIFENTER     ; issue UI_ENTER call
        INT     81H
        PUSH    BX              ; save registers not saved by compiler
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP       ; set up DS segment register
        MOV     DS,AX
        CALL    _c_code         ; call C function
        POP     DS              ; restore other registers
        POP     ES
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        MOV     AX,UIFEXIT      ; issue UI_EXIT call
        INT     81H
```

4.6 Installing Interrupt Service Routines

This section shows how to install an interrupt service routine for each of the processors supported by IFX. In each case, the symbol *vector* is the interrupt vector number, and the symbol *isr* is the starting address of the interrupt service routine. If the interrupt service routine is written in C, *isr* refers to the assembly language portion of the interrupt service routine.

68000

```
        LEA       isr,A0            * get interrupt service routine addres
        MOVEA.L   #vector*4,A1      * get interrupt vector address
        MOVE.L    A0,(A1)           * update interrupt vector
```

68010, 68020, and 68030

```
        LEA       isr,A0            * get interrupt service routine address
        MOVEC.L   VBR,A1            * get vector base register address
        ADDA.L    #vector*4,A1      * add interrupt vector offset
        MOVE.L    A0,(A1)           * update interrupt vector
```

80386

```
;Step 1:  Build the IDT alias descriptor for the IDT

;If:  IDT_ALIAS_DES = IDT alias within the GDT
;     IDT_BASE = the new IDT base address
;     IDT_SIZE = 2 * biggest interrupt number
;     STD_DATA_ACCESS = 092h
;     STD_DATA_GRAN = 040h
;     ES is prepared to have the GDT base address.
;        The structure of each GDT entry is of the DESC structure.

DESC STRUC
     lim_0_15     DW   0     ;limit bits (0..15)
     bas_0_15     DW   0     ;base bits (0..15)
     bas_16_23    DB   0     ;base bits (16..23)
     access       DB   0     ;access byte
     gran         DB   0     ;granularity byte
     bas_24_31    DB   0     ;base bits (24..31)
DESC ENDS

     MOV    EAX,IDT_BASE
     MOV    Word ptr ES:[IDT_ALIAS_DES].bas_0_15,AX
     SHR    EAX, 16
     MOV    Byte ptr ES:[IDT_ALIAS_DES].bas_16_23,AL
     MOV    Byte ptr ES:[IDT_ALIAS_DES].bas_24_31,AH
     MOV    Word ptr ES:[IDT_ALIAS_DES].lim_0_15,IDT_SIZE - 1
     MOV    Byte ptr ES:[IDT_ALIAS_DES].access,STD_DATA_ACCESS
     MOV    Byte ptr ES:[IDT_ALIAS_DES].gran,STD_DATA_GRAN


Step 2:  Set up the ISR interrupt gate

;If:  ISR_name = the ISR to be set
;     ISR_CODE_DES = the code descriptor in which ISR_name resides
;     INT_entry_no = the IDT entry number
:     INT_ACCESS = 0EE00h
:     ES is prepared to have the IDT base address.
;        The structure of each IDT entry is of the INTDESC structure

IDTDESC STRUC
     IDT_off_0_15     DW   0    ;code offset (0..15)
     IDT_selector     DW   0    ;code selector
     IDT_access       DW   0    ;access word
     IDT_off_16_31    DW   0    ;code offset (16..31)
IDTDESC ENDS

     MOV    EAX,offset ISR_name
     MOV    word ptr ES:[INT_entry_no*8].IDT_off_0_15,AX
     SHR    EAX,16
```

```
MOV   word ptr ES:[INT_entry_no*8].IDT_off_16_31,AX
MOV   word ptr ES:[INT_entry_no*8].IDT_selector,ISR_CODE_DES
MOV   word ptr ES:[INT_entry_no*8].IDT_access,INT_ACCESS
```

8086

```
MOV       DI,vector*4          ; get interrupt vector entry offset
MOV       AX,0                 ; load segment register with zero
MOV       ES,AX
MOV       AX,OFFSET isr        ; get offset portion of routine address
MOV       BX,SEG isr           ; get segment portion of routine address
PUSHF                          ; save interrupt status
CLI                            ; disable interrupts
MOV       ES:[DI],AX           ; update offset portion of vector
MOV       ES:[DI+2],BX         ; update segment portion of vector
POPF                           ; restore interrupt status
```

4.7 Using Time-outs

A typical device driver follows these steps:

1. Initiates the I/O operation
2. Waits for the I/O to complete
3. Returns results to the application

But what if step 2 never finishes; that is, the I/O never completes? Should the device driver then wait indefinitely and never return to the application?

There are many reasons why the operation might not terminate:

- External device does not respond
- Hardware failure
- Lost interrupt

Many of these causes are transient in nature. It is intolerable in a real-time system for a single failure, especially a temporary failure, to cause the application to suspend indefinitely.

It is the responsibility of the device driver to recover from these situations. The driver should either retry the operation or return an error code to the application, depending on the severity of the error.

The recommended way for a device driver to recover is to use time-outs. A **time-out** is a maximum period of time that the driver is willing to wait for the I/O operation to complete. If the operation does not finish within that interval, then the operation is said to have "timed out."

A device driver that supports time-outs follows these steps:

1. Initiates the I/O operation
2. Waits for the I/O to complete, or for a time-out
3. If the I/O completes, then returns results to the application
4. Otherwise, if the I/O times out, returns an error code to the application
5. Returns results to the application

All VRTX32 synchronization features, such as mailboxes, queues, semaphores, and event flags, support time-outs. All pend system calls, such as *sc_pend*, *sc_qpend*, *sc_spend*, and *sc_fpend*, include a time-out parameter. This parameter, if nonzero, specifies the maximum number of clock ticks that the caller is willing to pend. If the corresponding post call is not called within the time-out period, then the pend call returns error code ER_TMO.

A first attempt to support time-outs might look like this:

```
#define TIMEOUT 100L
char *mailbox;

int device_driver(opcode, dcb_ptr, plist)
int opcode;
IFXDCB *dcb_ptr;
struct {
    char *buffer;
    long desired_count;
    long *actual_count;
} *plist;
{
    int err, status;
    switch (opcode) {
    ...
    case IFXFREAD:
        mailbox = 0;
        /* set up DMA contoller here, using
           'plist->buffer' and 'plist->desired_count'
        */
```

```
                /* start read operation here */
                sc_pend(&mailbox, TIMEOUT, &err);
                if (err == RET_OK) {
                    *plist->actual_count = plist->desired_count;
                    status = RET_OK;
                } else if (err == ER_TMO) {
                    *plist->actual_count = 0L;
                    status = IFXETIMEOUT;
                }
                break;
            ...
        }
        return status;
    }
```

The interrupt handler is:

```
    void interrupt_handler()
    {
        int err;
        sc_post(&mailbox, (char *) 1, &err);
    }
```

There is a subtle race condition in the above code. What happens if the time-out occurs and *sc_pend* returns ER_TMO, then the I/O operation eventually completes long after the device driver has returned to the application? This could be disastrous, since the data buffer is overwritten, although the application no longer expects it to be modified.

The missing step is I/O cancellation. After a time-out, the device driver must cancel the I/O operation so that it cannot complete later on. The revised device driver code looks like this:

```
    ...
    } else if (err == ER_TMO) {
        /* cancel I/O operation here */
        *plist->actual_count = 0L;
        status = IFXETIMEOUT;
    }
    ...
```

There is still one more race condition in the above code. What happens if the I/O operation completes after the *sc_pend* call returns error code ER_TMO, but before the I/O is canceled? The window of time during which this could occur varies from a few microseconds to much longer, depending on whether the device driver is the highest priority task in the system.

A solution is to add a flag variable, which is set by the interrupt handler upon I/O completion. The device driver checks this flag after *sc_pend* returns, rather than rely on the error code returned by *sc_pend*. This check and the following I/O cancellation must be a single atomic (indivisible) operation. The recommended way to do this is to disable interrupts before checking, and enable interrupts again after the I/O is canceled.

The final device driver code looks like this:

```
#define TIMEOUT 100L

char *mailbox;
int flag;

int device_driver(opcode, dcb_ptr, plist)
int opcode;
IFXDCB *dcb_ptr;
struct {
    char *buffer;
    long desired_count;
    long *actual_count;
} *plist;
{
    int err, status;
    switch (opcode) {
    ...
    case IFXFREAD:
        mailbox = 0;
        flag = 0;
        /* set up DMA contoller here, using
           'plist->buffer' and 'plist->desired_count'
        */
            /* start read operation here */
        sc_pend(&mailbox, TIMEOUT, &err);
        disable_interrupts();
        if (flag) {
            enable_interrupts();
            *plist->actual_count = plist->desired_count;
            status = RET_OK;
        } else {
            /* cancel I/O operation here */
            enable_interrupts();
            *plist->actual_count = 0L;
            status = IFXETIMEOUT;
        }
        break;
    ...
    }
    return status;
}
```

The final interrupt handler is:

```
void interrupt_handler()
{
    int err;
    sc_post(&mailbox, (char *) 1, &err);
    flag = 1;
}
```

Note that even in this solution, there is a chance that the I/O could complete between the if flag test and the point where the I/O is canceled. This would not result in an interrupt since interrupts are disabled at that point. There is no way to avoid this race condition, as the processor and the device are inherently asynchronous. The worst that could happen is that the driver could declare a time-out error, although the I/O completed successfully, and it would attempt to cancel the I/O unnecessarily. Because of the slight possibility of this unavoidable condition, the device hardware must be designed in such a way that there are no ill effects from canceling the I/O after it has completed.

Chapter 5

Device Driver Guidelines

◆ READY SYSTEMS

5.1 Introduction

This chapter describes how to write a device driver for different kinds of devices. It includes information on disk, serial, and clock device drivers, and device managers. You should be familiar with the general information in Chapter 2 and your *IFX User's Guide* before proceeding with development of a device driver.

5.2 Disk Device Drivers

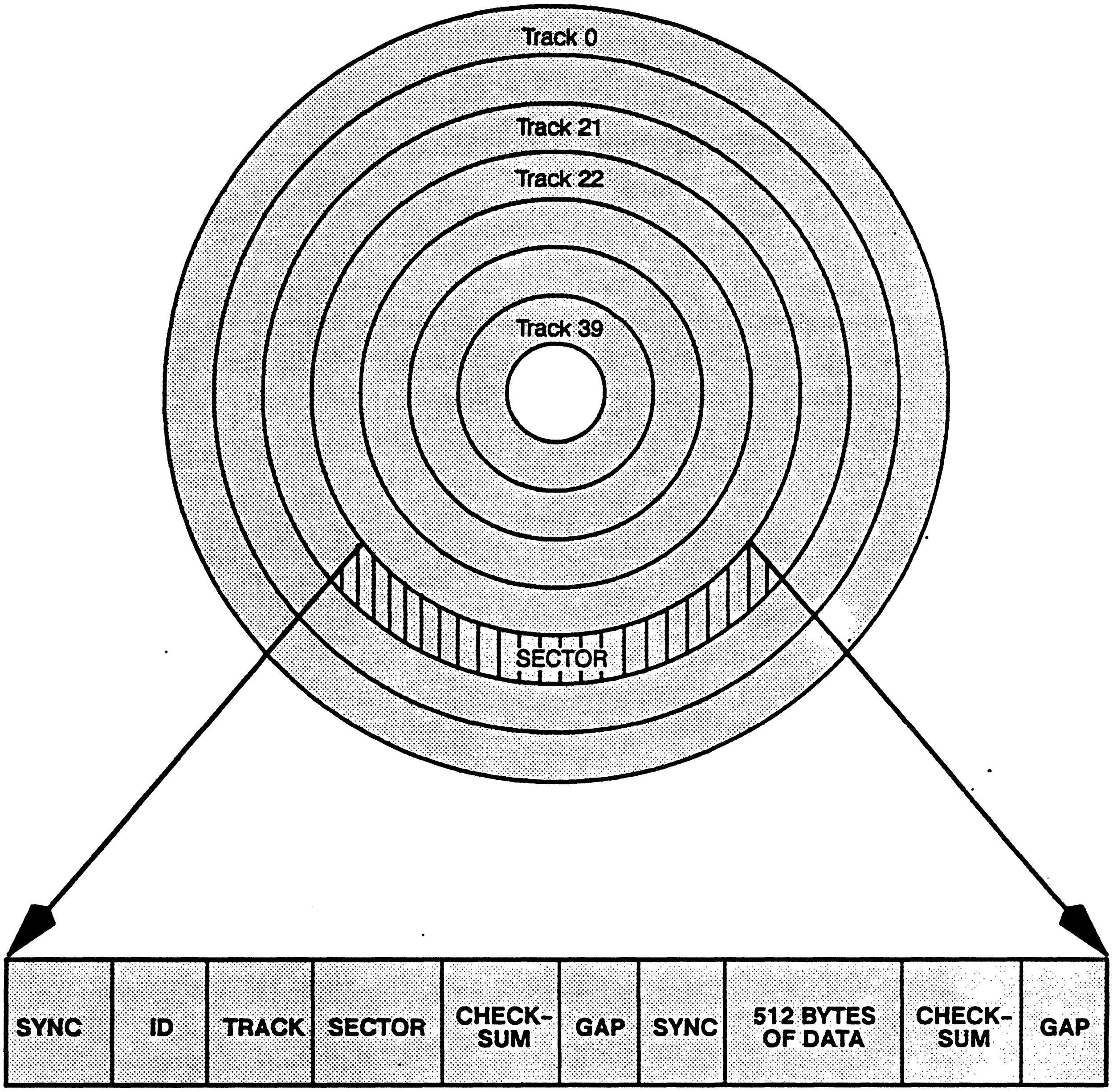
This section briefly explains the media structure of an IFX disk, then shows how to write a disk device driver. For each function code that IFX uses to call the driver, this section describes the parameter list and the actions the driver should perform. You should refer to Chapter 6 for detailed information on function calls.

5.2.1 Physical Format

A disk contains a fixed number of concentric **cylinders**. Each cylinder is made up of one or more **tracks** (one per surface). Each track is subdivided into several fixed-length data blocks called **sectors**. The number of sectors per track and tracks per cylinder varies depending on the media format. Similarly, the size of each sector also varies, but 512 bytes is the most common value.

A sector, as shown in Figure 5–1, is the unit of transfer from the disk to memory (for reading), and from memory to the disk (for writing). It is not possible to transfer a partial sector. However, it is possible to transfer more than one sector, if they are contiguous (next to each other in order by logical sector number).

IFX refers to sectors by their **sector number**, which is an integer from zero to the total number of sectors minus one. Except during track-by-track formatting, IFX does not deal with tracks or cylinders, and makes no assumptions about the order of sectors on the disk. This is considered the responsibility of the disk device driver. For this reason, IFX is compatible with media that do not have a uniform number of sectors per track, such as optical disks.



SYNC	ID	TRACK	SECTOR	CHECK-SUM	GAP	SYNC	512 BYTES OF DATA	CHECK-SUM	GAP

Figure 5–1 Physical Disk Format

During track-by-track formatting, IFX refers to cylinders by their cylinder number, which is an integer from zero to the total number of cylinders minus one. Similarly, IFX refers to tracks by their track number, which is an integer from zero to the total number of tracks per surface minus one.

Disk drive and controller manufacturers may use the term *sector number* to refer to the numbering of sectors within a given track. In this context, a sector number is an integer from one to the number of sectors per track. To avoid confusion, we refer to this type of sector number as a **physical sector number**. A sector number as defined in the previous paragraph is refered to as a **logical sector number**. Unless otherwise stated, Ready Systems documentation uses only logical sector numbers.

Apply the equations below (for devices that have uniform number of sectors per track only) to convert from logical sector numbers to physical sector, track, and cylinder numbers:

$$\mathit{cylinder} = \frac{\mathit{logical\ sector}}{\mathit{total\ cylinders}}$$

$$\mathit{track} = \frac{\mathit{logical\ sector}\ \mathrm{mod}\ (\mathit{sectors\ per\ track\ x\ tracks\ per\ cylinder})}{\mathit{sectors\ per\ track}}$$

$$\mathit{physical\ sector} = (\mathit{logical\ sector}\ \mathrm{mod}\ \mathit{sectors\ per\ track}) + 1$$

Table 5–1 illustrates these equations for a 5¼-inch diskette with 9 sectors per track, 2 tracks per cylinder, and 40 cylinders.

Table 5–1 Sector Number Conversion

Logical Sector	Physical Sector	Track	Cylinder
0	1	0	0
1	2	0	0
. . .	. . .	0	0
8	9	0	0
9	1	1	0
. . .	. . .	1	0
17	9	1	0
18	1	0	1
. . .	. . .	. . .	. . .
719	9	1	39

5.2.2 Logical Format

Application programs normally do not access a disk at the physical level of sectors, tracks, or cylinders. Instead they access the disk at the logical level of files, directories, and volumes. It is the responsibility of the disk driver to implement the physical level, and the responsibility of the IFX volume manager to implement the MS-DOS 4.0 logical level.

5.2.3 Formatting the Disk

As described above, data on the physical media is separated into sectors. The separation between sectors is called the **format** of the disk. The format has these functions:

- Defines the structure of a sector
- Locates the data of a sector
- Compensates for variations in the recording media, drive head, or drive motor

Formatting is the process of writing the format pattern and initial data to each sector of the disk. A distinction is made between **physical formatting** and **logical formatting.** Physical formatting writes the timing information for each sector onto the disk, as opposed to logical formatting, which imposes a logical structure of files and directories.

Physical formatting destroys any data previously stored in each sector by overwriting it with a meaningless pattern, while logical formatting merely makes that data inaccessible by name. If you were to compare a disk to an audio cassette tape, then physical formatting of a disk is equivalent to erasing the whole tape in a bulk eraser machine. Logical formatting is analogous to changing the label that indicates the songs on the tape.

It is the responsibility of the disk driver to do physical formatting only. Logical formatting is the responsibility of higher-level software, such as the volume manager.

Many disk controllers have the ability to format an entire disk in a single operation. For this reason, IFX first calls the driver with the control code IFXOFMTDSK to format a disk. If the driver returns IFXENOTIMP, indicating that the controller cannot format the whole disk at once, then IFX assumes that the controller supports track-by-track formatting. It then calls the the driver repeatedly with the control code IFXOFMTTRK, passing the cylinder and track numbers each time.

5.2.4 Byte-Swapping

At the physical media level, each disk sector contains a fixed number of 8-bit bytes. However, at the logical level, it is necessary to combine several bytes to represent 16-bit and 32-bit integers. There are two standard representations for integers as byte sequences, known as *big-endian* and *little-endian*. The big-endian order is used by Motorola processors, while little-endian is used by Intel processors.

Since MS-DOS disk media was originally used only by the Intel 8086 processor, the MS-DOS logical media format contains integers in the little-endian order. When accessed by a Motorola processor, it is necessary to swap the bytes to big-endian order.

However, your 68000 family disk device driver should *not* swap bytes to compensate for the different byte ordering. The IFX/68000 file manager is designed to do all the necessary byte-swapping. In general, your disk device driver should not do any byte-swapping at all.

The following value types are swapped by the IFX/68000 file manager:

- Directory entry date, time, file length, and starting cluster
- File Allocation Table (FAT) entries
- BIOS Parameter Block (BPB) field
- Master boot sector partition table fields

Data files are not swapped by IFX. Therefore, if you are transporting only ASCII data files between a PC and a 68000 system, you will have no problem. If any files contain binary integer data, then byte-swapping may need to be done at the application level (that is, after the *ifx_read* and before the *ifx_write*). Do not byte-swap in the device driver because that would swap all bytes, including those values that are not supposed to be swapped.

There is one exception to the rule for disk device driver byte-swapping: If your disk controller swaps bytes automatically in the hardware, then your device driver must unswap the bytes to restore them to the correct order. Virtually no disk controllers have this feature, but we know of at least one disk controller that does do this. Therefore, please check your disk controller reference manual to make sure that it doesn't do the swapping.

5.2.5 Sector Interleaving

Normally, the physical sector numbers (as defined in Section 5.2.1) are assigned in increasing numerical order, as shown in Figure 5–2 below.

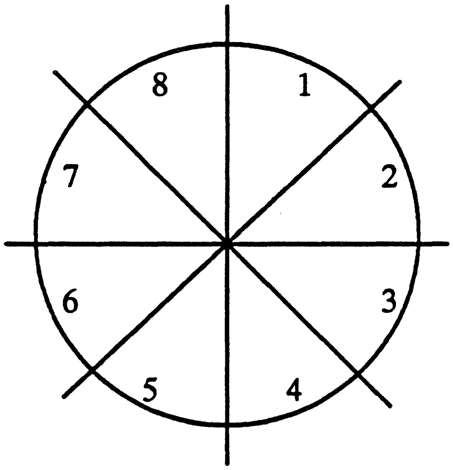


Figure 5–2 Interleave Factor 1:1

Sector interleaving is a convention for assigning physical sector numbers in an order that skips alternating sectors. The **interleave factor** is a ratio expressed as N:M, where N indicates how many physical sector numbers are assigned after every M sectors of rotation. For example, the disk in Figure 5–2 has an interleave factor of 1:1, or no interleaving. The disk in Figure 5–3 has an interleave factor of 1:2. The sector interleaving is chosen during formatting.

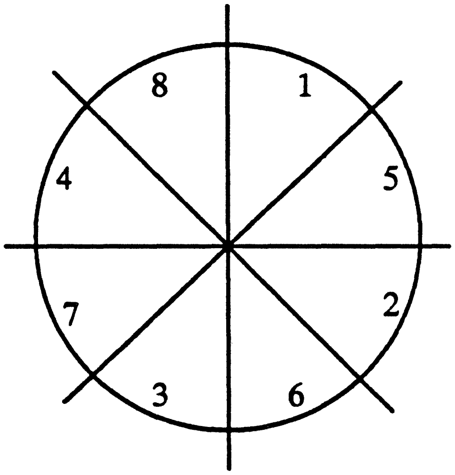


Figure 5–3 Interleave Factor 1:2

The disk in Figure 5–4 below has an interleave factor of 2:4.

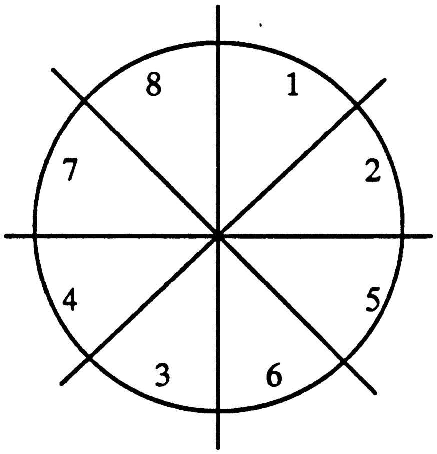


Figure 5–4 Interleave Factor 2:4

The mean time to read one sector on either of these disks is ½ revolution (to wait for the sector to arrive under the head), plus ⅛ revolution (to transfer the data). Now suppose that the application reads one sector and processes the data, which takes a few milliseconds, then reads the next sequential sector. On the disk with the 1:1 interleave factor, by the time the application is ready to read the second sector, the beginning of that sector has already passed by the head. Therefore, the application must wait for another full revolution before the second sector arrives under the head again. The mean time to read two sectors in this way is: ½ + ⅛ + 1 + ⅛, or 1¾ revolutions.

Compare this with a disk that has a 1:2 interleave factor. The time to read the second sector after reading the first sector is ⅛ + ⅛ revolutions, giving a mean time to read both sectors of only ½ + ⅛ + ⅛ + ⅛, or ⅞ revolutions. This is twice as fast!

From this discussion, it might seem that interleaving is the best way to improve your disk performance. A long time ago when memory was expensive, this was true. But now memory is relatively cheap, and so it is better to speed up your disk I/O through **buffering**. The IFX disk buffer cache manager reduces the number of disk operations by reading and writing more sectors than your applications ask for. Later, when the application asks for the next sector, it is taken from the buffer instead of having to do another disk operation. The reduced number of disk accesses more than compensates for the time wasted due to transferring unnecessary sectors.

For almost all applications, the disk buffer cache performs much better than interleaving. Only for systems that are extremely short on memory should you choose an interleave factor other than 1:1.

5.2.6 Multiple Sector Transfers

IFX assumes that the disk device driver can transfer an unlimited number of contiguous sectors in a single operation. Some floppy disk controllers can transfer up to one track per operation, while some SCSI controllers can transfer up to 256 sectors at once. If your disk controller has limitations such as these, then your device driver should contain a loop to do as many separate transfers as necessary, in order to satisfy the request. Be sure to update the actual count after each transfer, so that if an I/O error occurs, the actual count correctly reflects the number of sectors that were successfully transferred before the I/O error.

5.3 Serial Device Drivers

This section explains how to write a serial device driver. The serial device driver, supplied by either you or Ready Systems, implements low-level hardware-specific serial I/O operations, such as initialization and receiver/transmitter interrupts.

A serial device driver is different from other kinds of IFX device drivers because it contains several parts.

- The **device driver** proper is a routine that IFX only calls for device installation, device removal, and I/O control operations. The device driver also includes the **transmitter driver**, a routine that IFX calls to transmit the first character of a given output stream. IFX does not call the device driver routine for normal I/O transfers, except for the initial call to the transmitter driver to send the first character.

- The **receiver Interrupt Service Routine (ISR)** is an interrupt service routine that handles interrupts for incoming characters.

- The **transmitter Interrupt Service Routine (ISR)** is an interrupt service routine that handles interrupts when the transmitter is ready to accept another character.

This section discusses the serial I/O process, the Serial Control Block used by the serial device ISRs, ISR calling conventions, and the serial device driver proper.

5.3.1 Serial I/O

The IFX Circular Buffer Manager maintains separate first-in-first-out (FIFO) buffers for reads and writes from a serial device. The *ifx_write* function places characters into the output buffer. When the buffer is full, application tasks suspend until there is room in the buffer. The *ifx_read* function retrieves characters from the input buffer. When the buffer is empty, application tasks suspend until there is a character in the buffer. The default size of each FIFO buffer is 64 bytes, but this can be changed when the device is installed. Due to the way the buffers are implemented, 1-byte is reserved, so actually there are only 63 buffer bytes available for use.

ISRs call two routines within IFX to pass characters, one at a time, to and from the buffers. These two routines are called **serial_receive_character** and **serial_transmit_ready**. The routines are called by a direct procedure call instruction (such as JSR or CALL) to avoid the overhead of going through the VRTX32 component routing, which uses a software trap instruction such as TRAP or INT. The parameters are passed in registers instead of on the stack. This makes these routines extremely fast.

The receiver ISR calls *serial_receive_character* to transfer each character to the input buffer as the character is received from the USART. If an application task is waiting for a character during an *ifx_read*, the received character is transferred directly to the application buffer. Otherwise, the character is placed at the end of the input (type-ahead) buffer. If this buffer is full, error code IFXEBUFFULL is returned.

When the USART generates a transmit-ready interrupt, the transmitter ISR is invoked. This ISR calls *serial_transmit_ready* to tell IFX that the device is ready. If there are characters in the output buffer, the next one is returned to the ISR. The ISR can output the character directly, or it can call the transmitter driver routine to output the character. (The transmitter ISR does not have to use the transmitter driver, but it is recommended for well-structured programs.) The transmitter driver transmits the character to the USART, enables interrupts from the USART if they were disabled, then returns.

If the output buffer is empty, *serial_transmit_ready* returns the IFXEBUFEMPTY error code to the transmitter ISR. IFX notes the ready status of the USART. When the ISR sees the error code, it does not call the transmitter driver routine, but simply exits. The next time *ifx_write* puts a character into the empty output buffer and the device is ready, IFX calls the transmitter driver routine directly to transmit the character.

5.3.2 Serial Control Block (IFXSCB)

Because the IFX serial calls are designed to be called from high-speed interrupt service routines, the overhead associated with a device name lookup would be unacceptable. To increase performance, you specify the device by the address of the internal IFX data structure called the **Serial Control Block (IFXSCB)**, rather than by its name. IFX passes the IFXSCB address to the serial device driver when the device is installed. The driver should save the IFXSCB address in a global variable that is accessible to the interrupt service routines.

The IFXSCB looks like this:

```
typedef struct {
    void (*serial_receive_character)();
    int (*serial_transmit_ready)();
} IFXSCB;
```

The fields serial_receive_character and serial_transmit_ready contain the addresses of the corresponding IFX routines. In the C declaration of the IFXSCB, these fields are shown as being C function pointers. However, these routines can be called only from assembly language due to their special calling conventions. Refer to the *ifxvisi.inc* file on your shipping media for an assembly language definition of the IFXSCB.

A serial device driver can report I/O errors up to the application level as follows: To cause an error code to be returned from *ifx_read*, the driver should set the long word at offset 0x14 within the IFXSCB to that error code. The error code is then returned to the current *ifx_read* request along with a short actual count. If there is no current *ifx_read* request, then the error code is returned to the next *ifx_read* request along with a zero actual count. Similarly, to cause an error code to be returned from *ifx_write*, the driver should set the long word at offset 0x7C within the IFXSCB to that error code. This method of reporting errors up to the application level is likely to change in the next version of IFX/68000. Therefore, you should isolate any code that relies on this technique, so that it can be easily changed in the future.

5.3.3 ISR Operation

For higher performance, the conventions of the receiver ISR and transmitter ISR are different than those of the device driver.

The **receiver ISR** is invoked by an interrupt when the device receives a character. It should take the character from the device, then pass the character to IFX by calling IFX's *serial_receive_character* routine.

- **M68000 family processors.** Before calling the *serial_receive_character* routine, the receiver ISR should set register A0 to point to the IFXSCB, and D1[7:0] to the character that was just received. Upon return, D0 contains either RET_OK or IFXEBUFFULL, if the type-ahead input buffer is full. In the latter case, the character passed in D1[7:0] is discarded. The *serial_receive_character* routine destroys registers A0–A2/D0–D2.

- **iAPX86 processors.** The receiver ISR should set ES:BX to point to the IFXSCB, and CH to the character that was received. The *serial_receive_characte*r routine destroys registers AX/BX/CX/DX/SI/DI/ES.

The **transmitter ISR** is invoked by an interrupt when the device is ready to transmit another character. It should get the next character from IFX by calling IFX's *serial_transmit_ready* routine. If a character is available for transmission, it should transmit the character to the device.

- **M68000 family processors.** Before calling the *serial_transmit_ready* routine, the transmitter ISR should set register A0 to point to the IFXSCB. Upon return, either D0 contains RET_OK, and register D1[7:0] contains the character to be transmitted, or D0 has the error code IFXEEBUFEMPTY. There are two reasons why status code IFXEBUFEMPTY may be returned: Either the transmit buffer is empty, or the buffer is not empty, yet transmission is temporarily disabled because an XOFF (Control S) was received. The *serial_transmit_ready* routine destroys registers A0–A2/D0–D2.

- **iAPX86 processors.** The ISR should set ES:BX to point to the IFXSCB. Upon return, AX contains the status code and CH contains the character to be transmitted. The *serial_transmit_ready* routine destroys registers AX/BX/CX/DX/SI/DI/ES.

5.4 Clock Device Drivers

This section explains how to write a clock device driver. IFX calls the clock device driver periodically to determine the current date and time, or to change the date and time.

The time and date are stored in the IFXTIME data structure, shown below:

```
typedef struct IFXTIME {
   unsigned char   hour;      /* 0 to 23 */
   unsigned char   minute;    /* 0 to 59 */
   unsigned char   second;    /* 0 to 59 */
   unsigned char   year;      /* 0 = 1900 */
   unsigned char   month;     /* 1 to 12 */
   unsigned char   day;       /* 1 to 31 */
   short           reserved;  /* reserved */
} IFXTIME;
```

Clock device drivers should observe these properties of the IFXTIME structure:

- The hour is in the range 0 to 23 (military format), not 0 to 11 (AM/PM format).
- The year is in the range 0 to 199, corresponding to 1900 to 2099.
- The driver must handle February 29 during leap years. For the range 1901 to 2099, a simple test for leap year is "(year & 3) = = 0".
- The driver is not responsible for keeping track of the day of the week (Sunday, Monday, etc.).

When the application calls *ifx_stime*, IFX first checks the time for validity. If the time is invalid, IFX returns error code IFXEBADTIME to the application without calling the driver. Likewise, when the application calls *ifx_gtime*, IFX calls the driver with function code IFXFGTIME, then checks the time that the driver returned for validity. If the time is invalid, IFX returns error code IFXEBADTIME to the application.

5.5 Custom and Block Device Drivers

This section summarizes how to write a driver for a device type other than disk, serial, clock, manager, or pathname (volume). This information also applies to a serial driver that does not use the Circular Buffer Manager. Examples of such devices include printers, tape drives, etc.

Your driver only needs to implement five function codes: IFXFINSTALL, IFXFREMOVE, IFXFREAD, IFXFWRITE, and IFXFIOCTL. For all other function codes, the driver should return error code IFXENOTIMP. The driver should handle IFXFINSTALL and IFXFREMOVE in the normal way. When the driver is called with IFXFREAD or IFXFWRITE function codes, it should transfer the data between the application program's buffer and the device buffer. (You should consult your device controller hardware manual for information about how to do this, since it is very device-specific.) When the driver is called with the IFXFIOCTL (I/O control) function code, the driver should examine the first parameter, which is an integer control code. This control code indicates which control operation should be performed. You can add new control operations that are device-specific, such as "rewind tape" or "skip file." These control operation codes should have the form "0xFFXX" to avoid overlapping with standard control codes.

5.6 Device Managers

A device manager does not access hardware devices directly. Instead, it converts application I/O requests to simpler requests and passes these on to another device driver to do the real work.

5.6.1 Installation

Device managers are installed with the *ifx_mount* system call rather than *ifx_install*, as shown below:

```
extern int device_manager();

status = ifx_driver("MNGRDRVR", device_manager);
status = ifx_mount("manager:", "device:", "MNGRDRVR", parameters ...);
```

In this example, the underlying device is called *device:*, the managing device is called *manager:*, and the device manager is called *MNGRDRVR*. Additional parameters to *ifx_mount* are ignored by IFX, but are passed on to the device manager.

5.6.2 Calling the Underlying Driver

Two fields in the IFXDCB data structure are of particular interest to a device manager: mounted_on and device_driver. The mounted_on field contains a pointer to the IFXDCB of the underlying device. This pointer can be used to locate the address of the underlying device driver, device_driver. Then it is a simple matter to call the underlying driver by following the same calling conventions described in Chapter 2.

Example 5–1 shows a device manager that converts an application *ifx_write* request to a call to the underlying driver to write a sector to disk.

```
int device_manager(func_code, dcb_ptr, plist)
int func_code;
IFXDCB *dcb_ptr;
PL *plist;
{
    struct {
        char *buffer_address;
        long sector_number;
        long number_of_sectors;
        long *actual_count;
    } params;
    long actual;

    switch (func_code) {
    case IFXFWRITE:
        params.buffer_address = ...;
        params.sector_number = ...;
        params.number_of_sectors = 1L;
        params.actual_count = &actual_count;
        status = (*dcb_ptr->mounted_on->device_driver)
                    (IFXFWRITES, dcb_ptr->mounted_on, &params);
        break;
    }
    return status;
}
```

Example 5–1 Converting Application Write Calls

In Example 5–1, the parameter list *params*, which is passed to the device driver, is identical to the parameter list defined under IFXFWRITES in Chapter 6. The underlying device driver is called through an indirect C function call. The second parameter to the driver is its own IFXDCB pointer.

It is possible to write a device manager entirely in C, provided that the calling conventions of the C compiler are identical to those given in Chapter 2. For other compilers, you will need to write a small assembly language routine that converts the calling conventions of your compiler to those of IFX.

Appendix A contains an example device manager that maintains statistics about a disk device.

5.7 Pathname Driver

A **pathname driver** is a device driver or manager that implements the *pathname* function codes. These function codes are characterized by their first parameter, which is always a NULL-terminated ASCII string. By the time the driver is called with a pathname function code, the IFX dispatcher has already stripped off any leading device name from the pathname. For example, for both of the following calls to *ifx_create*, the driver is called with function code IFXFCREATE and pathname equal to the string *file*:

```
status = ifx_sdefault("device:");
status = ifx_create("file");
status = ifx_create("device:file");
```

5.7.1 Implementing IFXFOPEN

The implementation of all pathname function codes, except for IFXFOPEN, is fairly straightforward. However, the correct implementation of IFXFOPEN requires some additional discussion.

When a pathname driver is called with function code IFXFOPEN, the driver should call *ifx_attach* to allocate a descriptor ID. Consult the *IFX User's Guide* for the formal declaration of *ifx_attach*. The parameters to *ifx_attach* are:

I/O operation handler — The address of a subroutine that IFX calls to handle descriptor operations such as IFXFREAD, IFXFWRITE, and IFXFIOCTL.

Access mode — The access mode for the new descriptor.

Descriptor ID — The new descriptor ID is returned indirectly through this reference parameter.

Descriptor control block — The address of the descriptor control block (IFXCCB) is returned indirectly through this reference parameter.

If *ifx_attach* returns an error code, then the pathname driver should abort the open operation and return the error code back to the caller. If *ifx_attach* returns RET_OK, then the descriptor has been partially created in an embryo state. This means that the descriptor ID number is assigned, but attempts to do I/O operations on this descriptor fails with error code IFXEDESCNOPEN, as if the descriptor were not open.

To complete the initialization of the descriptor, the pathname driver must set two fields in the IFXCCB (refer to Section 5.7.3 for more information on IFXCCB):

- Set the u field of IFXCCB to point to a driver-defined data structure that identifies this descriptor.

- Set the ready field of IFXCCB to 1.

Now the descriptor creation is complete and the descriptor is ready for use.

5.7.2 I/O Handler

If the application performs an I/O operation on the descriptor, IFX calls the descriptor I/O handler with these parameters:

```
status = IO_handler(func_code, ccb_ptr, plist);
```

The I/O handler routine should be declared as follows:

```
int IO_handler(func_code, ccb_ptr, plist)
int func_code;
IFXCCB *ccb_ptr;
PL *plist;
{
    int status;

    switch (func_code) {
    case IFXFREAD:
        ...
        break;
    case IFXFWRITE:
        ...
        break;
    case IFXFIOCTL:
        ...
        break;
    default:
        status = IFXENOTIMP;
        break;
    }
    return status;
}
```

In many ways, an I/O handler routine looks almost identical to a device driver. It implements the same function codes, same parameter lists, and same calling conventions. The only difference is that the second parameter is a pointer to an IFXCCB, rather than to an IFXDCB.

5.7.3 Descriptor Control Block

The **Descriptor Control Block (IFXCCB)** is a data structure that is associated with each descriptor. There is one IFXCCB per descriptor. The IFXCCB is allocated by IFX when the descriptor is created with *ifx_attach*. The IFXCCB is released by IFX when the descriptor is deleted with *ifx_close*. A pointer to the IFXCCB is passed to an I/O handler as its second parameter. The IFXCCB contains a number of fields, some of which are undocumented and reserved for use by IFX, and others that are documented and of interest to an I/O handler. Your I/O handler should not refer to the undocumented fields, since they are subject to change. The documented fields are:

access_mode — The access mode of the descriptor, as specified in *ifx_open*. The *ifx_attach* call sets this field and the I/O handler should not change it thereafter.

current_position — The current position of the descriptor, as set by *ifx_sposn*.

u — This is a general-purpose field that is reserved for use by the I/O handler. The field is the size of a pointer (4 bytes for 68000 and 8086, 6 bytes for 80386). The field typically points to a data structure used by the I/O handler. The initial value is NULL (zero).

ready — This is a Boolean field that indicates whether I/O operations are allowed on the descriptor. The initial value is FALSE (zero). The I/O handler should set it to TRUE (one) when it is ready to accept I/O operations.

```
/* Descriptor Control Block */

typedef struct IFXCCB {
   long            reserved1[3];      /* reserved for use by Ready Systems */
   unsigned short  access_mode;       /* access mode option bits */
   char            reserved2[2];      /* reserved for use by Ready Systems */
   long            current_position;  /* position in units of bytes */
   char           *u;                 /* descriptor specific information */
   long            reserved3;         /* reserved for use by Ready Systems */
   unsigned char   ready;             /* whether descriptor is ready for I/O */
   char            reserved4[3];      /* reserved for use by Ready Systems */
} IFXCCB;
```

Chapter 6

Supporting Function Codes

6.1 Introduction

This chapter lists the functions that a device driver should support. When IFX calls the driver, it passes a function code, a pointer to the Device Control Block (IFXDCB), and a pointer to the parameter list.

Function codes are classified as either generic or device-specific. **Generic function codes** are applicable to all device types. **Device-specific function codes** are unique to one device type.

Within these two categories, there are required function codes and optional function codes. **Required function codes** must be implemented by the driver, but **optional function codes** may be ignored. If the driver chooses to ignore an optional function code, it should return the error code IFXENOTIMP.

The IFXDCB is a data structure that is associated with each IFX device. Refer to Chapter 2 for details about IFXDCB.

The parameter list supplies necessary data to the device driver that is specific to each function. The parameter list varies for each function code and so it is expressed as a C union:

```
union {
    struct {
        ...
    } u1;
    struct {
        ...
    } u2;
    ...
} parameter_list;
```

The parameter list for each function is defined within the discussion of that function.

6.2 Generic Function Codes

The function codes below must be supported by all device drivers. This section covers general information that is applicable to all device types. Device-specific information can be found in Sections 6.3 through 6.5.

6.2.1 IFXFDRIVER (Install Driver)

The parameter list points to the third parameter of *ifx_driver*. For example, if the driver is installed this way:

```
status = ifx_driver("DRIVER", device_driver, 0xFFFFC040L, 2.0);
```

then the parameter list is:

```
struct {
    long p1;      /* 0xFFFFC040L */
    double p2;    /* 2.0 */
};
```

This allows any driver-specific parameters to be put at the end of the *ifx_driver* call. These extra parameters are not used by *ifx_driver* itself, but can be used by the device driver.

As there are no devices installed yet when the driver is called with this function code, the IFXDCB data structure cannot be a real one. Instead, IFX constructs a temporary dummy IFXDCB whose only valid field is my_driver. This field points to the IFXDDCB of the driver.

The device driver should:

1. Initialize data structures shared by all devices serviced by this driver
2. Set the IFXDDCB ddt field to point to global variables used by the driver
3. Return RET_OK, IFXENOTIMP, or an error code

If the driver returns an error code other than IFXENOTIMP, then the driver is not installed.

6.2.2 IFXFRMDRIVER (Remove Driver)

There is no parameter list.

As there are no longer any devices installed when the driver is called with this function code, the IFXDCB data structure cannot be a real one. Instead, IFX constructs a temporary dummy IFXDCB whose only valid field is my_driver. This field points to the IFXDDCB of the driver.

The device driver should:

1. De-initialize data structures shared by all devices serviced by this driver
2. Return RET_OK, IFXENOTIMP, or another error code

By the time the driver is called with function code IFXFRMDRIVER, it is too late for the driver to prevent itself from being removed. If the driver returns an error code other than IFXENOTIMP, then the driver is removed, but the error code is still passed back to the application.

6.2.3 IFXFINSTALL (Install Device)

The parameter list points to the third parameter of *ifx_install.* For example, if the device is installed this way:

```
status = ifx_install("device:", "DRIVER", 12345, "hello");
```

then the parameter list is:

```
struct {
    int p1;       /* 12345 */
    char *p2;     /* "hello" */
};
```

This allows any driver-specific parameters to be put at the end of the *ifx_install* call. These extra parameters are not used by *ifx_install* itself, but can be used by the device driver.

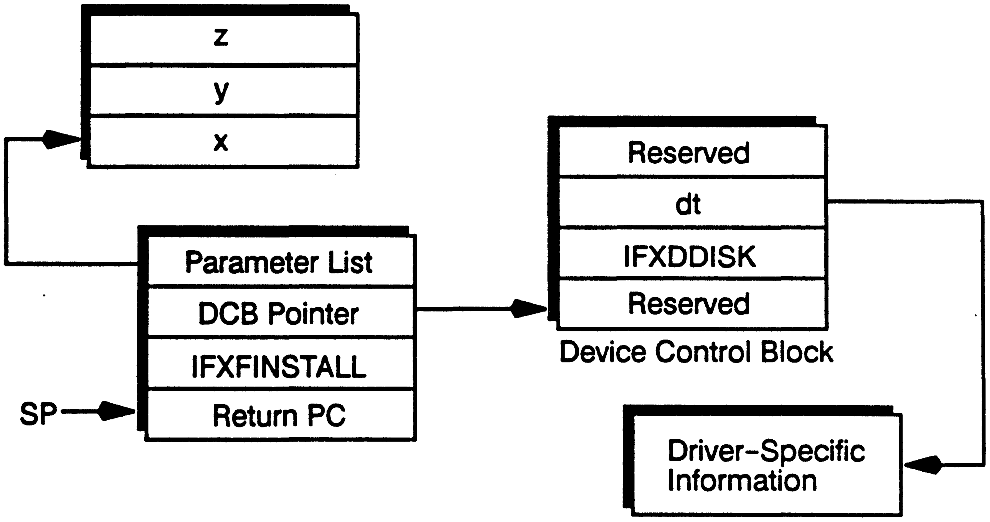


Figure 6-1 Installing a Device with IFXFINSTALL

The device driver should:

1. Initialize the device
2. Set up interrupt vectors
3. Set the IFXDCB device_type field to one of the codes in Table 6–1
4. Set the IFXDCB dt field to point to global variables used by the driver
5. Return RET_OK or an error code

If the driver returns an error code other than RET_OK or IFXENOTIMP, then the driver is not installed.

Table 6–1 Device Codes

Device Type	Device Code
Disk	IFXDDISK
Serial	IFXDSERIAL
Clock	IFXDCLOCK
Other	0xF0 to 0xFF

6.2.4 IFXFREMOVE (Remove Device)

There is no parameter list.

The device driver should:

1. De-initialize the device
2. Restore interrupt vectors
3. Return RET_OK, IFXENOTIMP, or another error code

By the time the driver is called with function code IFXFREMOVE, it is too late for the driver to prevent the device from being removed. If the driver returns an error code other than IFXENOTIMP, then the device is removed, but the error code is still passed back to the application.

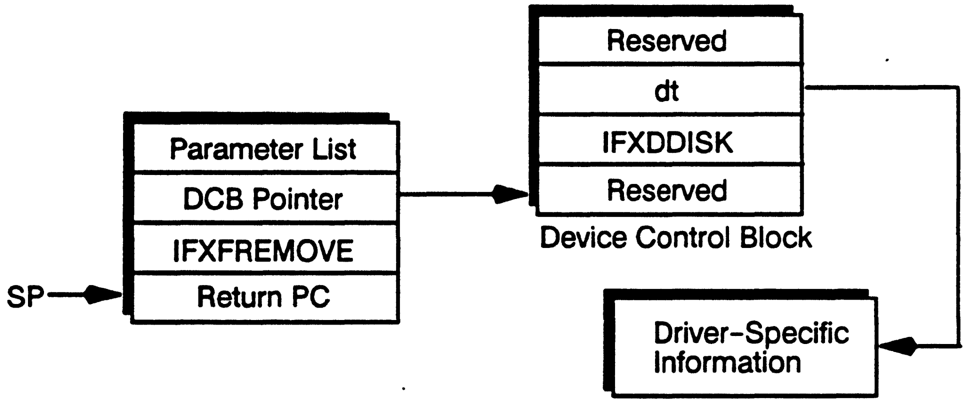


Figure 6-2 Removing a Device with IFXFREMOVE

6.2.5 IFXFIOCTL (I/O Control)

The parameter list is:

```
struct {
    int control_code;
    /* additional parameters here */
};
```

Control codes are classified as either generic or device-specific. Generic control codes are applicable to all device types, while device-specific control codes are unique to one device type.

Within these two categories, there are required control codes and optional control codes. Required control codes must be implemented by the driver, but optional control codes may be ignored. If the driver chooses to ignore an optional control code, it should return the error code IFXENOTIMP.

Following the control code are additional parameters, which supply necessary data to the device driver that is specific to each control operation. The additional parameters vary for each control code, so they are expressed as a C union:

```
struct {
    int control_code;
    union {
        struct {
            ...
        } u1;
        struct {
            ...
        } u2;
        ...
    } additional_parameters;
} parameter_list;
```

The additional parameters for each control operation are defined within the discussion of that operation.

Many I/O control operations are implemented by the device driver itself. However, there are a few control operations that are implemented by higher-level device managers or by the IFX dispatcher. Table 6–2 shows the standard control operations and which layer is responsible for implementing each. Optional operations are marked with an asterisk.

6.2.6 IFXFDEVCTL (Device Control)

The parameter list is:

```
struct {
    char *path_name;
    int control_code;
    /* additional parameters here */
};
```

It is not necessary for the device driver to implement the IFXFDEVCTL operation. The IFX dispatcher automatically translates IFXFDEVCTL to the corresponding IFXOIOCTL operations. The driver should return IFXENOTIMP for this function code.

Table 6–2 I/O Control Operations

Operation	Description	Responsible Layer
IFXOGGEOM	get disk geometry	disk driver
IFXOFLUSHOUT	flush output buffer	disk volume manager disk buffer manager circular buffer manager pipe driver, disk driver *
IFXODISCIN	discard input buffer	disk buffer manager circular buffer manager pipe driver, disk driver *
IFXODISCOUT	discard output buffer	disk buffer manager circular buffer manager pipe driver, disk driver *
IFXOGDVTYPE	get device type	dispatcher
IFXOGMTDVNM	get mounted-on device name	dispatcher
IFXOGOPNCNT	get open count	dispatcher
IFXOGMNTCNT	get mount count	dispatcher
IFXOGVOLCLS	get volume clusters	disk volume manager
IFXOGLEDCHR	get line-editing characters	line-editing manager
IFXOSLEDCHR	set line-editing characters	line-editing manager
IFXOGXNFFT	get XON/XOFF features	circular buffer manager
IFXOSXNFFT	set XON/XOFF features	circular buffer manager
IFXOGINBSIZ	get input buffer size	circular buffer manager pipe driver
IFXOGOUTBSIZ	get output buffer size	circular buffer manager pipe driver
IFXOGLEDFT	get line-editing features	line-editing manager
IFXOSLEDFT	set line-editing features	line-editing manager
IFXOGCOLPOSN	get column position	line-editing manager
IFXOSCOLPOSN	set column position	line-editing manager
IFXOGEOFBSIZ	get end-of-file buffer size	pipe driver
IFXOGLEDBSIZ	get line-editing buffer size	line-editing manager
IFXOGDISKBUF	get disk-buffer cache status	disk buffer manager
IFXOFMTTRK	format track	disk driver*
IFXOTXRDY	transmit character	serial driver
IFXOGSYNCERR	get synchronization errors	disk volume manager
IFXOFMTDSK	format disk	disk driver*

6.2.7 IFXFACANCEL (Asynchronous Cancel)

The parameter list is identical to one passed to a previous I/O function. It is *not* the parameter list of the IFXFACANCEL caller. Thus, if the device driver can support more than one I/O operation at once, the parameter list can be used to determine which operation should be canceled.

IFX may need to call a device driver while there is an I/O operation in progress (for example, canceling an asynchronous I/O). In this case, the three parameters to the device driver are:

1. The device driver should return RET_OK for opcode IFXFACANCEL, even if it cannot determine which I/O request is being canceled.
2. The driver should arrange for the other execution thread that was doing the I/O to receive a short actual transfer count and error code IFXEASCANCEL.
3. The driver should support asynchronous cancellation if the device is slow, meaning that it can take an indefinite time for the device to complete an I/O operation. For such devices, supporting asynchronous cancellation allows the application to recover from the pending I/O operation.

When the driver is called with opcode IFXFACANCEL, it should:

1. Check for I/Os in progress. If no I/O is in progress, then go to the last step.
2. Compare the parameter list address to that of the current I/O operation. If it does not match, then go to the last step. Note that there may be more than one active I/O operation. In this case, the driver should compare the parameter list address to that of each active I/O operation.
3. Attempt to cancel the I/O operation. If the I/O cannot be canceled because it has progressed too far, then go to the last step.
4. Arrange for the driver to return error code IFXEASCANCEL (asynchronous I/O canceled) to the I/O initiator, for the actual transfer count to be less than the desired transfer count.
5. Return RET_OK.

6.3 Supporting Disk Function Codes

The function codes below must be supported by disk device drivers.

6.3.1 IFXFREADS (Read Sectors)

The parameter list is:

```
struct {
    long sector_position;
    char *buffer_address;
    long number_sectors;
    long *actual_count;
};
```

The device driver should:

1. Transfer number_sectors sectors, beginning from disk sector number sector_position, to memory at buffer_address.
2. Set the variable pointed to by actual_count to the number of sectors that were successfully transferred.
3. Return RET_OK, IFXEIOERR, or a more specific I/O error code, such as IFXECRCERR, IFXEIOTIMOUT, IFXESECNTFND, or IFXESEEKFAIL.

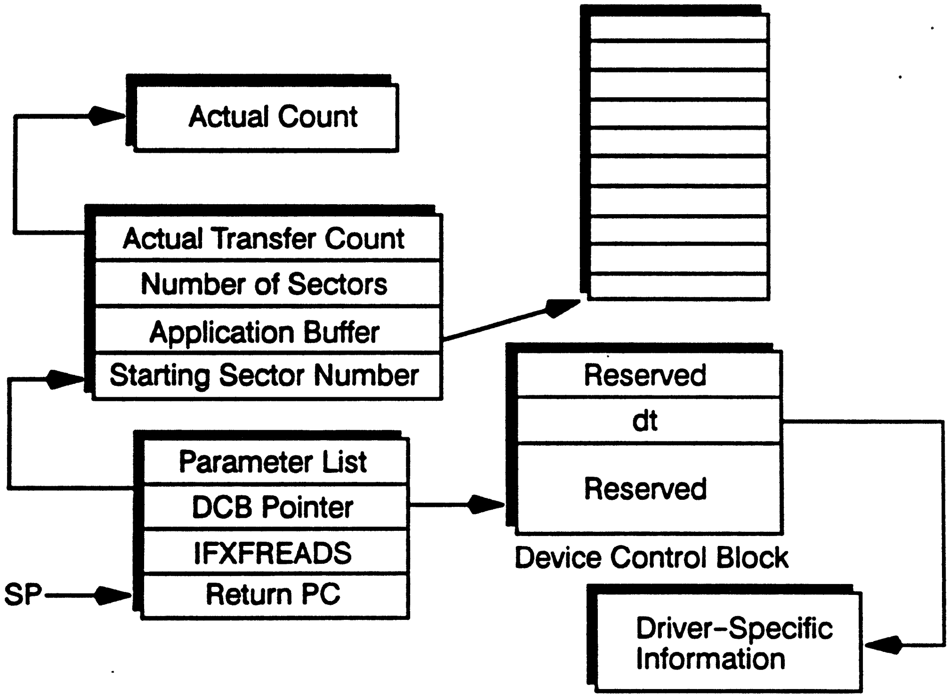


Figure 6–3 Reading Sectors with IFXFREADS

6.3.2 IFXFWRITES (Write Sectors)

The parameter list is:

```
struct {
    long sector_position;
    char *buffer_address;
    long number_sectors;
    long *actual_count;
};
```

The device driver should:

1. Transfer number_sectors sectors, from memory at buffer_address, to disk beginning with sector number sector_position.
2. Set the variable pointed to by actual_count to the number of sectors that were successfully transferred.
3. Return RET_OK, IFXEIOERR, IFXERDONLYM, or a more specific I/O error code, such as IFXECRCERR, IFXEIOTIMOUT, IFXESECNTFND, or IFXESEEKFAIL.

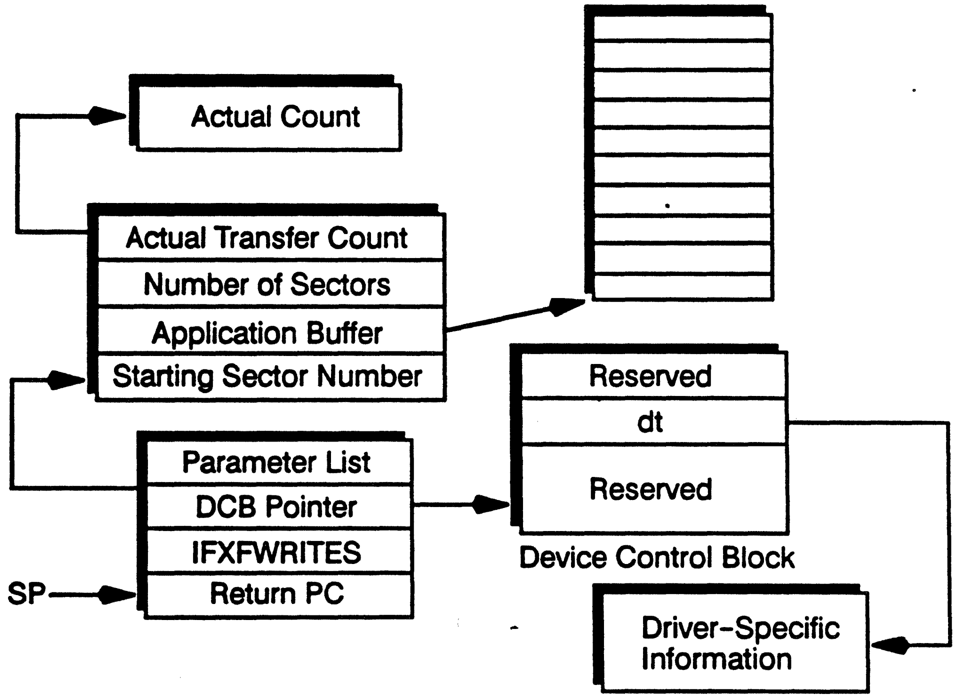


Figure 6–4 Writing Sectors with IFXFWRITES

6.3.3 IFXFIOCTL (I/O Control)

The control codes below must be supported by disk device drivers.

IFXOGGEOM (Get Disk Geometry)

The additional parameters are:

```
struct {
    IFXGEOMETRY *disk_geometry;
};
```

The device driver should:

1. Return information about the physical configuration in the various fields of the disk_geometry structure. This tells IFX what the disk looks like. The disk geometry returned by the driver must satisfy these constraints:

   - sector_size must be a power of 2 between 128 and 32768
   - sectors_per_track must be greater than 0
   - tracks_per_cylinder must be greater than 0
   - total_cylinders must be greater than 0
   - total_sectors must be equal to the product of sectors_per_track, tracks_per_cylinder, and total_cylinders

   The disk geometry should describe the entire disk, including any boot sectors, partition tables, and all partitions. The volume manager will take care of subdividing the disk, if necessary.

2. Return RET_OK

IFXOFMTDSK (Format Disk)

There are no additional parameters.

The device driver should format the entire disk. If the device driver is incapable of formatting the entire disk in one operation, then the driver should return error code IFXENOTIMP. IFX will then call the device driver with control opcode IFXOFMTTRK once per track and cylinder to format the disk a track at a time.

If the device driver is capable of formatting the entire disk in one operation, then the driver should:

1. Format the entire disk, or
2. Return RET_OK, IFXEIOERR, IFXERDONLYM, or a more specific I/O error code

IFXOFMTTRK (Format Track)

The additional parameters are:

```
struct {
    int cylinder;
    int track;
};
```

The device driver should:

1. Format the specified track.
2. Return RET_OK, IFXEIOERR, IFXERDONLYM, or a more specific I/O error code

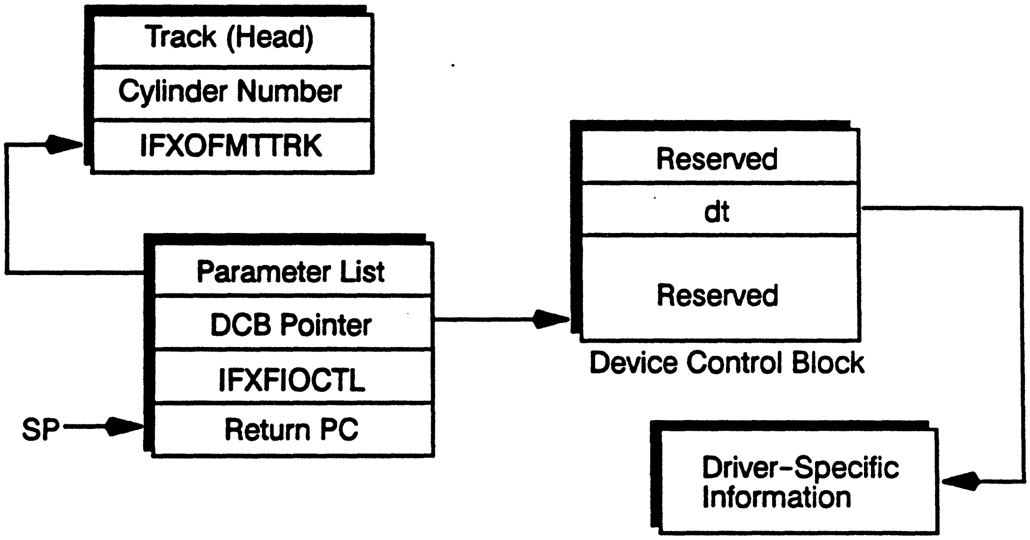


Figure 6–5 Formatting a Track with IFXOFMTTRK

IFXODISCIN (Discard Input Buffer)

There are no additional parameters.

IFXODISCIN tells the driver to discard all buffered data, including dirty buffers. The driver need only implement this operation if the disk controller supports hardware buffering (in addition to IFX's disk buffer cache). The only input parameter is the control operation code.

The device driver should:

1. Discard all buffered data (including dirty buffers), if the controller has hardware buffering.
2. Return RET_OK, if the controller has hardware buffering and the buffers were successfully discarded. Return IFXENOTIMP, if the controller does not have hardware buffering. Or, return an I/O error code, such as IFXEIOERR, IFXERDONLYM, or another I/O error code, if the operation was attempted but not successful.

IFXODISCOUT (Discard Output Buffer)

There are no additional parameters.

IFXODISCOUT tells the driver to discard only dirty buffered data. The driver need only implement this operation if the disk controller supports hardware buffering (in addition to IFX's disk buffer cache). The only input parameter is the control operation code.

The device driver should:

1. Discard all dirty buffers, if the controller has hardware buffering.
2. Return RET_OK, if the controller has hardware buffering and the buffers were successfully discarded. Return IFXENOTIMP, if the controller does not have hardware buffering. Or, return an I/O error code, such as IFXEIOERR, IFXERDONLYM, or another I/O error code, if the operation was attempted but not successful.

IFXOFLUSHOUT (Flush Output Buffer)

There are no additional parameters.

IFXOFLUSHOUT tells the driver to flush all dirty buffered data to the disk. The driver need only implement this operation if the disk controller supports hardware buffering (in addition to IFX's disk buffer cache). The only input parameter is the control operation code.

The device driver should:

1. Flush all dirty buffers to the disk, if the controller has hardware buffering.
2. Return RET_OK, if the controller has hardware buffering and the buffers were successfully discarded. Return IFXENOTIMP, if the controller does not have hardware buffering. Or, return an I/O error code, such as IFXEIOERR, IFXERDONLYM, or another I/O error code, if the operation was attempted but not successful.

6.3.4 IFXFACANCEL (Asynchronous Cancel)

IFX does not call the device driver with the IFXFACANCEL function code, if you access the disk through the Volume Manager or Disk Buffer Cache Manager. The two disk managers support asynchronous I/O, but not asynchronous cancel.

It is not necessary for your driver to implement the IFXFACANCEL opcode for disk device drivers, because modern disk controllers have a time-out mechanism. They either complete an I/O operation or return an error code. They do not hang if the disk is offline or if there is no media in the drive.

For most applications, we recommend that the disk device driver return error code IFXENOTIMP for this opcode.

If you access the disk directly by opening the raw sector-oriented device for asynchronous sector transfers, then you may want to implement asynchronous cancel in the driver.

IFXFACANCEL uses the same parameter list as that used to initiate the I/O for IFXFREADS, IFXFWRITES, or IFXFIOCTL (Sections 6.3.1, 6.3.2, and 6.3.3).

The device driver should:

1. Check if there is currently a IFXFREADS, IFXFWRITES, or IFXFIOCTL operation in progress. If no I/O is in progress, then go to the last step.
2. Compare the parameter list address to that of the current I/O operation. If it does not match, then go to the last step.
3. Attempt to cancel the I/O operation. If the I/O cannot be canceled because it has progressed too far, then go to the last step.
4. Arrange for the IFXFREADS, IFXFWRITES, or IFXFIOCTL to return error code IFXEASCANCEL (asynchronous I/O canceled), and for the actual transfer count to be less than the desired transfer count.
5. Return RET_OK.

6.4 Supporting Serial Function Codes

The function codes below must be supported by serial device drivers.

6.4.1 IFXFINSTALL (Install Device)

The parameter list points to the sixth parameter of *ifx_install*. For example, the device is installed like this:

```
status = ifx_install("serial:", "CIRCULAR",
                device_driver, 64, 64, 12345, "hello");
```

Then the parameter list is:

```
struct {
    int p1;         /* 12345 */
    char *p2;       /* "hello" */
};
```

This allows any driver-specific parameters to be put at the end of the *ifx_install* call. These extra parameters are not used by *ifx_install* itself, but can be used by the device driver.

The device driver should:

1. Initialize the device (reset the USART, set up baud rate generator, initialize USART features, set up interrupt vectors, and so forth).

2. Save the contents of the IFXDCB dt field in a global variable, for later use by interrupt service routines. This is a pointer to the IFXSCB. Do *not* change the value in the dt field. Also, do not set the device_type field. It has already been set to IFXDSERIAL.

3. Return RET_OK.

6.4.2 IFXFIOCTL (I/O Control)

IFX uses the IFXFIOCTL operation to call the device's transmitter driver routine. In addition, the driver can implement other device-specific control operations.

For calls to the transmitter driver, the parameter list is:

```
struct {
    int control_opcode;
    int character;
};
```

The control operation code for the transmit driver is IFXOTXRDY. The device driver should transmit the supplied character to the device (by calling an assembly language routine), then return RET_OK. This is how IFX calls your driver to transmit the very first character, or to transmit the first character after a period of not transmitting characters. It is not intended to be used by an application program.

For other control operations, the parameter list is:

```
struct {
    int control_opcode;
    ...
};
```

The number and type of parameters following control_opcode vary according to the control operation.

IFX does not define any standard control operations that the serial driver has to implement. However, typical operations might get and set the baud rate, number of data bits, number of stop bits, parity, control a modem, and so forth.

The device driver should return IFXENOTIMP if a control opcode is not implemented.

NOTE

You should define new control operation codes in the range of FF00H to FFFFH to distinguish them from future IFX opcodes.

6.4.3 IFXFACANCEL (Asynchronous Cancel)

IFX never calls the device driver with the IFXACANCEL function code. Asynchronous cancel is handled entirely by the Circular Buffer Manager without any assistance from the device driver.

6.5 Supporting Clock Function Codes

The function codes below must be supported by clock device drivers.

6.5.1 IFXFGTIME (Get System Time)

The parameter list is:

```
struct {
    IFXTIME *time;
};
```

The device driver should:

1. Determine the current date and time by reading from the hardware clock.
2. Fill in the fields of the structure pointed to by the IFXTIME parameter.
3. Return RET_OK.

6.5.2 IFXFSTIME (Set System Time)

The parameter list is:

```
struct {
    IFXTIME *time;
};
```

The device driver should:

1. Use the values from the IFXTIME parameter to reset the hardware clock.
2. Return RET_OK.

6.6 Supporting Pathname Function Codes

The function codes below may be supported by pathname device drivers. All function codes are optional except for IFXFOPEN, which is required.

6.6.1 IFXFCREATE (Create File)

The parameter list is:

```
struct {
    char *pathname;
};
```

The device driver should create the specified file and return a status code.

6.6.2 IFXFDELETE (Delete File)

The parameter list is:

```
struct {
    char *pathname;
    int options;
};
```

The possible option bits are:

IFXRMARKBAD	Mark file as bad
IFXRRDONLY	Delete read-only files

The device driver should delete the specified file and return a status code.

6.6.3 IFXFMKDIR (Make Directory)

The parameter list is:

```
struct {
    char *pathname;
};
```

The device driver should create the specified directory and return a status code.

6.6.4 IFXFRMDIR (Remove Directory)

The parameter list is:

```
struct {
    char *pathname;
    int options;
};
```

The possible option bits are:

IFXRRECURS	Remove subdirectories recursively
IFXRMARKBAD	Mark file as bad
IFXRRDONLY	Delete read-only files

The device driver should remove the specified directory, and, possibly, any subdirectories and files within that directory, then return a status code.

6.6.5 IFXFRENAME (Rename File)

The parameter list is:

```
struct {
    char *old_pathname;
    char *new_pathname;
};
```

The device driver should rename the specified file to the new name or directory and return a status code.

6.6.6 IFXFSWKDIR (Set Working Directory)

The parameter list is:

```
struct {
    char *pathname;
};
```

The device driver should check that the specified pathname exists and that it refers to a directory. It then should set the working directory of the calling task to this directory. The calling task can be determined by calling *sc_tinquiry*. After setting the working directory, the driver should return a status code.

6.6.7 IFXFGWKDIR (Get Working Directory)

The parameter list is:

```
struct {
    char *empty_string;
    char *pathname;
};
```

The device driver should check that the first parameter is an empty string. If so, the driver should copy the current working directory of the calling task to the pathname buffer. The calling task can be determined by calling *sc_tinquiry*. After copying the working directory, the driver should return a status code. If the first parameter is not an empty string, the driver should return error code IFXECHRAFTNAM.

6.6.8 IFXFSLABEL (Set Volume Label)

The parameter list is:

```
struct {
    char *empty_string;
    char *label;
};
```

The device driver should check that the first parameter is an empty string. If so, the driver should set the new volume label, then return a status code. If the first parameter is not an empty string, the driver should return error code IFXECHRAFTNAM.

6.6.9 IFXFGLABEL (Get Volume Label)

The parameter list is:

```
struct {
    char *empty_string;
    char *label;
};
```

The device driver should check that the first parameter is an empty string. If so, the driver should copy the current volume label to the label buffer, then return a status code. If the first parameter is not an empty string, the driver should return error code IFXECHRAFTNAM.

6.6.10 IFXFMARKBAD (Mark Bad Sectors)

The parameter list is:

```
struct {
    char *empty_string;
    long start_sector;
    long number_of_sectors;
};
```

The device driver should check that the first parameter is an empty string. If so, the driver should mark the specified range of sectors as unusable, then return a status code. If the first parameter is not an empty string, the driver should return error code IFXECHRAFTNAM.

6.6.11 IFXFOFFLINE (Mark Device Off-Line)

The parameter list is:

```
struct {
    char *empty_string;
    int error_code;
};
```

The device driver should check that the first parameter is an empty string. If so, the driver should mark the device as off-line, so that future calls to the driver return the specified error code. If the first parameter is not an empty string, the driver should return error code IFXECHRAFTNAM.

6.6.12 IFXFDEVCTL (Device Control)

The parameter list is:

```
struct {
    char *empty_string;
    int control_code;
    /* additional parameters here */
};
```

The device driver should return error code IFXENOTIMP. Then IFX will call the driver again with function code IFXFIOCTL and the same parameter list. In this way, the driver only needs to implement IFXFIOCTL, but the application can call both *ifx_ioctl* and *ifx_devctl.*

6.6.13 IFXFOPEN (Open)

The parameter list is:

```
struct {
    char *pathname;
    int access_mode;
    int *descriptor;
};
```

The device driver should:

1. Check that the specified pathname exists and that the access mode is compatible with this file.
2. Allocate a descriptor by calling *ifx_attach*.
3. Set the u field in IFXCCB to point to a driver-defined data structure that identifies this descriptor.
4. Set the ready field in IFXCCB to 1.
5. Return a status code.

Chapter 7

Installing IFX Devices

7.1 Introduction

Ready Systems provides a collection of device drivers and device managers that you can use with IFX. Some of the device drivers and managers are built-in to the IFX component, while others are samples that are included on your shipping media to be used as templates in creating your own device drivers.

This chapter explains how to install the device drivers included in the IFX package, as well as drivers that you may write.

To get started quickly using IFX devices, use the example routine provided in Appendix D of the *IFX User's Guide*. This routine installs all the IFX standard devices. Then, for more details, read this chapter.

For two common device types, there are additional software layers between the application and the device driver. These are called **device managers**. A device manager looks like a device driver to IFX. However, when a device manager is called by IFX to perform a low-level operation, the manager examines the parameters and calls another device driver or manager to actually do the data transfer.

IFX includes the following device managers:

MS-DOS File Manager	Maintains disks in MS-DOS-compatible format.
Disk Buffer Cache	Reduces the number of disk I/O operations by keeping copies of frequently used disk sectors in memory. It also converts byte-oriented data transfers to sector transfers.
Line Editor	Handles echoing, erase-character, erase-line, and related features for CRT terminals.
Circular Buffer Manager	Performs circular buffering for serial devices, such as USARTs, and supports the XON/XOFF flow control protocol.

Some of these device managers make use of one another to do their work. The MS-DOS Volume Manager requires the disk buffer cache or the byte-oriented RAM disk. The line editor requires the circular buffer manager or IFX's standard console device. The disk buffer cache requires a disk driver, and the Circular Buffer Manager requires a serial driver. Ultimately, all device managers end up calling a device driver.

Before installing a device manager, you must install the device driver. The parameters for installing a device driver vary depending on the driver. Refer to Section 7.7 for information on installing the IFX standard device drivers.

The rest of this chapter contains C code that refers to standard IFX device drivers and managers by name.

7.2 Installing and Removing Device Drivers and Managers

IFX includes several built-in device drivers and managers, which are listed below:

Driver name	Description
DISKBUF	Disk Buffer Cache Manager
VOLUME	MS-DOS Compatible File Manager
LINEEDIT	Terminal Line Editor
NULL	Null device
CONSOLE	Console terminal using VRTX32 character I/O
PIPE	Pipe (named first-in first-out queue)
SOFTCLCK	Software time-of-day clock using VRTX32 timer interrupts
BYTERAM	Byte-oriented RAM disk (used without the disk buffer cache manager)
SECTORAM	Sector-oriented RAM disk (can be used with the Disk Buffer Cache Manager)

These standard device drivers and managers are automatically installed by the *ifx_init* call. If you only use these drivers and managers, then you can skip the remainder of this section.

If you have written your own device driver or manager, you must install it before you can install the device that uses this driver. This is done with the *ifx_driver* call. You normally install device drivers during system initialization. However, you can also install device drivers dynamically, during application run-time. This allows you to have device driver code that is loaded into memory only when it is needed.

To complement the ability to dynamically install a device driver, IFX also provides the function *ifx_rmdriver* to take the device driver out of the system.

This section deals only with the installation and removal of device drivers in your system, and assumes that the device driver is already written.

7.2.1 *ifx_driver*

The *ifx_driver* call installs either a device driver or a manager. Call *ifx_driver* as follows:

```
status = ifx_driver(name, driver, parameters... );
```

The *ifx_driver* has two required parameters:

- The name parameter is the name your program uses to refer to the device driver.
- The driver parameter is the address of the device driver.

The *ifx_driver* call accepts additional parameters, which *ifx_driver* ignores and passes on to the device driver.

IFX uses a far call when it calls a device driver. When the driver's address is passed as a parameter, the compiler pushes its offset only. The descriptor of the driver should be passed separately, immediately after.

7.2.2 *ifx_rmdriver*

The *ifx_rmdriver* call removes either a device driver or a manager. Call *ifx_rmdriver* as follows:

```
status = ifx_rmdriver(name);
```

The *ifx_rmdriver* has one required parameter:

- The name parameter is the name your program uses to refer to the device driver.

7.3 Installing, Mounting and Removing Devices

IFX needs to know all about the devices in the system. It finds out about devices through the *ifx_install* and *ifx_mount* calls, which you should call during system initialization. However, you can also install devices dynamically during application run-time. This allows you to have devices that load into memory only when they are needed.

To complement the ability to install a device, IFX also provides the function *ifx_remove* to take the device out of the system.

This section deals only with the installation and removal of devices in your system, and assumes that the device driver or manager is already written.

7.3.1 *ifx_install*

The *ifx_install* call installs a simple device. (To install a mounted device, use the *ifx_mount* call described below.) Call *ifx_install* as follows:

```
status = ifx_install(name, driver, parameters... );
```

The *ifx_install* call has two required parameters:

- The name parameter is the name your program uses to refer to the device.
- driver is the parameter name of the device driver.

The *ifx_install* call accepts additional parameters, which *ifx_install* ignores and passes on to the device driver.

7.3.2 *ifx_mount*

This section explains how to mount the Disk Buffer Cache Manager and Line Device Manager. Note that IFX's Circular Buffer Device Manager is installed using *ifx_install* rather than *ifx_mount*. Consult the *IFX User's Guide* for information on installing the MS-DOS file manager.

The *ifx_mount* call mounts an IFX device manager on top of an installed device. Use *ifx_mount* with the following device managers:

DISKBUF	Disk Buffer Cache Manager
VOLUME	MS-DOS-compatible file manager, which is usually mounted on top of DISKBUF. Consult the *IFX User's Guide* for details on mounting parameters and options.
LINEEDIT	Terminal Line Editor Manager (mount on top of CIRCULAR)

Call *ifx_mount* as follows:

```
status = ifx_mount(name, devname, manager, parameters...);
```

The *ifx_mount* call has three required parameters:

- The name parameter is the name your program uses to refer to the mounted virtual device (for the MS-DOS File Manager, this device is called a *volume*).
- The devname parameter is the name you have assigned to a device in a previous *ifx_install* or *ifx_mount* call.
- The manager parameter is the name of an IFX standard device manager.

The *ifx_mount* call accepts additional parameters, which *ifx_mount* ignores and passes on to the installed device or manager. Note that you can mount a device manager on top of another device manager, as well as on top of a device. For example, the MS-DOS-compatible file manager is usually mounted on top of the Disk Buffer Cache Manager, which is mounted on top of the disk device driver.

7.3.3 *ifx_remove*

The *ifx_remove* call removes a simple device or mounted device. It calls the device driver to do any necessary cleanup, then removes the device name from the IFX tables. Call *ifx_remove* as follows:

```
status = ifx_remove(name);
```

Parameters

The *ifx_remove* call has one required parameter: the device name, which is described in Section 7.3.1.

7.4 Using the Disk Buffer Cache Manager

The Disk Buffer Cache Manager is designed to be used in conjunction with a disk device driver to increase disk performance and simplify the implementation of the driver. The Disk Buffer Cache Manager:

- Reduces the number of disk I/O operations by using a modified least-recently used (LRU) algorithm
- Converts the application's byte-oriented I/O requests into sector-oriented operations that the disk driver can understand
- Improves performance by writing data to disk in an order that reduces disk head motion

- Improves reliability by using advanced algorithms to determine when to write critical information to disk

The existence of the Disk Buffer Cache Manager means that any given application call may or may not cause an actual disk access. In some cases, an application *read* call can cause a disk *write* (if a buffer needs to be flushed so that data can be read into it). An application *write* call can cause a disk *read* (if the write does not completely cover a sector). Also, disk accesses do not necessarily occur strictly in LRU order. The Disk Buffer Cache Manager writes buffers in a way that improves reliability.

For example, the Disk Buffer Cache Manager writes all data first, followed by directory and File Allocation Table (FAT) updates. This means that in the worst case, data might be lost, but the disk structure is not corrupted. For the same reason, related directory and FAT information is written with consecutive write operations to reduce the chance that, for example, a file's allocation is changed without the corresponding FAT entry being changed to reflect the new allocation.

To improve performance, the disk buffer cache is bypassed for transfers of large quantities of data. The cutoff point is twice the buffer size. Data transfers for less than this amount go through the buffer cache. Data transfers for more than this amount go directly to the disk. There are a few exceptions to this rule. In particular, if the disk contains bad sectors in the region of the transfer, or if some of the data in the region of the transfer is dirty (needs to be written to disk), then the cutoff point is different.

7.4.1 Mounting the Disk Buffer Cache Manager

To mount the Disk Buffer Cache Manager on top of an installed disk driver, use the *ifx_mount* call. It is mounted on top of the disk device driver, which must already be installed. Consult the *IFX User's Guide* for information on mounting the MS-DOS-compatible file manager on top of the Disk Buffer Cache Manager.

Use this template to mount the Disk Buffer Cache Manager:

```
#define name "cache:"
#define devname "disk:"
#define manager "DISKBUF"
#define num_buffers 8
#define buffer_size 1

status = ifx_mount(name, devname, manager, num_buffers, buffer_size);
```

In this example, cache: is the name you assign to the mounted cache manager, disk: is the name of the previously installed disk device driver, and DISKBUF is the name of IFX's Disk Buffer Cache Manager.

The num_buffers parameter is the number of buffers in the cache (the default is one buffer if this parameter is zero). The buffer_size parameter is the size of each buffer in sectors (the default is one sector if this parameter is zero). These two parameters have an important role in the reliability and performance of the buffer cache, so they are discussed in some detail here.

7.4.2 Determining the Number of Buffers

In general, the more buffers, the better the performance and the better the reliability. The optimal number of buffers is roughly dependent on:

- The number of open files
- The number of tasks accessing the disk
- The number of directory and file allocation operations in your application

. A good rule-of-thumb to start with is this:

```
number_of_buffers = 2 * number of open files and directories + 3
```

Note that for real-time, high-performance applications, it is best to keep the directory and file allocation operations at a minimum. For example, it is best to pre-allocate the space for a file when it is first created. Subsequent write operations can then fill the file without having to constantly increase its allocation. This both ensures that the file's data is contiguous for fast access, and that time is not required for repeated allocation operations.

As a worst case, imagine two tasks, each extending a different file that had not been pre-allocated. This would result in space being allocated to the files in an interleaved fashion, thus fragmenting the disk. The files would take longer to extend, due to repeated allocation operations and FAT modifications. In addition, later sequential access to the files would take longer, because of their fragmented nature.

You should experiment to arrive at the best number of buffers. One way you can test whether your number was correct is to retrieve the disk buffer information by issuing *ifx_devctl* with the IFXOGDISKBUF control operation code. This returns disk buffer information in the IFXDISKBUF structure.

```
IFXDISKBUF diskbuf;

status = ifx_devctl("volume:", IFXOGDISKBUF, &diskbuf);
```

The IFXDISKBUF lock_failures field indicates the number of times the disk cache's internal buffer lock mechanism failed, which indicates that you should either:

- Use more buffers
- Mount the volume with a shorter synchronization interval, so the buffer cache is flushed to disk more often
- Use programming techniques such as file pre-allocation to reduce the number of directory and file allocation operations

Ideally, lock_failures should be zero. If it is not zero, it means your performance and reliability are probably not as high as they could be. A lock failure means that IFX was not able to write buffers containing related critical information to the disk consecutively. A lock failure does not mean that any disk corruption has occurred or that any data has been lost. However, it does indicate that there is a greater potential for disk corruption or data loss should the disk media be prematurely removed or a system crash occur.

7.4.3 Determining Size of Buffers

This parameter should take one of these values:

- A value of one sector is appropriate if your application's data access is totally random or if your memory is extremely limited. Neither of these conditions apply for most applications.
- A value equal to one track is usually best because the disk hardware can very quickly access an entire track. If memory is limited or data access is somewhat random, you can choose a number that divides evenly into the track size (for example, three sectors for a diskette with nine-sector tracks).
- If you have a multiple-head disk drive and are transferring fairly large amounts of data, a value that is a multiple of the track size might be appropriate. This is because once the disk head is at a given cylinder, it can quickly access all the tracks in that cylinder.

The above guidelines provide a starting point for determining buffer size. The final value, however, depends on your disk device and application. You may need to experiment before you arrive at the correct value.

7.5 Using the Line Editor Device Manager

The Line Editor Device Manager is designed to be mounted on top of an IFX terminal device driver/manager. It adds features such as buffering, echoing, and line-editing. It also simplifies the implementation of the driver.

You can install the Line Editor Device Manager on top of either IFX's Circular Buffer Manager or IFX's console device driver.

7.5.1 Mounting the Line Editor Device Manager

Mount a line editor device using the *ifx_mount* call:

```
#define name "console:"
#define physicalname "consraw:"
#define manager "LINEEDIT"
#define line_length 80

status = ifx_mount(name, physicalname, manager, line_length);
```

The name parameter is the name your program uses to refer to the Line Editor Device Manager. The rawname parameter is the name you used to install IFX's Circular Buffer Manager or console device.

The manager parameter is the name of IFX's standard Line Editor Device Manager (the string LINEEDIT). The line_length parameter is an integer value that gives the maximum line length in bytes.

7.6 Using the Circular Buffer Device Manager

The Circular Buffer Device Manager handles the circular buffer used by a terminal device driver. Its installation creates a physical device that can also serve as the underlying device for the line editor device manager.

The Circular Buffer Device Manager works differently than most device drivers and managers. Instead of being installed on top of a device driver, it is installed together with a device driver. The device driver does not need to be previously installed, for the installation uses the driver address rather than a logical name assigned at installation.

NOTE

The Circular Buffer Device Manager cannot be installed with IFX's console device driver. The console driver is a special device that provides its own circular buffering. Both can coexist in the same system, but they cannot be installed together as one unit.

The Circular Buffer Device Manager cannot be shared with RTscope. If you want to share a serial port between IFX and RTscope, and you also have several other serial ports, we recommend that you make the first device a console device and the others circular buffer devices. Then simply install RTscope in the *one channel* configuration. Consult your RTscope user's guide for more information.

7.6.1 Installing the Circular Buffer Device Manager

Install the Circular Buffer Device Manager with the *ifx_install* call (*not* the *ifx_mount* call). Note that *ifx_install* is used differently in this case than it is for other kinds of device driver installation.

```
#define name "serial:"
#define manager "CIRCULAR"
#define drive "MYDRIVER"
#define in_buf_size 64
#define out_buf_size 64

status = ifx_install(physicalname, manager,
    driver, in_buf_size, out_buf_size);
```

The physicalname parameter is the name your program uses to refer to the device. The circular_mgr parameter is the name of IFX's standard Circular Buffer Device Manager (the string *CIRCULAR*). The driver parameter is the name of the terminal device driver (the string *MYDRIVER*).

The in_buf_size parameter is an integer value that gives the size (in bytes) of the type ahead buffer used for input from the device. The out_buf_size parameter is an integer value that gives the size (in bytes) of the buffer used for output to the device. Both the input and output buffer sizes must be a power of 2. The buffers can hold one fewer character than the size specified. The default size is 64 when you specify a zero parameter.

Once you have successfully called *ifx_init*, you can issue other IFX calls. You must install an IFX device before you can make any I/O calls on that device. Because interrupts are disabled before the *vrtx_go* call and enabled after the *vrtx_go* call, you must install some devices either before or after *vrtx_go:*

- Install devices that can generate unsolicited interrupts *before* calling *vrtx_go*. This ensures that the device-handling interface is properly set up before *vrtx_go* enables interrupts.

- Install (or mount) device drivers and managers that might pend during installation *after vrtx_go*. In particular, you must mount MS-DOS volumes after *vrtx_go*. A pend that occurs while interrupts are disabled can result in a system hang.
- Install devices that meet neither of the above conditions either before or after *vrtx_go*.

7.6.2 Step-by-Step Summary

A typical program that uses IFX does the following:

1. Initializes VRTX32 with the *vrtx_init* call
2. Initializes IFX with the *ifx_init* call
3. Installs devices that generate interrupts that cannot be disabled with the *ifx_install* call
4. Sets multitasking in action with VRTX32's *vrtx_go* call
5. Installs remaining devices with the *ifx_install* call, and mounts device managers with the *ifx_mount* call
6. Performs I/O using the installed devices and mounted managers

If there are no devices to be installed at step 3, it is permissible to call *ifx_init* between steps 4 and 5, rather than at step 2.

Refer to Chapter 1 for information on IFX devices and virtual devices. Consult your VRTX32 user's guide for information on the *vrtx_init* and *vrtx_go* calls.

7.7 Standard Device Drivers

IFX comes packaged with several standard device drivers that you can install using *ifx_install*. Ready Systems also provides sample device drivers on your shipping media to use as templates in developing your own device drivers.

Each standard device driver is reentrant and position-independent, and uses no read/write memory other than IFX workspace. Standard device drivers are also available in source code format.

Below is a list of the standard device drivers currently available. The list gives the standard device name, followed by the device description.

NULL	Null device
CONSOLE	Console terminal using VRTX32 character I/O
PIPE	Pipe (named first-in-first-out queue)
SOFTCLCK	Software time-of-day clock using VRTX32 timer interrupts
BYTERAM	Byte-oriented RAM disk (used without the disk buffer cache manager)
SECTORAM	Sector-oriented RAM disk (can be used with the disk buffer cache manager)

Many of these device drivers can be installed several times using different device names and installation parameters. The following sections describe each of these devices and how to install them.

IFX/86 is delivered in several files. The main file includes the IFX dispatcher and managers. Each of the drivers above is delivered separately. In order to provide position independence, their actual address should be passed to IFX in RUN-TIME. Therefore, for all drivers you should give the logical name by using the *ifx_driver* call.

7.7.1 *NULL*

Description

The null device is useful as a sink or empty source for data during testing. Writes to the null device are ignored, and reads return an immediate IFXEEOF error code.

Installation

Install the null device this way:

```
#define name "null:"
#define driver "NULL"

status = ifx_install(name, driver);
```

The name parameter is the name your program uses to refer to the device. The null_driver parameter is the name of IFX's standard null device driver (the string *NULL*).

The null device can be installed more than once under different names.

7.7.2 CONSOLE

Description

The console terminal device translates *ifx_read* and *ifx_write* calls into VRTX32 *sc_putc* and *sc_getc* calls. You must provide the VRTX32 interrupt handlers that issue the *ui_rxchr* and *ui_txrdy* calls. This device is handy for bringing up IFX on a system that previously used VRTX32 alone.

This device does *not* implement any line-editing functions. For that, you must mount IFX's line editor manager on top of this device (refer to Section 7.5).

The console terminal device uses a server task that handles XON/XOFF protocol. The server task has task ID zero and the same priority as the task that installs the console device with *ifx_install.*

Installation

Install the console device as follows:

```
#define name "consraw:"
#define driver "CONSOLE"

status = ifx_install(name, driver);
```

The name parameter is the name your program uses to refer to the device. The console_driver parameter is the name of IFX's standard console device driver (the string *CONSOLE*).

You should install the console terminal device only under one name.

The console terminal device can be shared with RTscope, if desired, since it actually uses VRTX32 to do the work. To share the console device with RTscope, simply install RTscope in the *one channel* configuration. Consult your RTscope user's guide for more information.

7.7.3 PIPE

Description

This device is a first-in-first-out queue, similar to the UNIX pipe except that it has a name. One task can open the pipe for writing, and another task for reading. If more than one task opens it for writing (or reading), then the tasks intersperse data. However, each read or write operation is atomic; that is, all the data from a given operation is transferred before the data from another operation is transferred.

Associated with each pipe is a buffer containing data that has been written by one task, but has not yet been read. The larger this buffer, the fewer the number of task switches that occur.

When you open a descriptor to a pipe device for writing, then write some data and close the descriptor, a special end-of-file mark is placed into the stream. The task reading from the pipe receives the error code IFXEEOF when it tries to read the last data from the buffer before the end-of-file mark. Further reads succeed and return more data.

Installation

This example installs a pipe device with a buffer of 1024 bytes and up to eight end-of-file marks. The buffer is allocated from IFX workspace.

```
#define name "pipe:"
#define driver "PIPE"
#define bufsize 1024
#define eofmarks 8

status = ifx_install(name, driver, bufsize, eofmarks);
```

The name parameter is the name your program uses to refer to the device. The driver parameter is the name of IFX's standard pipe device driver (the string *PIPE*).

The bufsize parameter is an integer value that gives the pipe buffer size in bytes. The default size is 512 when you give a zero parameter. The eofmarks parameter is an integer value that specifies the number of end-of-file marks allowed. A default value of 16 end-of-file marks is used when you give a zero parameter. Both the pipe and the end-of-file buffer size must be a power of 2.

The pipe device can be installed more than once, under different names, to create several pipes.

7.7.4 *SOFTCLCK*

Description

IFX files have a time stamp that records the date and time a file was created or last modified. Many systems have a hardware time-of-day clock that can be used to obtain the date and time for this purpose.

If your system does not have a hardware time-of-day clock, but does have a repeating interrupt that invokes the VRTX32 *ui_timer* call, then this device driver can maintain the date and time in software. All you have to do is tell it how many clock ticks there are per second.

Using the software time-of-day clock requires that you call *ifx_stime* during system startup, or else the date and time are initialized to 01/01/80 00:00:00.

The software time-of-day clock depends upon the VRTX32 system call *sc_gtime*, which returns a count of clock ticks since *vrtx_go*. Every time an application or the volume manager calls *ifx_gtime*, the driver calls *sc_gtime*. It then subtracts the previous time from the current time. This shows how many clock ticks have passed since the last time *ifx_gtime* was called. This difference is converted to seconds, minutes, hours, etc., and it is added to the clock. There is no server task involved.

Installation

Install the software clock device as follows:

```
#define name "clock:"
#define driver "SOFTCLCK"
#define ticks 100

status = ifx_install(name, driver, ticks);
```

The name parameter is the name your program uses to refer to the device. The driver parameter is the name of IFX's standard software clock device driver (the string *SOFTCLCK*).

The ticks parameter is an integer value that specifies the number of *ui_timer* interrupts per second.

The software clock device should be installed only once. Once it is installed, it cannot be removed.

7.7.5 *BYTERAM - SECTORAM*

Description

IFX includes two RAM disk drivers:

- The *BYTERAM* driver performs byte-oriented I/O operations, where data transfers occur as byte strings of any length that do not have to start on a sector boundary. You must not use the disk buffer cache manager with this RAM disk.

- The *SECTORAM* driver performs sector-oriented I/O operations, where data transfer starts at sector boundaries and occurs in blocks that must be a multiple of the disk sector size. You must mount the disk buffer cache manager on top of this RAM disk (although you will not realize a performance improvement, since the RAM disk exists in memory, as do the disk buffers).

These device drivers emulate a disk drive with 1 cylinder, 1 track, and a specified number of sectors of fixed size. The data is kept in read/write memory rather than on a real disk. A RAM disk is useful in the initial stages of developing and debugging an application before the physical disk driver has been written.

Installation

This example installs the byte-oriented RAM disk within the IFX workspace. The RAM disk has 100 512-byte sectors.

```
#define sector_size 512
#define total_sectors 100
#define name "ram:"
#define driver "BYTERAM"
#define address (char *) 0

status = ifx_install(name, driver, sector_size,
        total_sectors, address);
```

The name parameter is the name your program uses to refer to the device. The driver parameter is the name of one of IFX's standard RAM disk device drivers. The sector_size parameter is an integer value that gives the size of each sector in bytes (512 is recommended). The total_sectors parameter is an integer value that gives the number of sectors desired.

The address parameter is a pointer value that specifies the starting address of the RAM disk. A zero address causes the driver to allocate the storage from IFX workspace. Because IFX workspace is initialized to zero, you cannot keep data on a RAM disk between installations if you select a zero address. Therefore, always format a RAM disk when you mount a volume on top of it (or on top of the disk buffer cache, if you are using a sector RAM disk).

Each RAM disk device can be installed more than once, under different names, to create several RAM disks.

7.8 MVME320 Disk

Description

The MVME320 disk device driver supports the Motorola MVME320, MVME320A, MVME320A–1, and MVME320B, and MVME320B–1 disk controllers.

Installation

The MVME320 disk device driver should be installed as follows:

```
#define name0 "disk0:"
#define namel "diskl:"
#define driver "MVME320"
#define unit0 0
#define unitl 1
#define eca (char *) 0
#define cylinders 0

extern int MVME320DeviceDriver();

status = ifx_driver(driver, MVME320DeviceDriver);
status = ifx_install(name0, driver, unit0, eca, cylinders);
status = ifx_install(namel, driver, unitl, eca, cylinders);
```

The name parameter is the name your program uses to refer to the device. The driver parameter is the name of the device driver (the string MVME320). The unit parameter consists of two bits. Bits 0 and 1 are the disk drive number and are determined by a jumper on the disk drive itself.

The eca parameter should either be zero, or the address of an ECA data structure. If the eca is zero, then the driver will assume an ECA for a 40-cylinder, 2-head, 9-sector, 5¼-inch floppy disk. Please see your MVME320 hardware manual for information about the ECA data structure.

The cylinders parameter should be the number of cylinders on the disk. If this parameter is zero, then a default value of 40 cylinders is used.

If the driver returns status code IFXENOMEMORY when it is installed, the *malloc* function can't allocate memory space for the ECA data structure. You should check your C run-time library malloc_table and make sure you have declared enough memory blocks.

Appendix A
Example Device Drivers

A.1 Introduction

This appendix contains code fragments that are used to illustrate IFX interaction with applications, device drivers, and Interrupt Service Routines (ISRs). Although these examples are accurate for the point they attempt to illustrate, they are not complete device drivers and should be regarded as such. These examples are not included on your shipping media.

A.2 Advanced Request Ordering

```
/* This device driver shows how to process I/O requests in any order */

#include <compiler.h>
#include <vrtxvisi.h>
#include <ifxvisi.h>

#define MAX_REQUESTS 10         /* max. waiting I/O requests */
#define TOO_MANY_REQUESTS 0xFF01

/* State of each I/O request in queue */

#define EMPTY  0                /* slot is available for use */
#define READY  1                /* I/O is ready to be started */
#define ACTIVE 2                /* I/O is currently in progress */
#define DONE   3                /* I/O is done */

/* Parameter list passed to driver for IFXFREADS or IFXFWRITES */

typedef struct {
    long starting_sector;       /* starting sector number */
    char *buffer;               /* application data buffer */
    long number_of_sectors;     /* desired transfer count */
    long actual_sectors;        /* actual transfer count */
} PLIST;

/* Information about one I/O request in queue */

typedef struct {
    int state;                  /* see I/O states defined above */
    int opcode;                 /* IFXFREADS or IFXFWRITES */
    PLIST *plist;               /* parameter list pointer */
    int status;                 /* I/O error code */
    char *mailbox;              /* posted to when done */
    int priority;               /* priority of requesting task */
    long count;                  /* unique count assigned to request */
} REQUEST;

/* Device driver entry point */

int device_driver(opcode, dcb_ptr, plist)
int opcode;
IFXDCB *dcb_ptr;
PLIST *plist;
{
    extern REQUEST *insert_request(), *pick_priority();
    static REQUEST requests[MAX_REQUESTS];
    static int sema;            /* used to ensure mutual exclusion */
    static int active=0;        /* whether an I/O request is active */
    static long count=0L;         /* number of requests made so far */
    REQUEST *r;                 /* current I/O request */
    int err, status, info[3]; /* error codes */
    REQUEST *(*pick)() = pick_priority;
    switch (opcode) {
    case IFXFINSTALL:
        sema = sc_screate(0, 0, &err);
        for (r = requests; r < &requests[MAX_REQUESTS]; ++r)
            r->state = EMPTY;
        /* do other initialization here */
        status = RET_OK;
        break;
    case IFXFREADS:
    case IFXFWRITES:
        sc_spend(sema, 0L, &err);
        if (active) {
            sc_spost(sema, &err);
            sc_pend(&r->mailbox, 0L, &err);
            status = r->status;
            r->state = EMPTY;
            break;
        } else {
          r=insert_request(opcode, plist)
            for (r = requests; r < &requests[MAX_REQUESTS]; ++r)
                if (r->state == EMPTY)
                    break;
            if (r >= &requests[MAX_REQUESTS]) {
                sc_spost(sema, &err);
                status = TOO_MANY_REQUESTS;
                break;
            }
            sc_tinquiry(info, 0, &err);
            r->count = ++count;
            r->priority = info[2];
            r->state = READY;
            r->opcode = opcode;
            r->plist = plist;
            r->mailbox = 0;
            active = 1;
            sc_spost(sema, &err);
            status = do_io(opcode, plist);
            for (;;) {
                sc_spend(sema, 0L, &err);
                r = (*pick)(requests, MAX_REQUESTS);
                r->state = ACTIVE;
                sc_spost(sema, &err);
                if (!r) {
                   active = 0;
                   break;
                }
                r->status = do_io(r->opcode, r->plist);
                r->state = DONE;
                sc_post(&r->mailbox, (char *) 1, &err);
            }
        }
        break;
    case IFXFREMOVE:
        sc_sdelete(sema, &err);
        status = RET_OK;
        break;
    }
    return status;
}

/* This function does the real I/O and returns a status code */

int do_io(opcode, plist)
int opcode;
PLIST *plist;
{
    return RET_OK;
}

/* Pick request which has the highest task priority */

REQUEST *pick_priority(requests, max_request)
REQUEST *requests;
int max_request;
{
    REQUEST *r, *best_request = 0;
    int highest_priority = -1, highest_count = 0;
    for (r = requests; r < &requests[max_request]; ++r)
        if (r->state == READY && r->priority >= highest_priority &&
                                r->count > highest_count) {
            highest_priority = r->priority;
            highest_count = r->count;
            best_request = r;
        }
    return best_request;
}

/* Pick request which has been waiting the longest time */

REQUEST *pick_fifo(requests, max_request)
REQUEST *requests;
int max_request;
{
    REQUEST *r, *best_request = 0;
    long highest_count = 0L;
    for (r = requests; r < &requests[max_request]; ++r)
        if (r->state == READY && r->count > highest_count) {
            highest_count = r->count;
            best_request = r;
        }
    return best_request;
}

/* Pick request which has the shortest seek time */

REQUEST *pick_shortest_seek(requests, max_request)
REQUEST *requests;
int max_request;
{
    static long current_sector = 0L;
    REQUEST *r, *best_request = 0;
    long distance, shortest_distance = 0x7FFFFFFFL;
    for (r = requests; r < &requests[max_request]; ++r)
        if (r->state == READY) {
            distance = labs(r->plist->starting_sector - current_sector);
            if (distance < shortest_distance) {
                shortest_distance = distance;
                best_request = r;
            }
        }
    if (best_request)
        current_sector = best_request->plist->starting_sector +
                              best_request->plist->actual_sectors;
    return best_request;
}

/* Pick request which has the smallest transfer count */

REQUEST *pick_smallest_transfer(requests, max_request)
REQUEST *requests;
int max_request;
{
    REQUEST *r, *best_request = 0;
    long smallest_transfer = -1L;
    for (r = requests; r < &requests[max_request]; ++r)
        if (r->state == READY && r->plist->actual_sectors < smallest_transfer)
{
            smallest_transfer = r->plist->actual_sectors;
            best_request = r;
        }
    return best_request;
}

/* Pick request according to the elevator algorithm */

typedef enum {UP, DOWN} DIRECTION;

REQUEST *pick_elevator(requests, max_request)
REQUEST *requests;
int max_request;
{
    static long current_sector = 0L;
    static DIRECTION current_direction = UP;
    REQUEST *r, *best_up = 0, *best_down = 0, *best_request = 0;
    long shortest_up = 0x7FFFFFFFL, largest_down = -1L;
    for (r = requests; r < &requests[max_request]; ++r)
        if (r->state == READY) {
            if (r->plist->starting_sector >= current_sector) {
                if (r->plist->starting_sector < shortest_up) {
                    shortest_up = r->plist->starting_sector;
                    best_up = r;
                }
            } else {
                if (r->plist->starting_sector > largest_down) {
                    largest_down = r->plist->starting_sector;
                    best_down = r;
                }
            }
        }
    best_request = 0;
    switch (current_direction) {
    case UP:
        if (best_up)
            best_request = best_up;
        else if (best_down) {
            current_direction = DOWN;
            best_request = best_down;
        }
        break;
    case DOWN:
        if (best_down)
            best_request = best_down;
        else if (best_up) {
            current_direction = UP;
            best_request = best_up;
        }
        break;
    }
    if (best_request)
        current_sector = best_request->plist->starting_sector;
    return best_request;
}
```

Appendix B
Sample 68000 Device Drivers

READY SYSTEMS

B.1 Introduction

This appendix contains source code for 68000 sample device drivers that are included on your IFX shipping media. You can use these sample device drivers as a template for writing a custom driver for your device. These drivers have been tested and are fully functional.

The following files, which are written in C, are included:

- *sectoram.c* and *ifxsram.h* together make up the sector-oriented RAM disk driver. They illustrate how a disk device driver should interact with IFX.
- *mm58274.c* is a device driver for the MM58274 clock device.
- *mvme133.c* includes the device driver for the MVME133 board serial I/O chip. This driver handles installation, *ifx_ioctl* operations, and device removal.
- *mvme133i.a68* is a 68000 assembly language file that contains the ISRs and device-initialization code. It illustrates ISR and device-initialization format, so you can use it as a template even if you are using a processor other than the 68000.
- *mvme320.c*, *mvme320i.a68*, and *ifxmv320.h* files make up a device driver for the MVME320 disk controller. They illustrate how to write a driver for a 5¼-inch floppy disk.
- *rf3500.c* and *rf3500i.a68* files together make up a device driver for the Ciprico Rim-Fire 3500 and 3510 SCSI disk controllers. They illustrate how to write a driver for a SCSI controller.

IFX also includes additional files for external routines, such as *LockSemaphore*, *UnlockSemaphore*, *AllocateMemory2*, and *FreeMemory2*.

B.2 sectoram.c

```
/* IFX device driver for sector-oriented RAM disk */

#include <compiler.h>
#include <vrtxvisi.h>
#include <ifxvisi.h>
#include <ifxsram.h>

/* Define some helpful macros */

#define Nil     0
#define CB      char

#ifdef I386
#define FAR _far
#define FIL1 short int filler1;
#else
#define FAR
#define FIL1
#endif

/* Parameter list */

typedef union {
    struct {
        unsigned int sector_size;
        long total_sectors;
        char *RAM_address;
    } u1;
    struct {
        long starting_sector;
        char *buffer;
        FIL1
        long number_of_sectors;
        long *actual_count;
    } u2;
    struct {
        int opcode;
        int cylinder;
        int track;
    } u3;
    struct {
        int opcode;
        IFXGEOMETRY *geometry;
    } u4;
} PL;

/* Externals */

extern CB *AllocateMemory2();
extern void bcopy(), FreeMemory2();
```

```
/*+ SectorRamDiskDriver

Description:

    This sector-oriented RAM disk driver is at the same logical level as a real
    disk driver, as opposed to the byte-oriented RAM disk driver, which is at
    the same level as the Disk buffer Cache. The difference is that this driver
    accepts IFXFREADS and IFXFWRITES operations while the byte-oriented RAM
    disk driver accepts IFXFREADP and IFXFWRITEP operations.

   Input :  opcode   - IFXFREADS
                       IFXFWRITES
                       IFXFINSTALL
                       IFXFREMOVE
            dcb      - Pointer to the IFXDCB structure
            pl       - Pointer to a parameter list containing:

           starting_sector      First sector where operation begins
           buffer               Pointer to a buffer (input buffer for
                                Read, output buffer for Write)
           number_of_sectors    Number of sectors to read or write
           actual_count         Actual number of sectors transferred

-*/

FAR int SectorRamDiskDriver(opcode, dcb, pl)
int opcode;
IFXDCB *dcb;
PL *pl;

{
    int err;
    SectorRamDiskDCB *r;
    long total_bytes;
    long num_bytes;
    long offset;
    IFXGEOMETRY *dc;

    /* Perform operation according to opcode */

    r = (SectorRamDiskDCB *) dcb->dt;
    switch (opcode) {
     /* Install device */

        case IFXFINSTALL:
            r = (SectorRamDiskDCB *) AllocateMemory2(
                            sizeof(SectorRamDiskDCB));
            if (r == Nil)
                return IFXENOMEMORY;
            r->sector_size = pl->u1.sector_size;
            r->total_sectors = pl->u1.total_sectors;
            r->in_workspace = pl->u1.RAM_address == Nil;
            if (r->in_workspace) {
                total_bytes = r->sector_size * r->total_sectors;
                r->RAM_address = (char *) AllocateMemory2(
                                (unsigned int) total_bytes);
                if (r->RAM_address == Nil) {
                    FreeMemory2((CB *) r, sizeof(SectorRamDiskDCB));
                    return IFXENOMEMORY;
                }
            } else
                r->RAM_address = pl->u1.RAM_address;
            dcb->device_type = IFXDDISK;
            dcb->dt = (CB *) r;
            err = RET_OK;
            break;

        /* Read sectors */

        case IFXFREADS:
            if (pl->u2.starting_sector < 0L ||
                pl->u2.starting_sector > r->total_sectors)
                err = IFXEBADPOSN;
            else if (pl->u2.starting_sector + pl->u2.number_of_sectors >
                                                        r->total_sectors)
                err = IFXEBADXFERCT;
            else {
                offset = r->sector_size * pl->u2.starting_sector;
                num_bytes = r->sector_size * pl->u2.number_of_sectors;
                bcopy(&r->RAM_address[offset], pl->u2.buffer,
                                    (unsigned int) num_bytes);
                *pl->u2.actual_count = pl->u2.number_of_sectors;
                err = RET_OK;
            }
            break;
```

```
/* Write sectors */

case IFXFWRITES:
    if (pl->u2.starting_sector < 0L ||
        pl->u2.starting_sector > r->total_sectors)
        err = IFXEBADPOSN;
    else if (pl->u2.starting_sector + pl->u2.number_of_sectors >
                                                  r->total_sectors)
        err = IFXEBADXFERCT;
    else {
        offset = r->sector_size * pl->u2.starting_sector;
        num_bytes = r->sector_size * pl->u2.number_of_sectors;
        bcopy(pl->u2.buffer, &r->RAM_address[offset],
                              (unsigned int) num_bytes);
        *pl->u2.actual_count = pl->u2.number_of_sectors;
        err = RET_OK;
    }
    break;

/* Remove device */

case IFXFREMOVE:
    if (r->in_workspace)
        FreeMemory2((CB *) r->RAM_address,
            (unsigned int) (r->sector_size * r->total_sectors));
    FreeMemory2((CB *) r, sizeof(SectorRamDiskDCB));
    err = RET_OK;
    break;

/* I/O control operation */

case IFXFIOCTL:
    switch (pl->u3.opcode) {

    /* Get disk geometry */

    case IFXOGGEOM:
        dc = pl->u4.geometry;
        dc->sector_size = r->sector_size;
        dc->sectors_per_track = r->total_sectors;
        dc->tracks_per_cylinder = 1;
        dc->total_cylinders = 1;
        dc->total_sectors = r->total_sectors;
        err = RET_OK;
        break;
        /* Format track */

        case IFXOFMTTRK:
            /* ignore pl->u3.cylinder and pl->u3.track */
            err = RET_OK;
            break;

        /* Unimplemented control operation */

        default:
            err = IFXENOTIMP;
            break;
        }
        break;

    /* Unimplemented function code */

    default:
        err = IFXENOTIMP;
        break;
    }

    /* Return status code to IFX */

    return err;
}
```

B.3 *ifxsram.h*

```
/* Sector-oriented RAM disk device control block */

typedef struct {
    char *RAM_address;          /* pointer to data */
    long total_sectors;         /* total number of sectors on disk */
    unsigned short sector_size; /* sector size in bytes */
    unsigned short in_workspace;/* whether data is located in workspace */
    long reserved;
} SectorRamDiskDCB;
```

B.4 *mm58274.c*

```
/* IFX device driver for MM58274 time-of-day clock */

#include <compiler.h>
#include <vrtxvisi.h>
#include <ifxvisi.h>

/* Define some helpful macros */

#define Nil     0
#define CB      char
#define B(x)    base[(x) << shift]  /* Access all or every other odd byte */

/* Parameter list */

typedef union {
    struct {
        IFXTIME *time;
    } u1;
    struct {
        unsigned char *base;
    } u2;
} PL;

/* MM58274 device control block */

typedef struct {
    IFXSEMA sema;               /* used to ensure mutual exclusion */
    unsigned char *base;        /* base address of M58274 device */
    unsigned char shift;        /* shift count needed to access each byte */
    char reserved[3];
} MM58274DCB;

/* Externals */

extern void LockSemaphore(), UnlockSemaphore(), FreeMemory2();
extern CB *AllocateMemory2();

/*+ MM58274DeviceDriver

Description:

    This is the clock device driver for the MM58274 time-of-day
    clock.  This driver was especially written for the MM58274
    on the MVME117 and MVME133 boards, but it should be easy to
    make it work in other systems.
```

```
Installation:

    To install this driver on a MVME117 system:

        status = ifx_install("CLOCK:", "MM58274", 0xF4C001L);

    To install this driver on a MVME133 system:

        status = ifx_install("CLOCK:", "MM58274", 0xFB0000L);

-*/

int MM58274DeviceDriver(opcode, dcb_ptr, pl)
int opcode;
IFXDCB *dcb_ptr;
PL *pl;

{
        MM58274DCB *xs;
        IFXTIME *dt;
        int dummy, err;
        unsigned char *base, shift;

        /* If opcode is IFXFINSTALL, then get base address and shift count */

        if (opcode == IFXFINSTALL) {
            xs = (MM58274DCB *) AllocateMemory2(sizeof(MM58274DCB));
            if (xs == Nil)
                return IFXENOMEMORY;
            xs->base = pl->u2.base;
            xs->shift = (unsigned char) pl->u2.base & 1;
            dcb_ptr->dt = (CB *) xs;
            dcb_ptr->device_type = IFXDCLOCK;
        }

        /* Get a quick copy of values in the device control block */

        xs = (MM58274DCB *) dcb_ptr->dt;
        base = xs->base;
        shift = xs->shift;

        /* Dispatch based on opcode */

        switch (opcode) {

        /* Install device */

        case IFXFINSTALL:
            B(0) = 3;
            B(15) = 0;
            B(0) = 1;
            err = RET_OK;
            break;
```

```
/* Remove device */

case IFXFREMOVE:
    FreeMemory2((CB *) xs, sizeof(MM58274DCB));
    err = RET_OK;
    break;

/* Get current time-of-day */

case IFXFGTIME:
    dt = pl->ul.time;
    LockSemaphore(&xs->sema);
    dummy = B(0);
    do {
        dt->second = (B(3) & 7)*10 + (B(2) & 15);
        dt->minute = (B(5) & 7)*10 + (B(4) & 15);
        dt->hour = (B(7) & 3)*10 + (B(6) & 15);
        dt->day = (B(9) & 3)*10 + (B(8) & 15);
        dt->month = (B(11) & 1)*10 + (B(10) & 15);
        dt->year = (B(13) & 15)*10 + (B(12) & 15);
        if (dt->year < 80)
            dt->year += 100;
        dummy = B(0);
    } while (dummy & 8);
    UnlockSemaphore(&xs->sema);
    err = RET_OK;
    break;

/* Get current time-of-day */

case IFXFSTIME:
    dt = pl->ul.time;
    LockSemaphore(&xs->sema);
    B(0) = 5;
    B(15) = 1;
    B(2) = dt->second % 10;
    B(3) = dt->second / 10;
    B(4) = dt->minute % 10;
    B(5) = dt->minute / 10;
    B(6) = dt->hour % 10;
    B(7) = dt->hour / 10;
    B(8) = dt->day % 10;
    B(9) = dt->day / 10;
    B(10) = dt->month % 10;
    B(11) = dt->month / 10;
    B(12) = dt->year % 10;
    B(13) = (dt->year / 10) % 10;
    B(14) = 1;
    B(15) = ((dt->year & 3) << 2) + 1;
```

```
            B(0) = 1;
            UnlockSemaphore(&xs->sema);
            err = RET_OK;
            break;

        /* Unimplemented function code */

        default:
            err = IFXENOTIMP;
            break;
        }

        return err;
}
```

B.5 *mvme133.c*

```
/* Serial device driver for AmZ8530 on the MVME133 board */

#include <compiler.h>
#include <vrtxvisi.h>
#include <ifxvisi.h>

/* Externals */

extern void AmZ8530InitializeA(), AmZ8530TransmitDriverA();
extern char **GetVBR();

#define TBCB_PTRA 0x54

/* Parameter list */

typedef union {
    struct {
        int p1;
        int p2;
    } u35;
} PL;

/* Device driver for channel A */

int AmZ8530DeviceDriverA(opcode, dcb_ptr, pl)
int opcode;
IFXDCB *dcb_ptr;
PL *pl;

{
        int err;

        switch (opcode) {
        case IFXFINSTALL:
            (GetVBR())[TBCB_PTRA] = dcb_ptr->dt;
            AmZ8530InitializeA();
            err = RET_OK;
            break;
        case IFXFIOCTL:
            if (pl->u35.p1 == IFXOTXRDY) {
                AmZ8530TransmitDriverA(pl->u35.p2);
                err = RET_OK;
                break;
            }
        default:
            err = IFXENOTIMP;
            break;
        }
        return err;
}
```

B.6 *mvme133l.a68*

```
* Serial device driver for AmZ8530 on MVME133 board

    INCLUDE 'vrtxvisi.inc'
    INCLUDE 'ifxvisi.inc'

    XDEF    AmZ8530InitializeA
    XDEF    AmZ8530TransmitDriverA
    XDEF    AmZ8530InterruptServiceRoutine
    XREF    AmZ8530DeviceDriverA

SIO          EQU   $FA0000   * Base address of Z8530
SIOA_CNTRL   EQU   SIO       * Address of channel A control byte
SIOA_DATA    EQU   SIO+1     * Address of channel A data byte
SIOB_CNTRL   EQU   SIO+2     * Address of channel B control byte
SIOB_DATA    EQU   SIO+3     * Address of channel B data byte
SIO_EXC      EQU   $50       * Exception vector for SIO Z8530
SIO_VEC      EQU   $140
TBCB_PTRA    EQU   $54*4

    SECTION 0

AmZ8530:
    BRA     AmZ8530DeviceDriverA

* Zilog 8530 Serial I/O Controller Initialization
* Motorola MVME 133

* Initialize channel A

AmZ8530InitializeA:

    DC.W    $4E7A,$8801          * MOVEC.L VBR,A0
    LEA     AmZ8530InterruptServiceRoutine(PC),A1
    MOVE.L  A1,SIO_VEC(A0)       * Set up interrupt vectors

    MOVE.B  #$00,SIOB_CNTRL      * Reset pointer to WR0

    MOVE.B  #$09,SIOB_CNTRL      * Master Interrupt Control Register
    MOVE.B  #$C8,SIOB_CNTRL      * Force hardware reset; Select vis
    NOP                          *-Delay #4 to process reset
    NOP                          * Delay #3 to process reset
    NOP                          * Delay #2 to process reset
    NOP                          * Delay #1 to process reset
    NOP                          * Delay #0 to process reset

    MOVE.B  #$02,SIOB_CNTRL      * Interrupt Vector Register
    MOVE.B  #SIO_EXC,SIOB_CNTRL   * SIO Interrupt Vector Number/Digit

    MOVE.B  #$00,SIOA_CNTRL      * Reset pointer to WR0
    MOVE.B  #$09,SIOA_CNTRL     * Master Interrupt Control Register
    MOVE.B  #$C8,SIOA_CNTRL     * Force hardware reset; Select vis
    NOP                         *~Delay #4 to process reset
    NOP                         * Delay #3 to process reset
    NOP                         * Delay #2 to process reset
    NOP                         * Delay #1 to process reset
    NOP                         * Delay #0 to process reset

    MOVE.B  #$02,SIOA_CNTRL     * Interrupt Vector Register
    MOVE.B  #SIO_EXC,SIOA_CNTRL  * SIO Interrupt Vector Number/Digit

    MOVE.B  #$0B,SIOA_CNTRL     * Clock Mode Control Register
    MOVE.B  #$50,SIOA_CNTRL     * Use Baud Rate for Rx/Tx clocks
    MOVE.B  #$0C,SIOA_CNTRL     * Lower byte Baud Rate Time Constant
    MOVE.B  #$02,SIOA_CNTRL     * LSB for baud 9600
    MOVE.B  #$0D,SIOA_CNTRL     * Upper byte Baud Rate Time Constant
    MOVE.B  #$00,SIOA_CNTRL     * MSB for baud 9600
    MOVE.B  #$0E,SIOA_CNTRL     * Digital Phase-Locked Loop Command
    MOVE.B  #$01,SIOA_CNTRL     * Enable baud rate generator
    MOVE.B  #$04,SIOA_CNTRL     * Tx/Rx Misc. Parameters and modes
    MOVE.B  #$44,SIOA_CNTRL     * Clock x16; 1 stop bit; no parity
    MOVE.B  #$03,SIOA_CNTRL     * Receiver parameters and control
    MOVE.B  #$C1,SIOA_CNTRL     * 8 bits per character; Rx enable
    MOVE.B  #$0F,SIOA_CNTRL     * External/Status Interrupt Control
    MOVE.B  #$00,SIOA_CNTRL     * Disable all External/Status ints
    MOVE.B  #$05,SIOA_CNTRL     * Transmitter parameters and control
    MOVE.B  #$EA,SIOA_CNTRL     * 8 bit/char; Tx Enable; DTR, RTS
    MOVE.B  #$01,SIOA_CNTRL     * Tx/Rx Interrupt and Data Transfer
    MOVE.B  #$12,SIOA_CNTRL     * Poll or Int mode; enable Rx, Tx

    RTS

* Zilog 8530 Interrupt Service Routine for MVME133

* First the interrupt must be identified as a Rx or a Tx.
* If neither is found, the spurious interrupt is ignored.
* The Transmitter buffer will be checked follwing a Rx interrupt.

AmZ8530InterruptServiceRoutine:

    MOVE.L   D0,-(SP)           * Preserve D0; Restored by UI_EXIT
    MOVEM.L  D1-D2/A0-A1,-(SP)  * Preserve D1 and D2
    MOVE.B   #3,SIOB_CNTRL      * Reset Port Pointer to status reg
    MOVE.B   SIOB_CNTRL,D2      * Load the SIO status register into D2

* Check for a Rx character and then for an empty Tx Buffer

    BTST     #2,D2              * Check Rx character available bit
    BNE.S    SIO_RX             * If set, Rx Character
    BTST     #1,D2              * Check Tx Buffer empty bit
    BNE.S    SIO_TX             * If set, Tx Character
```

```
* Assume spurious interrupt and hope for the best

    MOVEM.L (SP)+,A0-A1/D1-D2  * Restore registers
    MOVE.L  (SP)+,D0
    RTE                        * Return from interrupt

SIO_TX:
    DC.W    $4E7A,$8801        * MOVEC.L VBR,A0
    MOVEA.L TBCB_PTRA(A0),A0   * A0 should contain the TBCB address
    MOVEA.L IFXSCBtransmit_ready(A0),A1
    JSR     (A1)
    TST.L   D0                 *Check if IOS discovered a character
    BEQ.S   SIO_TXCHAR         * Transmit character if present
    MOVE.B  #$28,SIOA_CNTRL    * Clear Tx Interrupt for channel
    BRA.S   SIO_RTN
SIO_TXCHAR:
    MOVE.B  D1,SIOA_DATA       * If char is present, output it
    BRA.S   SIO_RTN            * Return from SIO Routine

SIO_RX:
    MOVEQ.L #$7F,D1            * Mask out msb to get Ascii range
    AND.B   SIOA_DATA,D1       * Read the character from port
    DC.W    $4E7A,$8801        * MOVEC.L VBR,A0
    MOVEA.L TBCB_PTRA(A0),A0   * A0 should contain the TBCB address
    MOVEA.L IFXSCBreceive_character(A0),A1
    JSR     (A1)

SIO_RTN
    MOVE.B  #$38,SIOB_CNTRL  * Signal End of Interrupt for channel
    MOVEM.L (SP)+,D1/D2/A0/A1  * Restore registers
    MOVE.L  #UIFEXIT,D0        * Load the UI_EXIT function code
    TRAP    #0                 * Announce end of interrupt

* Transmitter driver routine

AmZ8530TransmitDriverA:
    MOVE.B  7(SP),SIOA_DATA    * output character to port
    RTS

    END
```

B.7 mvme320.c

```
/* $Header: mvme320.c,v 1.4 89/04/19 16:49:05 glenn Exp $ */

/* IFX device driver for MVME320 disk controller */

#include <compiler.h>
#include <vrtxvisi.h>
#include <ifxvisi.h>
#include <ifxmv320.h>

/* Define some helpful macros */

#define Nil     0
#define CB      char

/* Parameter list */

typedef union {
    struct {
        int drive;
        ECA *eca;
        int totalCylinders;
    } u1;
    struct {
        long starting_sector;
        char *buffer;
        long number_of_sectors;
        long *actual_count;
    } u2;
    struct {
        int opcode;
        int cylinder;
        int track;
    } u3;
    struct {
        int opcode;
        IFXGEOMETRY *geometry;
    } u4;
} PL;

/* Externals */

extern void MVME320_InterruptServiceRoutine(), FreeMemory2(),
            LockSemaphore(), UnlockSemaphore();
extern unsigned long MVME320_Offset;
extern unsigned char *MVME320_CSR;
extern unsigned short MVME320_InterruptVector;
extern CB **GetVBR(), *AllocateMemory2();

/* Default ECA for floppy disks */
```

```
ECA MVME320_DefaultECA = {
    0,           /* command_code */
    0,           /* main_status */
    0,           /* extended_status */
    10,          /* maximum_retries */
    0,           /* actual_retries */
    0,           /* dma_type */
    0,           /* command_option */
    0,           /* buffer_address */
    0,           /* buffer_length */
    0,           /* actual_count */
    0,           /* cylinder */
    0,           /* surface */
    0,           /* sector */
    0,           /* current_position */
    {0,0,0,0,0},/* reserved1 */
    0x50,        /* pre_index_gap */
    0x32,        /* post_index_gap */
    0x0C,        /* sync_byte_count */
    0x16,        /* post_id_gap */
    0x36,        /* post_data_gap */
    0x03,        /* address_mark_count */
    0x02,        /* sector_length_code */
    0xE5,        /* fill_byte */
    {0,0,0},     /* reserved2 */
    0x05,        /* drive_type */
    0x02,        /* number_of_surfaces */
    0x09,        /* sectors_per_track */
    0x18,        /* stepping_rate */
    0x46,        /* head_settling_time */
    0x46,        /* head_load_time */
    0,           /* seek_type */
    0,           /* phase_count */
    0x28,        /* low_write_current_track */
    0x28,        /* precompensation_track */
    {0,0,0},     /* ecc_remainder */
    {0,0,0},     /* append_ecc_remainder */
    0,           /* reserved3 */
    {0,0,0},     /* working_area */
    0,           /* reserved4 */
    0,           /* mailbox */
    0            /* reserved5 */
};

/* Initialize the ECA table and controller for all drives (call once only) */


int MVME320_Initialize()
```

```
{
        ECATable *ecat;
        unsigned long eca_ul;
        unsigned char *csr;
        int err;

        /* Allocate memory for ECA table and clear it out */

        ecat = (ECATable *) AllocateMemory2(sizeof(ECATable));
        if (ecat == Nil)
            return IFXENOMEMORY;

        /* Put the ECA table pointer into controller */

        eca_ul = ((unsigned long) ecat + MVME320_Offset) >> 1;
        csr = MVME320_CSR;
        csr[1] = eca_ul;
        csr[3] = eca_ul >> 8;
        csr[5] = eca_ul >> 16;
        csr[7] = eca_ul >> 24;

        /* Set up interrupt vector register */

        csr[9] = MVME320_InterruptVector;

        /* Make entry in exception vector table for interrupt handler */

        (GetVBR())[MVME320_InterruptVector] =
            (CB *) MVME320_InterruptServiceRoutine;

        return RET_OK;
}

/* Compute address of ECA table and return pointer to it */

ECATable *MVME320_GetECATable()

{
        unsigned long eca_ul;
        unsigned char *csr;

        csr = MVME320_CSR;
        eca_ul = (unsigned long) csr[1] << 1;
        eca_ul |= (unsigned long) csr[3] << 9;
        eca_ul |= (unsigned long) csr[5] << 17;
        eca_ul |= (unsigned long) csr[7] << 25;
        eca_ul -= MVME320_Offset;

        return (ECATable *) eca_ul;
}
```

```
int MVME320_InstallDevice(ddcb, eca, initial_eca, totalCylinders)
MVME320DiskDCB *ddcb;
ECA *eca, *initial_eca;
int totalCylinders;

{
        int drive;

        drive = ddcb->drive;
        if (initial_eca == Nil)
            initial_eca = &MVME320_DefaultECA;
        if (totalCylinders == 0)
            totalCylinders = 40;

        /* Set up ECA */

        *eca = *initial_eca;

        /* Set up MVME320DiskDCB */

        ddcb->sectorSize = 128 << eca->sector_length_code;
        ddcb->sectorsPerTrack = eca->sectors_per_track;
        ddcb->tracksPerCylinder = eca->number_of_surfaces;
        ddcb->totalCylinders = totalCylinders;
        ddcb->totalSectors = ddcb->totalCylinders * ddcb->tracksPerCylinder *
                                                         ddcb->sectorsPerTrack;

        /* all done */

        return RET_OK;
}

int MVME320_Operation(ddcb, eca)
MVME320DiskDCB *ddcb;
ECA *eca;

{
        int err, mask;
        unsigned char *csr;

        mask = 0x10 << ddcb->drive;
        eca->main_status = 0xFF;
        eca->extended_status = 0;
        eca->phase_count = 0;
        eca->mailbox = Nil;
        csr = MVME320_CSR;
        csr[13] |= mask;
        (void) sc_pend(&eca->mailbox, 500L, &err);
        if (err != 0) {
            csr[13] &= ~mask;
            (void) sc_pend(&eca->mailbox, 0L, &err);
        }
        if (eca->main_status == 0)
            err = RET_OK;
        else if (eca->main_status == 2)
            err = IFXEDVNTREADY;
        else if (eca->extended_status & 0x0040)
            err = IFXERDONLYM;
        else
            err = IFXEIOERR;
        return err;
}

int MVME320_FormatTrack(ddcb, eca, cylinder, track)
MVME320DiskDCB *ddcb;
ECA *eca;
int cylinder, track;

{
        unsigned char format_table[5*16];
        int sector, drive_type;
        unsigned char *p;

        drive_type = eca->drive_type & 0x7F;
        eca->command_code = 7;
        eca->buffer_address = (unsigned long) format_table + MVME320_Offset;
        eca->cylinder = cylinder;
        eca->surface = track;
        for (sector = 1, p = format_table; sector <= ddcb->sectorsPerTrack;
                                                               ++sector) {
            /* drive types 2 and 3 are for hard disk */
            if (drive_type == 2 || drive_type == 3)
                *p++ = cylinder >> 8;
            *p++ = cylinder;
            *p++ = track;
            *p++ = sector;
            *p++ = eca->sector_length_code;
        }
        return MVME320_Operation(ddcb, eca);
}

int MVME320_ReadWriteSectors(ddcb, eca, functionCode, sectorPosition,
                            bufferAddress, numberSectors, actualCount)
MVME320DiskDCB *ddcb;
ECA *eca;
int functionCode;
long sectorPosition;
char *bufferAddress;
```

```
long numberSectors;
long *actualCount;

{
    int err;
    long thisTime, maxSectors;

    if (sectorPosition < 0L || sectorPosition > ddcb->totalSectors)
        return IFXEBADPOSN;
    if (sectorPosition + numberSectors > ddcb->totalSectors)
        return IFXEBADXFERCT;
    maxSectors = (65536L / ddcb->sectorSize) - 1L;
    for (*actualCount = 0L; *actualCount < numberSectors;
                            *actualCount += thisTime) {
        eca->command_code = functionCode == IFXFWRITES ? 6 : 5;
        eca->buffer_address = (unsigned long) bufferAddress + MVME320_Offset;
        eca->cylinder = sectorPosition / (ddcb->sectorsPerTrack *
                                          ddcb->tracksPerCylinder);
        eca->surface = (sectorPosition / ddcb->sectorsPerTrack) %
                                          ddcb->tracksPerCylinder;
        eca->sector = (sectorPosition % ddcb->sectorsPerTrack) + 1;
        thisTime = numberSectors - *actualCount;
        if (thisTime > maxSectors)
            thisTime = maxSectors;
        eca->buffer_length = thisTime * ddcb->sectorSize;
        err = MVME320_Operation(ddcb, eca);
        if (err) break;
    }
    return err;
}

int MVME320_DeviceDriver(functionCode, dcb, pl)
int functionCode;
IFXDCB *dcb;
PL *pl;

{
        MVME320DiskDCB *ddcb;
        ECATable *ecat;
        ECA *eca;
        int drive, err, err2;
        IFXGEOMETRY *dc;

        /* Handle IFXFDRIVER specially by initializing ECA table */

        if (functionCode == IFXFDRIVER)
            return MVME320_Initialize();

        /* Get pointer to ECA table */
```

```
ecat = MVME320_GetECATable();

/* Handle IFXFRMDRIVER specially by releasing ECA table */

if (functionCode == IFXFRMDRIVER) {
    FreeMemory2((CB *) ecat, sizeof(ECATable));
    return RET_OK;
}

/* Don't let any other task use the disk controller */

LockSemaphore(&ecat->mutex);

if (functionCode == IFXFINSTALL) {
    drive = pl->ul.drive & 3;
    if (drive < 0 || drive > 3)
        err = IFXEBADDRIVE;
    else if (ecat->eca_table[drive] != Nil)
        err = IFXEDVEXISTS;
    else {
        ddcb = (MVME320DiskDCB *)
                    AllocateMemory2(sizeof(MVME320DiskDCB));
        if (ddcb == Nil)
            err = IFXENOMEMORY;
        else {
            eca = (ECA *) AllocateMemory2(sizeof(ECA));
            if (eca == Nil) {
                FreeMemory2((CB *) ddcb, sizeof(MVME320DiskDCB));
                err = IFXENOMEMORY;
            } else {
                ecat->eca_table[drive] = (ECA *)
                    ((unsigned long) eca + MVME320_Offset);
                ddcb->drive = drive;
                dcb->device_type = IFXDDISK;
                dcb->dt = (CB *) ddcb;
                err = RET_OK;
            }
        }
    }

} else {
    ddcb = (MVME320DiskDCB *) dcb->dt;
    drive = ddcb->drive;
    eca = (ECA *) ((unsigned long) ecat->eca_table[drive] -
                                            MVME320_Offset);
    err = RET_OK;
}

if (!err)
    switch (functionCode) {
            case IFXFINSTALL:
                err = MVME320_InstallDevice(ddcb, eca, pl->u1.eca,
                                            pl->u1.totalCylinders);
                break;
            case IFXFREMOVE:
                FreeMemory2((CB *) eca, sizeof(ECA));
                ecat->eca_table[drive] = Nil;
                FreeMemory2((CB *) ddcb, sizeof(MVME320DiskDCB));
                err = RET_OK;
                break;
            case IFXFREADS:
            case IFXFWRITES:
                err = MVME320_ReadWriteSectors(ddcb, eca, functionCode,
                            pl->u2.starting_sector, pl->u2.buffer,
                            pl->u2.number_of_sectors, pl->u2.actual_count);
                break;
            case IFXFIOCTL:
                switch (pl->u4.opcode) {
                case IFXOGGEOM:
                    dc = pl->u4.geometry;
                    dc->sector_size = ddcb->sectorSize;
                    dc->sectors_per_track = ddcb->sectorsPerTrack;
                    dc->tracks_per_cylinder = ddcb->tracksPerCylinder;
                    dc->total_cylinders = ddcb->totalCylinders;
                    dc->total_sectors = ddcb->totalSectors;
                    err = RET_OK;
                    break;
                case IFXOFMTTRK:
                    err = MVME320_FormatTrack(ddcb, eca,
                            pl->u3.cylinder, pl->u3.track);
                    break;
                default:
                    err = IFXENOTIMP;
                    break;
                }
                break;
            default:
                err = IFXENOTIMP;
                break;
            }

        /* Allow other users to access the disk controller */

        UnlockSemaphore(&ecat->mutex);

        return err;
}
```

B.8 *mvme320i.a68*

```
* Assembly language portion of MVME320 disk device driver

ECALEN  EQU     $58

        INCLUDE vrtxvisi.inc

        SECTION 0

        XDEF    .MVME320
        XDEF    .MVME320_CSR,.MVME320_Offset,.MVME320_InterruptVector
        XDEF    .MVME320_InterruptServiceRoutine
        XREF    .MVME320_DeviceDriver

.MVME320:
        BRA     .MVME320_DeviceDriver
.MVME320_CSR:
        DC.L    $FFFFB000        * default MVME320 CSR address
.MVME320_Offset:
        DC.L    0                * default VME bus address offset
.MVME320_InterruptVector:
        DC.W    $0060            * default interrupt vector number

.MVME320_InterruptServiceRoutine:
        MOVE.L  D0,-(SP)         * save registers
        MOVEM.L D1/A0-A1,-(SP)
        MOVEA.L .MVME320_CSR(PC),A1
        MOVE.B  7(A1),D1         * find address of ECA table and put in A0
        LSL.L   #8,D1
        MOVE.B  5(A1),D1
        LSL.L   #8,D1
        MOVE.B  3(A1),D1
        LSL.L   #8,D1
        MOVE.B  1(A1),D1
        LSL.L   #1,D1
        SUB.L   .MVME320_Offset(PC),D1
        MOVEA.L D1,A0
        BTST.B  #4,11(A1)        * check whether interrupt is from drive 0
        BEQ.S   not0
        BCLR.B  #4,13(A1)        * acknowledge the interrupt
        MOVE.L  0(A0),A0         * put address of ECA for drive 0 into A0
        BRA.S   common
not0:
        BTST.B  #5,11(A1)        * check whether interrupt is from drive 1
        BEQ.S   not1
        BCLR.B  #5,13(A1)        * acknowledge the interrupt
        MOVE.L  4(A0),A0         * put address of ECA for drive 1 into A0
        BRA.S   common
```

```
not1:
        BTST.B   #6,11(A1)        * check whether interrupt is from drive 2
        BEQ.S    not2
        BCLR.B   #6,13(A1)        * acknowledge the interrupt
        MOVE.L   8(A0),A0         * put address of ECA for drive 2 into A0
        BRA.S    common
not2:
        BTST.B   #7,11(A1)        * check whether interrupt is from drive 3
        BEQ.S    not3
        BCLR.B   #7,13(A1)        * acknowledge the interrupt
        MOVE.L   12(A0),A0        * put address of ECA for drive 3 into A0
common:
        SUBA.L   .MVME320_Offset(PC),A0
        ADDA.W   #ECALEN,A0       * compute mailbox address
        MOVEQ.L  #SCFPOST,D0      * wake up task waiting for this interrupt
        TRAP     #VRTX
not3:
        MOVEM.L  (SP)+,D1/A0-A1   * restore registers
        MOVEQ.L  #UIFEXIT,D0      * return from interrupt via VRTX
        TRAP     #VRTX

        END
```

B.9 *ifxmv320.h*

```
/* Event control area */

typedef struct {
    unsigned char      command_code;
    unsigned char      main_status;
    unsigned short     extended_status;
    unsigned char      maximum_retries;
    unsigned char      actual_retries;
    unsigned char      dma_type;
    unsigned char      command_option;
    unsigned long      buffer_address;
    unsigned short     buffer_length;
    unsigned short     actual_count;
    unsigned short     cylinder;
    unsigned char      surface;
    unsigned char      sector;
    unsigned short     current_position;
    unsigned short     reserved1[5];
    unsigned char      pre_index_gap;
    unsigned char      post_index_gap;
    unsigned char      sync_byte_count;
    unsigned char      post_id_gap;
    unsigned char      post_data_gap;
    unsigned char      address_mark_count;
    unsigned char      sector_length_code;
    unsigned char      fill_byte;
    unsigned short     reserved2[3];
    unsigned char      drive_type;
    unsigned char      number_of_surfaces;
    unsigned char      sectors_per_track;
    unsigned char      stepping_rate;
    unsigned char      head_settling_time;
    unsigned char      head_load_time;
    unsigned char      seek_type;
    unsigned char      phase_count;
    unsigned short     low_write_current_track;
    unsigned short     precompensation_track;
    unsigned short     ecc_remainder[3];
    unsigned short     append_ecc_remainder[3];
    unsigned long      reserved3;
    unsigned long      working_area[3];
    unsigned short     reserved4;
    /* used by device driver only */

    char                *mailbox;
    unsigned long       reserved5;
} ECA;

typedef struct {
    ECA                 *eca_table[4];
    IFXSEMA             mutex;
    long                reserved[2];
} ECATable;

/* MVME320 disk device control block */

typedef struct {
    unsigned short sectorSize;          /* sector size in bytes */
    unsigned short sectorsPerTrack;     /* number of sectors per track */
    unsigned short tracksPerCylinder;   /* number of tracks per cylinder */
    unsigned short totalCylinders;      /* total number of cylinders on disk */
    unsigned long totalSectors;         /* total number of sectors on disk */
    unsigned char drive;                /* drive number (0 to 3) */
    char reserved1[3];
    long reserved2[4];
} MVME320DiskDCB;

From glenn@ready@ Mon Apr 24 11:41:32 1989
Return-Path: <glenn@ready@>
Received: from harvax.RDYNET by hollywood.sun.com (3.2/SMI-3.2)
        id AA01807; Mon, 24 Apr 89 11:41:30 PDT
Received: by harvax.RDYNET (5.51/)
        id AA00687; Mon, 24 Apr 89 11:37:17 PST
Received: by ready.RDYNET (3.2/)
        id AA07365; Mon, 24 Apr 89 11:41:38 PDT
Date: Mon, 24 Apr 89 11:41:38 PDT
From: glenn@ready@ (Glenn Kasten)
Message-Id: <8904241841.AA07365@ready.RDYNET>
To: cindy@
Subject: ifxmv320.h
Status: R

/* $Header: ifxmv320.h,v 1.3 89/04/19 16:55:17 glenn Exp $ */
/* Event control area */

typedef struct {
    unsigned char       command_code;
    unsigned char       main_status;
    unsigned short      extended_status;
    unsigned char       maximum_retries;
    unsigned char       actual_retries;
    unsigned char       dma_type;
    unsigned char       command_option;
    unsigned long       buffer_address;
    unsigned short      buffer_length;
    unsigned short      actual_count;
    unsigned short      cylinder;
    unsigned char       surface;
    unsigned char       sector;
    unsigned short      current_position;
    unsigned short      reserved1[5];
    unsigned char       pre_index_gap;
    unsigned char       post_index_gap;
    unsigned char       sync_byte_count;
    unsigned char       post_id_gap;
    unsigned char       post_data_gap;
    unsigned char       address_mark_count;
    unsigned char       sector_length_code;
    unsigned char       fill_byte;
    unsigned short      reserved2[3];
    unsigned char       drive_type;
    unsigned char       number_of_surfaces;
    unsigned char       sectors_per_track;
    unsigned char       stepping_rate;
    unsigned char       head_settling_time;
    unsigned char       head_load_time;
    unsigned char       seek_type;
    unsigned char       phase_count;
    unsigned short      low_write_current_track;
    unsigned short      precompensation_track;
    unsigned short      ecc_remainder[3];
    unsigned short      append_ecc_remainder[3];
    unsigned long       reserved3;
    unsigned long       working_area[3];
    unsigned short      reserved4;

    /* used by device driver only */

    char                *mailbox;
    unsigned long       reserved5;
} ECA;

typedef struct {
    ECA                 *eca_table[4];
    IFXSEMA             mutex;
    long                reserved[2];
} ECATable;

/* MVME320 disk device control block */
```

```
typedef struct {
    unsigned short sectorSize;          /* sector size in bytes */
    unsigned short sectorsPerTrack;     /* number of sectors per track */
    unsigned short tracksPerCylinder;   /* number of tracks per cylinder */
    unsigned short totalCylinders;      /* total number of cylinders on disk */
    unsigned long totalSectors;         /* total number of sectors on disk */
    unsigned char drive;                /* drive number (0 to 3) */
    char reserved1[3];
    long reserved2[4];
} MVME320DiskDCB;
```

B.10 *rf3500.c*

```
/* IFX disk driver for Ciprico Rimfire 3500/3510 VMEbus SCSI host adapter */

/* Installation instructions:

Compilation:

        This driver consists of two source files

                rf3500.c        Main driver code
                rf3500i.a68     Interrupt handler

        A makefile is included which will build the driver.  The driver
        uses a few external functions which must be supplied by the
        user: malloc, free, and bzero.  In addition, the printf
        function is called if you define the macro DEBUG (see rf3500_printf
        below).  These functions are all standard functions available
        in any C run-time library, including the Ready Systems RTL
        product.

Installing driver with IFX:

        The general method looks like this:

        int status;
        static IFXGEOMETRY disk_geometry = {
                SECTOR_SIZE, SECTORS_PER_TRACK, TRACKS_PER_CYLINDER,
                TOTAL_CYLINDERS, TOTAL_SECTORS
        };
        extern int driver();

        status = ifx_driver("RF3500", driver);
        status = ifx_install("device:", "RF3500", UNIT, &disk_geometry);

        where UNIT is a combination of several things.

         -------------------------------
        |       |       |       |       |
        | FLOPPY| ID 2  | ID 1  | ID 0  |
        |       |       |       |       |
         -------------------------------

        INIT should be 1 when installing the first disk,
        and 0 for all other disks.

        FLOPPY should be 1 when installing a floppy disk,
        or 0 for installing a SCSI disk.

        ID 0, 1, and 2 should be the SCSI device ID number
        for installing a SCSI disk.  ID 0 and 1 should be
        the floppy ID number when installing a floppy disk,
        and ID 2 should always be 0 in this case.

        Example of installing two 5-1/4" 360 Kbyte floppies with ID 0 and 1,
        and one SCSI disk with ID 6.

        static IFXGEOMETRY floppy_geometry = {512,9,2,40,720L};
        static IFXGEOMETRY maxtor_geometry = {512,36,7,1224,308448L};
        status = ifx_driver("RF3500", driver);
        status = ifx_install("floppy0:", "RF3500", 0x08, &floppy_geometry);
        status = ifx_install("floppy1:", "RF3500", 0x09, &floppy_geometry);
        status = ifx_install("scsi:", "RF3500", 0x06, &maxtor_geometry);

For further information:

        Please consult your IFX Device Driver Developer's Guide and the
        Ciprico Rimfire 3500 User's Manual.

*/

#include <compiler.h>
#include <vrtxvisi.h>
#include <ifxvisi.h>

/* Define some helpful macros */

#define Nil     0
#define CB      char

/* Constants */

#define RF3500_ADDRESS  0xFFFFEF00      /* Rimfire 3500 board address */
#define INT_VECTOR      0xFE            /* VME bus interrupt vector */
#define INT_LEVEL       2               /* VME bus interrupt level */
#define VME_ADDRESS     0               /* offset of VME bus addresses */
#define INT_TIMEOUT     60000L          /* 10 minutes at 100 ticks/sec */
#define RESET_TIMEOUT   1000000L        /* max time to reset Rimfire */
#define ENTER_TIMEOUT   1000L           /* max time to enter a command */
#define printf          rf3500_printf

/* Type declarations */

typedef unsigned short word;
typedef unsigned char byte;

/* Ciprico Header Block

   +------------------------------------------------------------+
   |                    command identifier                      |
   +----------------------------+---------------+---------------+
   |            Reserved        | Addr mod      | Target ID     |
   +-----------------------------+---------------+----------------+
   |                    VME MEMORY ADDRESS                        |
   +--------------------------------------------------------------+
   |                    Transfer Count                            |
   +-------------+---------------+---------------+----------------+

*/

struct std_parmblk {            /* STANDARD PARAMETER BLOCK                   */
        long    id;             /* unique command identifier                  */
        byte    reserved;       /* reserved                                   */
        byte    flags;          /* flags                                      */
        byte    addrmod;        /* address modifier used to access VME memory*/
        byte    targetid;       /* SCSI target ID                             */
        long    vmememaddr;     /* VME address to read from or write to       */
        long    tcount;         /* transfer count                             */
};

/* SCSI Command Block

General layout:

   +----+--------+-------+-------+-------+-------+-------+--------+
   |                    COMMAND CODE                              |
   +----+--------+-------+-------+-------+-------+-------+--------+
   |       LUN           |      REST OF BYTE 1                    |
   +-------------+---------------+---------------+----------------+
   |                    BYTES 2 TO 11                             |
   +-------------+---------------+---------------+----------------+

Read layout:

   +----+--------+-------+-------+-------+-------+-------+--------+
   |                    COMMAND CODE = $08                        |
   +----+--------+-------+-------+-------+-------+-------+--------+
   |       LUN           |      LOGICAL BLOCK ADDRESS (MSB)       |
   +-------------+---------------+---------------+----------------+
   |                    LOGICAL BLOCK ADDRESS                     |
   +--------------------------------------------------------------+
   |                    LOGICAL BLOCK ADDRESS (LSB)               |
   +--------------------------------------------------------------+
   |                    NUMBER OF BLOCKS                          |
   +--------------------------------------------------------------+
   |                    UNUSED (7 BYTES)                          |
   +--------------------------------------------------------------+

Write layout:

   +----+--------+-------+-------+-------+-------+-------+--------+
   |                    COMMAND CODE = $0A                        |
   +----+-------+-------+-------+-------+-------+-------+-------+
   |       LUN         |       LOGICAL BLOCK ADDRESS (MSB)       |
   +------------+---------------+---------------+---------------+
   |                    LOGICAL BLOCK ADDRESS                    |
   +------------------------------------------------------------+
   |                    LOGICAL BLOCK ADDRESS (LSB)              |
   +------------------------------------------------------------+
   |                    NUMBER OF BLOCKS                         |
   +------------------------------------------------------------+
   |                    UNUSED (7 BYTES)                         |
   +------------------------------------------------------------+

*/

struct  scsicmdblk {            /* SCSI COMMAND DESCRIPTOR BLOCK             */
        byte    cmd;            /* byte 0 is the cmd code for operation      */
        byte    lun;            /* logical unit number                       */
        byte    lba;            /* logical block address                     */
        byte    lbalsb;         /* logical block address (LSB)               */
        byte    numblocks;      /* number of blocks                          */
        byte    vu;
        byte    byte6;
        byte    byte7;
        byte    byte8;
        byte    byte9;
        byte    byte10;
        byte    byte11;
};

/* Ciprico Trailer Block */

struct intrblk {
        word                   resv0;           /* reserved             */
        word                   intr;            /* interrupt vector/level */
        long                   resv1;           /* reserved, must be 0  */
};

/* Ciprico Status Block

   +------------+---------------+---------------+---------------+
   |                    Command Identifier                       |
   +------------+---------------+---------------+---------------+
   |    0       |  SCSI status  |     Error     |     Flags     |
   +------------+---------------+---------------+---------------+
   |                    Extra  Information                       |
   +------------+---------------+---------------+---------------+

*/

struct  stdstatusblk {          /* STANDARD STATUS BLOCK                     */
        long    id;             /* command identifier generating status      */
        byte    reserved;       /* reserved                                 */
        byte    scsistatus;     /* SCSI status, device specific             */
        byte    error;          /* rf3500 specific error                    */
        byte    flag;           /* indicates type of status                 */
        byte    cc;             /* class/code                               */
        byte    segment;        /* segment                                  */
        byte    scsiflags;      /* SCSI flags                               */
        byte    infobyte3;      /* information byte3                        */
        byte    infobyte4;      /* information byte4                        */
        byte    infobyte5;      /* information byte5                        */
        byte    infobyte6;      /* information byte6                        */
        byte    exlen;          /* extra length                             */
};

/* Complete Command + Status Block */

struct type0 {
        struct std_parmblk     type0pblk;       /* type0 parameter block*/
        struct scsicmdblk      type0cmdblk;     /* type0 command block  */
        struct intrblk         type0intrblk;    /* type0 interrupt block */
        struct stdstatusblk    type0statusblk;  /* type0 status blk     */
};

/* SCSI Basic Mode Select Block                         */

struct mode_select {
        byte    byte0;          /* reserved             */
        byte    medium_type;    /* medium type          */
        byte    bufmod_spd;     /* buffered mode & speed*/
                                /* (tape exclusive)     */
        byte    blk_des_len;    /*block descriptorlength*/
        byte    density_code;   /* Density code         */
        byte    nblk[3];        /* number of blocks     */
                                /* (msb) - (lsb)        */
        byte    byte8;          /* reserved             */
        byte    blen[3];        /* block length         */
                                /* (msb)-(lsb)          */
        /* vendor unique parameter              */
};

/* SCSI Page 5 Mode Select Block */

struct page_5 {
        byte    pagecode;       /* must be 5            */
        byte    page_length;
        word    xfer_rate;      /* transfer rate        */
        byte    nheads;         /* number of heads      */
        byte    spt;            /* sector per track     */
        word    nbps;           /*number of bytes/sector*/
        word    ncyls;          /* number of cylinders  */
        word    s_wpre;         /* starting cylinder    */
                                /* write pre comp       */
        word    s_rwc;          /* starting cylinder    */
                                /* reduced write current*/
        word    dsr;            /* drive step rate      */
        byte    dspw;           /* drive step pulsewidth*/
        byte    hd_st_delay;    /* head settle delay    */
        byte    on_delay;       /* motor on delay       */
        byte    off_delay;      /* motor off delay      */
        byte    trdy;           /* drive provides a true*/
                                /* ready signal         */
        byte    hd_ld_delay;    /* head load delay      */
        byte    ssn_s0;         /*starting sector #,    */
                                /* side 0               */
        byte    ssn_s1;         /* starting sector #,   */
                                /* side 1               */
};

/* SCSI Page 20 Mode Select Block (Ciprico floppy disk configuration) */

struct page_20 {
        byte    page_code;
        byte    page_length;
        byte    post_index;     /* post index gap       */
        byte    inter_sector;   /* inter sector gap     */
        byte    tverify;        /* seek verification    */
        byte    tsteps;         /* steps per track      */
        byte    resv0;          /* reserved set to 0H   */
        byte    resv1;          /* reserved set to 0H   */
};

/* SCSI Complete Mode Select Block */

struct  init_mode_select {
        struct  mode_select     init_mode;
        struct  page_5          page_5;
        struct  page_20         page_20;
};

/* Parameter list passed to driver by IFX */

typedef union {

    /* IFXFREADS or IFXFWRITES */

    struct {
        long    sector_position;        /* starting disk sector number  */
        byte    *buffer_address;        /* address of the read/write buffer */
        long    number_sectors;         /* # of sectors to be transferred */
```

```
            long    *actual_count;          /* # of sectors successfully xfrd */
        } u1;

        /* IFXFIOCTL with IFXOGGEOM */

        struct {
            int control_opcode;             /* control opcode = IFXOGGEOM */
            IFXGEOMETRY *disk_geometry;     /* pointer to disk geometry block */
        } u2;

        /* IFXFINSTALL */

        struct {
            int unit;                       /* 0 to 7=SCSI, 8 to 11 = floppy */
            IFXGEOMETRY *disk_geometry;     /* pointer to disk geometry block */
        } u3;

    } PL;

    /* Structure specific for each device installed */

    typedef struct {
            int     target_ID;
            int     LUN;
            IFXGEOMETRY disk_geometry;
    } RF3500DCB;

    /* Global variables */

    IFXSEMA sem;    /* semaphore for mutual exclusion */
    int mbox;       /* mailbox posted to by interrupt service routine */

    /* External function declarations */

    extern void LockSemaphore(), UnlockSemaphore(), FreeMemory2(),
                bzero(), printf(), isr();
    extern CB *AllocateMemory2();

    /* Function declarations */

    /*

            The function mode_select_floppy() initilizes the mode
            select buffer, assigns values to it and the vendor unique
            structures page_5 and page_20 appropriate for a floppy disk,
            and then calls mode_select to actually set it up.

    */

    int mode_select_floppy(dt)
    RF3500DCB       *dt;
```

```
{
        struct init_mode_select mb;

        /* Initialize mode select buffer */

        bzero((char *) &mb, sizeof(struct init_mode_select));

        /* Assign values to the mode select buffer */

        mb.init_mode.medium_type = 0x12;/* medium type 48 tpi   */
        mb.init_mode.blk_des_len = 8;   /* block length         */
        mb.init_mode.blen[1] = 2;       /* block length =512    */

        mb.page_5.pagecode = 5;         /* page 5               */
        mb.page_5.page_length = 22;     /* page length          */
        mb.page_5.xfer_rate = 0xFA;     /* transfer rate 250K   */
        mb.page_5.nheads = 2;           /* number of heads      */
        mb.page_5.spt = 9;              /* sectors per track    */
        mb.page_5.nbps = 512;           /* bytes per sector     */
        mb.page_5.ncyls = 40;           /* number of cylinders  */
        mb.page_5.s_wpre = 255;         /* write precomp        */
        mb.page_5.s_rwc = 255;          /* reduced write current*/
        mb.page_5.dsr = 4096;           /* drive step rate      */
        mb.page_5.dspw = 0;             /* drive step pulsewidth*/
        mb.page_5.hd_st_delay = 40;     /* head settle delay    */
        mb.page_5.on_delay = 40;        /* motor on delay       */
        mb.page_5.off_delay = 40;       /* motor off delay      */
        mb.page_5.hd_ld_delay = 10;     /* head load delay      */
        mb.page_5.ssn_s0 = 1;           /* starting sector #    */
        mb.page_5.ssn_s1 = 1;           /* starting sector #    */

        mb.page_20.page_code = 0x20;    /* page 20              */
        mb.page_20.page_length = 6;     /* page length          */
        mb.page_20.tsteps = 1;          /* steps per track      */

        return mode_select((struct mode_select *) &mb, sizeof(mb),dt);
}

/*
        The mode_select function takes as a parameter a pre-initialized
        mode select parameter block, and then calls the do_command.
        This is an optional command in the common command set, but is
        mandatory for the operation of the floppy disk option.

*/

int mode_select(mode_buff, mode_size,dt)
struct mode_select *mode_buff;
int mode_size;
RF3500DCB *dt;
{
        struct  type0   pb;

        bzero((char *) &pb,sizeof(struct type0));

        pb.type0pblk.targetid = dt->target_ID;  /* floppy ID             */
        pb.type0pblk.addrmod = 0x39;            /* vme address modifier */
        pb.type0pblk.vmememaddr =(long)mode_buff + VME_ADDRESS;
        pb.type0pblk.tcount = mode_size;        /* transfer count       */
        pb.type0cmdblk.cmd = 0x15;              /* mode select command  */
        pb.type0cmdblk.numblocks = mode_size;   /* number of bytes requested*/

        return docommand(dt, &pb);
}

/*
        The function rezero initializes the parameter and command blocks
        for the rezero command and then calls the docommand to execute
        the command. This command causes a seek to track zero on the
        specified disk unit.

*/

int rezero(dt)
RF3500DCB *dt;

{
        struct  type0   pb_rezero;

        bzero((char *) &pb_rezero,sizeof(struct type0));
        pb_rezero.type0pblk.targetid =dt->target_ID;
        pb_rezero.type0pblk.addrmod =0x39;
        pb_rezero.type0cmdblk.cmd = 1;  /* rezero unit command          */

        return docommand(dt, &pb_rezero);

}

/*
        The function format_unit initializes the parameter and command
        blocks for the format command and then calls docommand to
        execute the command.  This command erases the entire disk and
        puts new timing information on the tracks.

*/

int format_unit(dt)
RF3500DCB *dt;

{
        struct type0 pb_format;
        bzero((char *) &pb_format,sizeof(struct type0));
        pb_format.type0pblk.targetid =dt->target_ID;
        pb_format.type0pblk.addrmod = 0x39;     /* VME address modifier */
        pb_format.type0cmdblk.cmd = 4;          /* format unit command  */

        return docommand(dt, &pb_format);
}

/*
        The function driver_reads(dt,rdwr) initializes the parameter
        block & the command block for the read command and then calls
        docommand to execute the command.

*/

int driver_reads(dt, rdwr)
RF3500DCB *dt;
PL *rdwr;

{
        struct type0 pb_read;
        long    high,middle,l_unit_num;
        int     err;

        /* special check for zero sectors */

        if (rdwr->ul.number_sectors == 0) {
                *rdwr->ul.actual_count  = 0;
                return RET_OK;
        }

        /* work-around bug in IFX 1.04; reads and writes last sector in mount
*/

        if (rdwr->ul.sector_position == dt->disk_geometry.total_sectors - 1L &&
                rdwr->ul.number_sectors == 1L) {
                *rdwr->ul.actual_count  = 1L;
                return RET_OK;
        }

        /* special check for transferring more than 256 sectors at a time */

        if (rdwr->ul.number_sectors > 256) {
                *rdwr->ul.actual_count  = 0;
                return IFXEBADLEN;
        }

        bzero((char *) &pb_read,sizeof(struct type0));

        pb_read.type0pblk.targetid =dt->target_ID;/* floppy ID          */
        pb_read.type0pblk.addrmod=0x39; /* VME address modifier         */
        pb_read.type0pblk.vmememaddr=
                (long)(rdwr->u1.buffer_address + VME_ADDRESS);
        pb_read.type0pblk.tcount = rdwr->u1.number_sectors *
            dt->disk_geometry.sector_size; /* transfer count             */
        pb_read.type0cmdblk.cmd = 8;    /* read command                 */

        high = (rdwr->u1.sector_position>> 16) & 0x1F;
        l_unit_num = (dt->LUN << 5);
        pb_read.type0cmdblk.lun =(l_unit_num | high);

        middle = ((rdwr->u1.sector_position >> 8) & 0xFF);
        pb_read.type0cmdblk.lba = middle;/* logical block address*/

        pb_read.type0cmdblk.lbalsb= rdwr->u1.sector_position;
                                        /*logical block address(lsb)*/

        pb_read.type0cmdblk.numblocks =(byte)rdwr->u1.number_sectors;
                                         * number of sectors*/

        err = docommand(dt, &pb_read);
        *rdwr->u1.actual_count  = err ? 0 : rdwr->u1.number_sectors;
        return err;
}

/*
        The function driver_writes(dt, rdwr) initializes the parameter
        block and the command block for the write command. It then
        calls the docommand to execute write command.

*/

int driver_writes(dt, rdwr)
RF3500DCB *dt;
PL *rdwr;

{
        struct type0 pb_write;
        long    high,middle,l_unit_num;
        int     err;

        /* special check for zero sectors */

        if (rdwr->u1.number_sectors == 0) {
                *rdwr->u1.actual_count  = 0;
                return RET_OK;
        }

        /* work-around bug in IFX 1.04; reads and writes last sector in mount
*/

        if (rdwr->u1.sector_position == dt->disk_geometry.total_sectors - 1L &&
                rdwr->u1.number_sectors == 1L) {
```

```
                *rdwr->ul.actual_count  = 1L;
                return RET_OK;
        }

        /* special check for transferring more than 256 sectors at a time */

        if (rdwr->ul.number_sectors > 256) {
                *rdwr->ul.actual_count  = 0;
                return IFXEBADLEN;
        }

        bzero((char *) &pb_write,sizeof(struct type0));

        pb_write.type0pblk.targetid = dt->target_ID;    /* floppy ID    */
        pb_write.type0pblk.addrmod=0x39;        /* VME address modifier */
        pb_write.type0pblk.vmememaddr=
                        (long)(rdwr->ul.buffer_address+VME_ADDRESS);
        pb_write.type0pblk.tcount = rdwr->ul.number_sectors *
            dt->disk_geometry.sector_size; /* transfer count          */
        pb_write.type0cmdblk.cmd =0x0A; /*write command               */

        high = (rdwr->ul.sector_position >> 16) & 0x1F;
        l_unit_num = (dt->LUN << 5);
        pb_write.type0cmdblk.lun =(l_unit_num | high);

        middle = ((rdwr->ul.sector_position >> 8) & 0xFF);
        pb_write.type0cmdblk.lba = middle;/* logical block address*/

        pb_write.type0cmdblk.lbalsb= rdwr->ul.sector_position;
                                        /*logical block address(lsb)*/

        pb_write.type0cmdblk.numblocks=(byte)rdwr->ul.number_sectors;
                                                /*number of sectors*/

        err = docommand(dt, &pb_write);
        *rdwr->ul.actual_count  = err ? 0 : rdwr->ul.number_sectors;
        return err;
}

/*

        The function request_sense initializes the parameter block and
        the command block for the request sense command and then calls
        the docommand to execute the command. This command returns the
        sense data to the indicated area in memory.

*/

int request_sense(dt)
RF3500DCB *dt;

{
        struct type0    pb_request_sense;
        int err;
        byte    sense_buffer[8];

        bzero((char *) &pb_request_sense,sizeof(struct type0));

        pb_request_sense.type0pblk.targetid =dt->target_ID;     /* floppy ID */
        pb_request_sense.type0pblk.addrmod =0x39;/* VME address modifier*/
        pb_request_sense.type0pblk.vmememaddr =
                (long)( &sense_buffer[0] + VME_ADDRESS);
        pb_request_sense.type0pblk.tcount = 4;
                                                /* transfer count       */
        pb_request_sense.type0cmdblk.cmd = 3;   /* request sense command*/

        err = docommand(dt, &pb_request_sense);
        if (err)
            return err;
        switch (sense_buffer[0]) {
        case 0x00:
            err = RET_OK;
            break;
        case 0x02:
            err = IFXESEEKFAIL;
            break;
        case 0x25:
            err = IFXEBADUNIT;
            break;
        case 0x27:
            err = IFXERDONLYM;
            break;
        case 0x28:
            err = IFXEMEDCHANGE;
            break;
        case 0x29:
            printf("RF3500: power restored\n");
            err = RET_OK;
            break;
        default:
            printf("RF3500: sense %02X\n", sense_buffer[0]);
            err = IFXEIOERR;
            break;
        }
        return err;
}

/*
        This function sets up the Address buffer port (EF00), Channel
        Attention Port (EF09),and checks the Board Status Port (EF11)
        and executes the command. It then returns an error message in
        error.

*/

int docommand(dt, pb_command)
RF3500DCB *dt;
struct  type0   *pb_command;

{
        byte    *channel_atten, *board_status,d0;
        word    *address_buffer;
        int     zero = 0;
        int     error;
        long    address, i;

retry:

/*      initilize the flag in the status block                                */
        pb_command -> type0statusblk.flag =zero;

/*      set up the interrupt field in the type0 parameter block */
        pb_command -> type0intrblk.intr = INT_VECTOR | (INT_LEVEL << 8);
/*      Check the Status Port                                                 */

        board_status = (byte*) (RF3500_ADDRESS + 0x11);
        if (!(*board_status & 2)) {
            printf("RF3500: not ready\n");
            return IFXEIOERR;
        }
        d0 = *board_status & 1;

/*      set up the address buffer port                                        */

        address_buffer = (word*) RF3500_ADDRESS;
        *address_buffer = 0x8039;       /* control and AM bits for PB         */
        address = VME_ADDRESS + (long)pb_command;
        *address_buffer = address >> 16;        /* send PB address MSW        */
        *address_buffer = address;              /* send PB address LSW        */

/*      Issue  command by writing zero into the channel attention port        */

        channel_atten = (byte *) (RF3500_ADDRESS + 0x09);
        mbox= 0;
        *channel_atten = zero;          /* execute the type1 command          */

/*      Wait for command to be entered */

        for (i = 0L; (*board_status & 1) == d0; ++i)
            if (i > ENTER_TIMEOUT) {
                printf("RF3500: command not entered\n");
                return IFXEIOERR;
            }

/* Wait for interrupt */

        sc_pend(&mbox, INT_TIMEOUT, &error);
        if (error) {
            printf("RF3500: no interrupt\n");
            return IFXEIOERR;
        }

        if (!(pb_command->type0statusblk.flag & 128)) {
            printf("RF3500: interrupt before done\n");
            return IFXEIOERR;
        }

/* Convert error code to IFX numbering system */

        if (pb_command->type0statusblk.flag & 64) == 0)
            error = RET_OK;
        else {
            switch (pb_command->type0statusblk.error) {
            case 0x1E:
                error = IFXEBADUNIT;
                break;
            case 0x23:
                switch(pb_command->type0statusblk.scsistatus){
                case 0x02:      /* check condition */
                    if (pb_command->type0pblk.targetid != 0xFE) {
                        if (pb_command->type0cmdblk.cmd != 3)
                            error = request_sense(dt);
                        else
                            error = IFXEIOERR;
                    } else
                        switch ( pb_command->type0statusblk.scsiflags) {
                        case 0x21:
                            error = IFXEBADPOSN;
                            break;
                        case 0x25:
                            error = IFXEBADUNIT;
                            break;
                        case 0x41:
                            error = IFXEIOTIMOUT;
                            break;
                        case 0x43:
                            error = IFXESEEKFAIL;
                            break;
                        case 0x44:
                        case 0x46:
                            error = IFXECRCERR;
                            break;
                        case 0x45:
                            error = IFXERDONLYM;
                            break;
                        case 0x47:
                            error = IFXESECNTFND;
                            break;

                        case 0x42:
                        case 0x20:
                        case 0x26:
                        case 0x29:
                        case 0x40:
                        case 0x48:
                        case 0x49:
                        default:
                            error = IFXEIOERR;
                            break;
                        }
                    break;
                case 0x08:      /* busy */
                    printf("RF3500 busy, retrying...\n");
                    sc_delay(100L);
                    goto retry;
                default:
                    printf("RF3500 SCSI status %02X\n",
                            pb_command->type0statusblk.scsistatus);
                    error = IFXEIOERR;
                    break;
                }
                break;
            default:
                printf("RF3500 error %02X\n",
                    pb_command->type0statusblk.error);
                error = IFXEIOERR;
                break;
            }
        }
        return error;

}

/*


        The function reset does a write to the Reset Port(EF19) This
        causes a reset of the adapter. This action is similar to a
        hardware reset. The function also checks the status port for
        the RDY bit to be set.

*/

int reset()
{
        byte    *reset_port;
        byte    *board_status;
        int     zero = 0;
        long    i;

        /* Initialize interrupt vector to point to isr routine */

        * (void (**)()) (INT_VECTOR * 4) = isr;

        /* Reset the board */

        reset_port = (byte*) (RF3500_ADDRESS + 0x19);
        *reset_port = zero;

        /* Wait for board to finish initialization sequence */

        board_status = (byte*) (RF3500_ADDRESS + 0x11);
        for (i = 0L; !(*board_status & 2); ++i)
            if (i > RESET_TIMEOUT)
                return IFXEIOERR;
        return RET_OK;
}

/* Handle IFX device installation */

int driver_install(dcb_ptr, plist)
IFXDCB *dcb_ptr;
PL *plist;

{
        int status;
        RF3500DCB *dt;

        dt = (RF3500DCB *) AllocateMemory2(sizeof(RF3500DCB));
        if (dt == Nil)
            return IFXENOMEMORY;
        dt->target_ID = plist->u3.unit & 8 ? 0xFE : plist->u3.unit & 7;
        dt->LUN =       plist->u3.unit & 8 ? plist->u3.unit & 3 : 0;
        dt->disk_geometry = *plist->u3.disk_geometry;
        dcb_ptr->device_type = IFXDDISK;
        dcb_ptr->dt = (char *) dt;

        /* Discard the first error code, if any */
        (void) request_sense(dt);

        /* Perform mode select for floppy disk drives only */

        if (plist->u3.unit & 8) {
            status = mode_select_floppy(dt);
            if (status != RET_OK)
                return status;
        }

        /* Recalibrate the heads */

        status = rezero(dt);
        return status;
}

/* Handle IFX I/O control operation requests */

int driver_ioctl(dt, plist)
RF3500DCB *dt;
PL *plist;

{
        int status;

        switch (plist->u2.control_opcode) {
        case IFXOGGEOM:
            *plist->u2.disk_geometry = dt->disk_geometry;
            status = RET_OK;
            break;
        case IFXOFMTDSK:
            status = format_unit(dt);
            break;
        default:
            status = IFXENOTIMP;
            break;
        }
        return status;
}

/* This function is called when the device is removed */

int driver_remove(dt)
RF3500DCB *dt;

{
        FreeMemory2((CB *) dt, sizeof(RF3500DCB));
        return RET_OK;
}
```

```
/*
        The function driver is the main function that IFX calls.  The
        three parameters are the function code that tells the driver
        what to do, a pointer to its Device control block defined as
        the IFXDCB structure, and finally a pointer to a Parameter list
        containing any other parameters needed for the specified
        function.
*/

int driver(func_code, dcb_ptr, plist)
int func_code;
IFXDCB *dcb_ptr;
PL *plist;
{
        int status;
        RF3500DCB *dt = (RF3500DCB *) dcb_ptr->dt;

        switch (func_code) {
        case IFXFDRIVER:
            bzero((char *) &sem, sizeof(sem));
            status = reset();
            break;
        case IFXFRMDRIVER:
            status = IFXENOTIMP;
            break;
        case IFXFINSTALL:
            status = driver_install(dcb_ptr, plist);
            break;
        case IFXFREMOVE:
            status = driver_remove(dt);
            break;
        default:
            LockSemaphore(&sem);
            switch (func_code) {
            case IFXFREADS:
                status = driver_reads(dt, plist);
                break;
            case IFXFWRITES:
                status = driver_writes(dt, plist);
                break;
            case IFXFIOCTL:
                status = driver_ioctl(dt, plist);
                break;
            default:
                status = IFXENOTIMP;
                break;
            }
            UnlockSemaphore(&sem);
        }

        return status;
}

/* Define DEBUG if you want debugging messages printed on console terminal */

void rf3500_printf(fmt, args)
char *fmt;
int args;
{
#ifdef DEBUG
        char buf[80];
        int i;
        xprintf(buf,fmt,&args,0);
        for (i=0;buf[i];++i)
        {
        if (buf[i]=='\n')
                sc_putc('\r');
        sc_putc(buf[i]);
        }
#endif
}
```

B.11 *rf3500i.a68*

```
* Ciprico Rimfire 3500 IFX disk driver interrupt service routine

        XREF    .driver,.mbox
        XDEF    start,.isr

SCFPOST EQU     $08
UIFEXIT EQU     $11

        SECTION 0

start:
        BRA     .driver                 * first instruction of driver .

.isr:
        MOVE.L  D0,-(SP)                * save D0 for UI_EXIT
        MOVEM.L D1/A0,-(SP)             * save other registers used by ISR
        LEA     .mbox(PC),A0            * wake up task waiting for interrupt
        MOVEQ.L #1,D1
        MOVEQ.L #SCFPOST,D0
        TRAP    #0
        MOVEM.L (SP)+,D1/A0             * restore registers
        MOVEQ.L #UIFEXIT,D0             * reschedule and restore D0
        TRAP    #0

        END
```

Appendix C

Sample 386 Device Driver

READY
SYSTEMS

C.1 Introduction

This appendix contains source code for a 386 sample device driver. This device driver is for a CIPRICO RimFire 2500 SCSI disk controller, which implements the diskette driver in polling mode. You can use this sample device driver as a template for writing a custom driver for your device. This driver has been tested and is fully functional.

C.2 IFX Device Driver

This section contains an IFX driver for the RimFire 2500 disk controller. This driver is composed of an interface assembly routine, main C routine, specific operations routines, and message passing high-level services (such as preparing the message in the final structure, or checking the operation results). The lower-level services for transmit/receive messages or data are provided in Section C.3.

C.2.1 *descrp.inc*

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;; filename:  descrp.inc                                              ;;
;;                                                                    ;;
;; description:  This file defines the structures needed for accessing ;;
;;               the decriptors in the GDT.  It also contains several ;;
;;               descriptors for the driver.                          ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

DESC  STRUC
      lim_0_15       DW   0        ; imit bits (0..15)
      bas_0_15       DW   0        ; base bits (0..15)
      bas_16_23      DB   0        ; base bits (16..23)
      access         DB   0        ; access byte
      gran           DB   0        ; granularity byte
      bas_24_31      DB   0        ; base bits (24..31)
DESC  ENDS

GDT_ALIAS_DES        equ  08h      ; GDT alias
APPLIC_AS_DATA       equ  0B8h     ; application code as data desc
```

C.2.2 rfmain.asm

```
#include "descrp.inc"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                     ;;
;; This is an interface routine, which stores IFX DS + restores the DS ;;
;; of the driver - on enterance, and restores IFX DS on exit.          ;;
;;                                                                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.386P   ;Directive to Phar Lap assembler
    ;to give access to 386 protected commands

cseg    segment use32 'code' para public
    extrn   Rimfire2500Driver:near
    assume  cs:cseg
    public  rfmain

rfmain    PROC   far
   push    ebp
   mov     ebp,esp
   push    es
   push    ds
   mov     ax,APPLIC_AS_DATA
   mov     ds,ax
   les     eax,[ebp+018h]
   push    es
   push    eax
   les     eax,[ebp+010h]
   push    es
   push    eax
   mov     eax,[ebp+0Ch]
   push    eax
   call    Rimfire2500Driver
   add     esp,014h
   pop     ds
   pop     es
   pop     ebp
   DB      0CBH                    ; RETF - far return to the caller.
rfmain     ENDP

cseg    ends

    end
```

C.2.3 *ifxr2500.h*

```
/******************************************/
/*                                        */
/* Rimfire 2500 disk device control block */
/*                                        */
/******************************************/

typedef struct {
    IFXGEOMETRY    flp_struc;
    unsigned char drive;        /* drive number (0 to 3) */
    char reserved1[3];
    long reserved2[4];
} Rimfire2500DiskDCB;
```

C.2.4 *rf2500.h*

```
/*********************************************************************/
/* Description: Include file for Rimfire 2500 Driver.                */
/*              Adaptor specific and MultiBus II contents, defines,  */
/*              and some general miscellaneous stuff.                */
/*********************************************************************/
/* Some basic constants... */
#ifndef TRUE
#define TRUE    1
#define FALSE   0
#endif

/* Some basic types... */
typedef unsigned char   u_char;
typedef unsigned char   byte;
typedef unsigned short  word;
typedef unsigned long   dword;

/* The HOST ID : PORT ID at which the cpu board communicates. */
/* -- They are exist also in MPC.INC, for the MB2 library. -- */
#define CPU_HOST_ID           0x01
#define CPU_PORT_ID         0x0100

/* The HOST ID : PORT ID at which we communicate with our board. */
/* --- They are exist also in MPC.INC, for the MB2 library. ---- */
#define SLOT_NO                  5
#define RF2500_HOST_ID        0x05
#define RF2500_PORT_ID      0x2500

/* Record types to be used with ics_find_rec(). */
#define UNIT_DEF_RECORD     0xFE
#define HOST_ID_RECORD      0x10
#define FIRMWARE_RECORD     0x0F
```

```
/* Offsets within the Header Record for various registers. */
#define HD_GEN_STATUS       24  /* General Status */
#define HD_SUP_LEVEL        26  /* BIST Support Level */
#define HD_SLV_STATUS       29  /* BIST Slave Status */
#define HD_MSTR_STATUS      30  /* BIST Master Status */
#define HD_TEST_ID          31  /* BIST Test ID */
/* What the master status should be after power-up. */
#define HD_MS_REBOOT        0x20

/* BIST Slave Status Register defines */
#define SS_DO_VALID         0x01    /* BIST Data Out Valid for reading */
#define SS_RD_DI            0x02    /* Slave had read Data In value */
#define SS_TST_RUNNING      0x04    /* Slave is running a test */
#define SS_MORE             0x08    /* Slave has more bytes to send */
#define SS_IN_PROGRESS      0x10    /* BIST Test In Progress */
#define SS_ABORT_LAST_TEST  0x40    /* Aborted Last Test */
#define SS_FAIL_LAST_TEST   0x80    /* Failed Last Test */

/* Offset within the Unit Definition Record for various registers. */
#define UD_REC              32
#define UD_PCI_COMPAT       2   /* PCI Compatibility */
#define UD_GEN_STATUS       10  /* General Status register */
#define UD_PORTID           11  /* The Port ID */
#define UD_CTLR_INIT        13  /* The Controller Initialize register */
/* What the general status should be after power-up. */
#define UDGS_POR_COMPLETE   0xE0
/* What the general status should be after successful controller init */
#define UDGS_INIT_COMPLETE  0xE1

/* Offset within the Host ID Record for various registers. */
#define HID_REC             50
#define HID_HOSTID          2   /* The Host ID */

#define FIRM_REC            56

/* These are the board types that this driver can drive. */
#define RF2500_TYPE 0

/* Internal driver structure for keeping specific board information. */
typedef struct {
    word        port_id;        /* Port ID for the board */
    word        host_id;        /* Host ID for the board */
    u_char      bd_udf_rec;     /* The Unit Definition Record offset */
    u_char      bd_hostid_rec;  /* The Host ID Record offset */
    u_char      bd_tid;         /* A Transaction ID for Commands */
} BDINFO;

#define send   0
#define recv   1
#define UNS_STD_LEN   32    /* The length of a standard unsolicited msg */
```

```
#define UNS_TYPE       0    /* Unsolicited message type */
#define BRQ_TYPE      36    /* Buffer request message type */
#define MEMSIZE    0x200

/* The Peripheral Command Messages */
typedef struct {          /* rf2500 adaptor specific command */
    byte    targetid;
    byte    flags;
    byte    cmd;
    byte    tbd[12];
} ADPCM;

typedef struct {          /* rf2500 SCSI command */
    byte    targetid;
    byte    flags;
    byte    resvl;
    byte    scb[12];
} SCPCM;

typedef union {
    ADPCM   adpcm;
    SCPCM   scpcm;
} PCM;

typedef struct {
     byte     dst_addr;
     byte     src_addr;
     byte     msg_type;
     byte     resrvl;
     byte     prot_id;
     byte     tx_ctl;
     short    dst_port;
     short    src_port;
     byte     trnsc_id;
     byte     trnsc_ctl;
     byte     protoid;
     PCM      cmd_buf;
     long     buffer_len;   /* will be set by the asm routine */
} UNS_MSG;

typedef struct {
     byte     dst_addr;
     byte     src_addr;
     byte     msg_type;
     byte     liaison;
     long     buffer_len;   /* will be set by the asm routine */
     byte     prot_id;
     byte     tx_ctl;
     short    dst_port;
     short    src_port;
     byte     trnsc_id;
     byte     trnsc_ctl;
     byte     protoid;
     PCM      cmd_buf;
} BRQ_MSG;

/* RF2500 Adaptor specific commands using ADPCM */
#define AD_INITP    0x01    /* initialize parameters */
#define AD_UNITP    0x02    /* unit parameters */
#define AD_RES_DEV  0x03    /* reserve device */
#define AD_REL_DEV  0x04    /* release device */
#define AD_R_INITP  0x05    /* return (read) initialize parameters */
#define AD_R_UNITP  0x06    /* return unit parameters */
#define AD_R_ERTXT  0x07    /* return error text */
#define AD_R_BSTAT  0x08    /* return board statistics */
#define AD_BBCOPY   0x09    /* board to board copy */
#define AD_TUNEP    0x0a    /* set tuning parameters */
#define AD_R_TUNEP  0x0b    /* return tuning parameters */
#define AD_RWBFTST  0x0c    /* read/write buffer test */

/* The value that pcm_protoid always gets set to... */
#define RF2500_PROTOID  0x22

/* The Status Message returned with a buffer request */
typedef struct {
    byte     dst_addr;
    byte     src_addr;
    byte     msg_type;
    byte     liaison;
    long     buffer_len;
    byte     prot_id;
    byte     tx_ctl;
    short    dst_port;
    short    src_port;
    byte     trnsc_id;
    byte     trnsc_ctl;
    byte     protoid;
    byte     error;
    byte     resv1;
    byte     resv2;
    byte     retries;
    byte     resv3;
    byte     scsi_status;
    byte     status_type;
    byte     class_code;
    byte     segment;
    byte     scsi_flags;
    byte     infob3;
    byte     infob4;
    byte     infob5;
    byte     infob6;
    byte     exlen;
} RCV_BRQ;

/* The Peripheral Status Message */
typedef struct {
    long    hw_overhead[3];
    byte    protoid;
    byte    error;
    byte    resvl;
    byte    resv2;
    byte    retries;
    byte    resv3;
    byte    scsi_status;
    byte    status_type;
    byte    class_code;
    byte    segment;
    byte    scsi_flags;
    byte    infob3;
    byte    infob4;
    byte    infob5;
    byte    infob6;
    byte    exlen;
    byte    extra[4];
} PSM;

/* Defines for status_type, these tell what the device specific status is. */
#define ST_NU       0x00        /* remaining bytes Not Used */
#define ST_MULTERR  0x01        /* Multiple errors */
#define ST_RQSENS   0x80        /* status from Request Sense command */
#define ST_INITP    0x81        /* Adaptor Initialization Parameters */
#define ST_UNITP    0x82        /* Unit Initialization Parameters */

/* Some interesting defines for the device specific status.. */
/*  For byte 1: */
#define ST_BOT  0x01    /* at Beginning Of Tape */
#define ST_WPT  0x02    /* Write Protected */
#define ST_RWD  0x04    /* tape is Rewinding */
#define ST_ONL  0x08    /* drive is Online */
#define ST_RDY  0x10    /* drive is Ready */
#define ST_SPD  0x20    /* tape speed - 1 is high */
#define ST_DEN  0x40    /* tape denisty - 1 is high */
#define ST_BSY  0x80    /* formatter is Busy */
/*  For byte 2: */
#define ST_FMK  0x02    /* File Mark */
#define ST_EOT  0x04    /* at End Of Tape */
```

```
#define INITTING    0x01    /* Some process is initializing the drive.. */

struct xFMT_INFO {
    byte    cmd;
    byte    fill1;
    word    alt_cyl;
    byte    alt_head;
    byte    alt_sect;
};
#define FMC_MTRACK  0x00    /* Multiple track format */
#define FMC_TRACK   0x40    /* One track format */
#define FMC_MAPTRK  0x80    /* Map a track */
#define FMC_MAPSECT 0x08    /* Map a sector */

/* Possible values for the pcm_count field when formatting/mapping. */
#define SV_DATA_IGNE    0x0 /* Save the data, ignore data errors */
#define SV_DATA_ABT     0x1 /* Save the data, abort on errors */
#define USE_FMT_FILL    0x2 /* Discard the data, use format fill character */

/* =================================================================== */
/* Unordered additions.                                                */
/* =================================================================== */

/* You should not have any other devices on the SCSI bus set to this target
id. */
#define RF2500_ID   0xFF    /* Adaptor command Target ID */
#define OWN_ID      0x00    /* SCSI Target ID for RF3500 -> 0 */
#define DSC_PAR     0x03
#define THROTTLE    8       /* Bus throttle */

/* =================================================================== */
/* Floppy diskette parameters.                                         */
/* =================================================================== */

#define FLP_DEV        0x60
#define FLP_TARGET     0xFE
#define FLP_TYPE       0x16
#define FLP_BLKSIZE     512

/* These two defines should be the same */
#define LOGDSK  8       /* Partitions on physical disk */
#define FLPFMT  8       /* Number of different floppy formats */

#define BYTE0(n)    ((byte)(n) & 0xff)
#define BYTE1(n)    BYTE0((dword)(n)>>8)
#define BYTE2(n)    BYTE0((dword)(n)>>16)
#define BYTE3(n)    BYTE0((dword)(n)>>24)

#define LFLP    3           /* Local floppy (On RF3500) */
```

```
#define DSWAB(x)      ((((x) >> 24) & 0xFF) | (((x) >> 8) & 0xFF00) | (((x) <<
8) & 0x00FF0000L) | (((x) << 24) & 0xFF000000L))
#define SWAB(x) (((((x) >> 8) & 0xff) | (((x) << 8) & 0xff00)))

/* Bits in flags field */
#define SGO     0x01        /* Scatter/gather operation */
#define DAT     0x02        /* Data transmitted in this operation */
#define DIR     0x04        /* Direction of data transfer 1=to the target */
#define IRS     0x08        /* Inhibit request sense */
#define VALID   0x80        /* Valid */

/* Status Block Flags Field bit masks */
#define ST_RTY  0x20        /* Retry required */
#define ST_ERR  0x40        /* Error, check error code */
#define ST_CC   0x80        /* Command complete, last status block */

/* Status Block Class bit masks */
#define ADVALID 0x80        /* Address field information valid */

/* Retry control bit fields */
#define SCINT   0x01        /* Issue "error" interrupt for each retry */
#define RCISB   0x02        /* Issue status block for each retry */
#define RCRPE   0x04        /* Retry parity error */
#define RCRCE   0x08        /* Retry command errors (SCSI errors) */
#define RCRBE   0x10        /* Retry bus errors (selection timeouts, etc) */

/* SCSI commands */
#define SC_READY     0x00   /* Test unit ready */
#define SC_REZERO    0x01   /* Rezero unit */
#define SC_REWIND    0x01   /* Rewind */
#define SC_SENSE     0x03   /* Request sense */
#define SC_FORMAT    0x04   /* Format unit */
#define SC_RDBLKLIM  0x05   /* Read Block Limits - SEQ devices only */
#define SC_REASSIGN  0x07   /* Reassign blocks - Map Sector(s) */
#define SC_READ      0x08   /* Read */
#define SC_WRITE     0x0A   /* Write */
#define SC_SEEK      0x0B   /* Seek */
#define SC_WFM       0x10   /* Write filemark */
#define SC_SPACE     0x11   /* Space blocks, filemarks, EOT */
#define SC_INQUIRY   0x12   /* Inquiry */
#define SC_SELMODE   0x15   /* Mode select */
#define SC_RESERVE   0x16   /* Reserve */
#define SC_RELEASE   0x17   /* Release */
#define SC_ERASE     0x19   /* Erase */
#define SC_SENMODE   0x1A   /* Mode sense */
#define SC_LOAD      0x1B   /* Load/Unload & Start/Stop Device */
#define SC_RDCAP     0x25   /* Read capacity */
#define SC_VERIFY    0x2F   /* Verify the disk surface */
```

```
/* Put these into byte4 of scdb for the SC_SPACE command to tell it what */
/* to search for. */
#define BLOCK   0x0
#define SFM     0x1
#define SQFM    0x2
#define PEOM    0x3

/* Put these into byte4 of scdb for the SC_LOAD command to tell it what */
/* to do.  These are bit masks...to not put them in is to do the opposite. */
#define LOAD    0x1
#define RETEN   0x2


/**************************************************************
 *       SCSI structures
 **************************************************************/

/* MODE SELECT parameter list */
typedef struct  {
    byte    byte0;          /* Reserved */
    byte    medium_type;    /* Medium type */
    byte    byte2;          /* Reserved */
    byte    blk_des_len;    /* Block descriptor length */
    byte    density_code;   /* Density code */
    byte    nblk[3];        /* Number of blocks (MSB) - (LSB) */
    byte    byte8;          /* Reserved */
    byte    blklen[3];      /* Block length (MSB) - (LSB) */
    byte    vend_uniq[50];  /* Vendor Unique parameter bytes */
} mode_sel;

typedef struct {
    byte    page_code;
    byte    page_length;
    word    xfer_rate;  /* Transfer rate */
    byte    nheads;     /* Number of heads */
    byte    spt;        /* Sectors per Track */
    word    nbps;       /* Number of bytes per sector */
    word    ncyls;      /* Number of cylinders */
    word    s_wpre;     /* Starting cylinder - write precomp */
    word    s_rwc;      /* Starting cylinder - reduced write current */
    word    dsr;        /* Drive Step Rate */
    byte    dspw;       /* Drive Step Pulse Width */
    byte    hd_st_dly;  /* Head Settle Delay */
    byte    on_dly;     /* Motor On Delay */
    byte    off_dly;    /* Motor Off Delay */
    byte    trdy;       /* Drive Provides a True Ready Signal */
    byte    hd_ld_dly;  /* Head Load Delay */
    byte    ssn_s0;     /* Starting Sector #, Side Zero */
    byte    ssn_s1;     /* Starting Sector #, Side One */
} page_5;

/* page 20H (Vendor Unique) Floppy Disk Configuration */
typedef struct {
        byte    page_code;
        byte    page_length;
        byte    post_index;     /* Post Index Gap */
        byte    inter_sector;   /* Inter Sector Gap */
        byte    tverify;        /* Seek Verification */
        byte    tsteps;         /* Steps Per Track */
        byte    resv0;          /* Reserved set to 0H */
        byte    resv1;          /* Reserved set to 0H */
} page_20;

/* Reassign blocks defect list */
typedef struct {
    byte    resv0;
    byte    resv1;
    word    dll;            /* Defect List Length */
    dword   lba;            /* Defect Logical Block Address */
} defect_list;

/* READ CAPACITY data list */
typedef struct {
    byte    nblk[4];        /* Logical block address */
    byte    blklen[4];      /* Block length */
} read_cap;

/* INQUIRY Data */
typedef struct {
    byte    dtype;
    byte    rmb_dtq;
    byte    version;
    byte    byte3;
    byte    add_len;
    byte    vend_uniq[41];  /* This could be a MAX of 507 */
} inq_data;

/* Read Block Limits Data */
typedef struct {
    byte    byte0;
    byte    mxblklen[3];
    byte    mnblklen[2];
} blk_lim;
```

```
/**************************************************************
 *       Structures used internally by the driver
 **************************************************************/

typedef struct {
    byte    id;         /* SCSI target ID */
    byte    unit;       /* Target unit number for this device */
} target;

/* This structure holds the size and offset in blocks
 * for each logical disk on a hard disk
 */
typedef struct {
    dword   nblocks;    /* Number of blocks */
    dword   blkoff;     /* Block offset */
} sizes;

/* Device parameters */
typedef struct {
    dword   blklen;     /* Block length */
    dword   nblk;       /* Number of blocks */
} dev_param;

/***************************************************************************
 * Adaptor command parameter structures
 */

typedef struct {
    word            max_scsi_buf_size;
    word            scsi_pad;
    word            active_cmds;
    word            max_qblocks;
    word            min_qblocks;
    word            buf_threshold;
    dword           prealloc_threshold;
    dword           bytes_per_req;
    byte            dutycycle;
    byte            sortlimit;
    byte            retryalg;
    byte            nack_count;
    dword           num_nxt_frag;
    dword           num_mint;
    dword           wrts_prefetched;
    dword           wrts_not_prefetched;
    dword           mb_queue_empty;
    dword           dtr_sorted;
    dword           dtr_combined;
    dword           sort_reached;
    dword           prealloc_buf_in_use;
    word            no_qblocks_for_cmd;
    word            no_buffer;
} set_tuning_tab;

/* Set Board Parameter flag bits */
#define DIS 0x01    /* set   = Report parity errors on SCSI bus */
#define PAR 0x02    /* set   = Allow disconnect/reselect in SCSI
                               operation */

/* Set Unit Parameter flag bits */
#define SYN 0x01    /* enable synchronous transfers to SCSI device */
#define INH 0x02    /* inhibit disconnect */
#define ATN 0x04    /* inhibit attention during select */
#define SRT 0x08    /* enable command sorting */
#define CMB 0x10    /* enable command combining */
#define RAH 0x20    /* enable read ahead */

/* Command codes for ioctl() calls */
#define RFIOC       ('r'<<8)
#define RFIOCGDEBUG (RFIOC|0x01)    /* Get driver debug level */
#define RFIOCSDEBUG (RFIOC|0x02)    /* Set driver debug level */
#define RFIOCSFM    (RFIOC|0x04)    /* Search file mark */
#define RFIOCWFM    (RFIOC|0x05)    /* Write file mark */
#define RFIOCREWIND (RFIOC|0x06)    /* Rewind */
#define RFIOCFMT    (RFIOC|0x07)    /* Format drive */
#define RFIOCGSTAT  (RFIOC|0x08)    /* Get RF3500 statistics */
#define RFIOCIDENT  (RFIOC|0x09)    /* Identify */
#define RFIOCANY    (RFIOC|0x0A)    /* Any SCSI command */
#define RFIOCRDCAP  (RFIOC|0x0B)    /* Read capacity */
#define RFIOCMAP    (RFIOC|0x0C)    /* Map sectors */
#define RFIOCVFY    (RFIOC|0x0D)    /* Verify sectors */
#define RFIOCGPART  (RFIOC|0x0E)    /* Get the partition information */
#define RFIOCSPART  (RFIOC|0x0F)    /* Set the partition information */
#define RFIOCGTARGET(RFIOC|0x10)    /* Get the target id & lun for dev */
#define RFIOCERASE  (RFIOC|0x11)    /* Erase the tape */
#define RFIOCRETEN  (RFIOC|0x12)    /* Retension the tape */
#define RFIOCLOAD   (RFIOC|0x13)    /* Load the media */
#define RFIOCUNLOAD (RFIOC|0x14)    /* Unload the media */

struct dk_mapr {
    dword   dkm_blkno;                  /* The starting block to map */
    dword   dkm_nblk;                   /* How many blocks to map */
};

struct dk_vfy {
    dword   dkv_blkno;              /* The starting block to verify */
    word    dkv_nblk;               /* How many blocks to verify */
    dword   dkv_error;              /* The error code returned by verify */
    dword   dkv_badblock;       /* Where the verify failed. */
};

#define EE_SCSIERR  0x23        /* SCSI returned bad status */
#define ERR_MULTIPLE 0x0E       /* multiple errors */
#define EE_RECSMAL  0x3D        /* actual read smaller than count field */
#define EE_FRMERR   0x80        /* 80H and above are firmware errors */

/* SCSI status (only relevent bits are defined) */
#define CHECK_COND  0x02        /* Check condition */
#define RESV_CONF   0x18        /* Reservation conflict */
#define STATMASK    0x1E        /* Status mask */

#define FM          0x80        /* file mark */
#define ILI         0x20        /* illegal length indicator */
#define EOM         0x40        /* end of media */

#define NOSENSE     0x00        /* No sense? */
#define RECOVERED   0x01        /* Recovered error */
#define UNIT_ATTEN  0x06        /* Unit attention */
#define PROTECTED   0x07        /* Data Protected */
#define BLANK       0x08        /* Blank check */
#define MISCOMPARE  0x0E        /* Data Verify failed */
#define SENSEMASK   0x0F        /* Sense mask */

#define AV      0x80        /* Valid information */

#define SEL_TIMEOUT 10      /* SCSI selection timeout */

/* Defines for operations that this driver can do. */
/* These may apply across device types or be device type specific. */
/* These defines appear in the various *ops[] arrays. */
#define NOOPN_RDCAP 0x000001    /* Do Not issue Read Capacity at open() */
#define NOOPN_MDSEL 0x000002    /* Do Not issue Mode Select at open() call. */
#define ONEFILEMARK 0x000004    /* Tape drive can only write 1 filemark at a
time */
#define GEN_MODE    0x000008    /* Tape drive has "modes" of operations. */
                                /* Add this if you can only do certain   */
                                /* commands (i.e., MODE SELECT) when the */
                                /* drive is in "general" mode        */
                                /* (in contrast to "read" or "write" mode*/
#define NORESERVE   0x000010    /* Don't do Reserve & Release commands */
```

C.2.5 rfdriver.c

```
/*************************************************************/
/*                                                           */
/* IFX device driver for RIMFIRE 2500 disk/diskette adapter */
/*                                                           */
/*************************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "ifxvisi.old"
#include "ifxr2500.h"
#include "rf2500.h"
#define FAR _far
#define FIL1 short int filler1

extern int rfInit();
extern int rFloppyInit();
extern int rfWrite();
extern int rfRead();

Rimfire2500DiskDCB  rfdcb;
BDINFO       board_info;
UNS_MSG      uns_buf;
BRQ_MSG      brq_buf;
dev_param    flpdiskparm;
char         iobuf[MEMSIZE];
char *driver_mbox;

/* Parameter list */
typedef union
{   struct  { unsigned int sector_size;
              unsigned int sectors_per_track;     /* sectors per track */
              unsigned int tracks_per_cylinder;   /* tracks per cylinder */
              unsigned int total_cylinders;       /* total cylinders */
              long total_sectors;
            } u1;
    struct  { long starting_sector;
              char *buffer;
              FIL1;
              long number_of_sectors;
              long *act_count;
            } u2;
    struct  { int opcode;
              int cylinder;
              int track;
            } u3;
    struct  { int opcode;
              IFXGEOMETRY *geometry;
            } u4;
} PL;

/*+ Rimfire2500Driver ===============================================

Description: This is a driver for a floppy diskette controlled by a
             a RIMFIRE 2500 adapter.

Input:       opcode   -
             dcb      - Pointer to the IFXDCB structure.
             pl       - Pointer to a parameter list containing
                        opcode specific parameters.

=================================================================== -*/

int Rimfire2500Driver(opcode, dcb, pl)
int opcode;
IFXDCB *dcb;
PL *pl;
{
  int   err, err2, strt_sect, sect_size, sect_num;
  long  num_bytes, offset;
  IFXGEOMETRY *dc;
  Rimfire2500DiskDCB  *r;
  long actual;

  sc_pend(&driver_mbox, 0L, &err2);
  if (err2)  return err2;

  /* Perform operation according to opcode */
  if (opcode != IFXFINSTALL)
     r = (Rimfire2500DiskDCB *) dcb->dt;
  err = RET_OK;
  switch (opcode)
  {
    case IFXFINSTALL:          /* Install device */
         rfdcb.flp_struc.sector_size = pl->u1.sector_size;
         rfdcb.flp_struc.sectors_per_track = pl->u1.sectors_per_track;
         rfdcb.flp_struc.tracks_per_cylinder = pl->u1.tracks_per_cylinder;
         rfdcb.flp_struc.total_cylinders = pl->u1.total_cylinders;
         rfdcb.flp_struc.total_sectors = pl->u1.total_sectors;
         dcb->device_type = IFXDDISK;
         dcb->dt = (char *)&rfdcb;
         err = RET_OK;
         err = rfInit();
         if (err == RET_OK)
            err = rfFloppyInit();
         if (err != RET_OK)
            err = IFXEIOERR;
         break;
    case IFXFREADS:            /* Read sectors */
    case IFXFWRITES:           /* Write sectors */
         strt_sect = pl->u2.starting_sector;
         sect_size = r->flp_struc.sector_size;
         sect_num  = pl->u2.number_of_sectors;
         if (strt_sect  < 0L  ||  strt_sect > r->flp_struc.total_sectors)
         {  err = IFXEBADPOSN;
            break;
         }
         if ((sect_num <= 0)  ||
             (strt_sect + sect_num > r->flp_struc.total_sectors))
         {  err = IFXEBADXFERCT;
            break;
         }
         offset = (sect_size * strt_sect) / flpdiskparm.blklen;
         num_bytes = sect_size * sect_num;
         if (opcode == IFXFREADS)
            err = rfRead (offset, num_bytes, pl->u2.buffer, &actual);
         else
            err = rfWrite (offset, num_bytes, pl->u2.buffer, &actual);
         *pl->u2.act_count = actual/sect_size;
         break;

       case IFXFREMOVE:           /* Remove device */
            err = RET_OK;
            break;

       case IFXFIOCTL:            /* I/O control operation */
            switch (pl->u3.opcode)
            {
               case IFXOGGEOM:           /* Get disk geometry */
                    dc = pl->u4.geometry;
                    dc->sector_size = r->flp_struc.sector_size;
                    dc->sectors_per_track = r->flp_struc.sectors_per_track;
                    dc->tracks_per_cylinder =
r->flp_struc.tracks_per_cylinder;
                    dc->total_cylinders = r->flp_struc.total_cylinders;
                    dc->total_sectors = r->flp_struc.total_sectors;
                    err = RET_OK;
                    break;

               case IFXOFMTTRK:         /* Format track */
                    if (pl->u3.cylinder == 0  &&  pl->u3.track == 0)
                       err = rfCntl(pl->u3.opcode,pl);
                    break;

               case IFXODISCIN:         /* Discard input buffers */
               case IFXODISCOUT:        /* Discard output buffers */
               case IFXOFLUSHOUT:       /* Flush "dirty" buffers */
```

```
              case IFXFACANCEL:        /* Cancel asynchronous operation */
                   err = RET_OK;
                   break;

              default:           /* Unimplemented control operation */
                   err = IFXENOTIMP;
                   break;
           }
           break;

      default:                   /* Unimplemented function code */
           err = IFXENOTIMP;
           break;
   }
   sc_post(&driver_mbox, (char *)1, &err2);
   if (err2)
      err = err2;
   return err;    /* Return status code to IFX */
}
```

C.2.6 rferror.c

```
/************************************************/
/* rferror(): Reads the RIMFIRE status message, */
/*                and generates the error code. */
/************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "rf2500.h"
extern void mpc_recv_uns();

int rferror()
{
    int     err;
    PSM     status_msg, *psmp;

    psmp = &status_msg;
    mpc_recv_uns (psmp,UNS_STD_LEN);
    err = psmp->error & 0x3F;

    if (err != RET_OK)
    {  printf("Peripheral Status Message:\n");
       printf("ProtoId %x\t Error    %x\n", psmp->protoid, psmp->error);
       printf("Retries %x\t SCSI St  %x\n", psmp->retries, psmp->scsi_status);
       printf("Status  %x\t ClassCd  %x\n", psmp->status_type,
psmp->class_code);
       printf("Segment %x\t SCSI Fl  %x\n", psmp->segment, psmp->scsi_flags);
       printf("infob3  %x\t infob4   %x\n", psmp->infob3, psmp->infob4);
       printf("infob5  %x\t infob6  %x\n", psmp->infob5, psmp->infob6);
       printf("exlen   %x\t extra   %x\n", psmp->exlen, psmp->extra[0]);
    }
    if (err == EE_SCSIERR)
       if ((psmp->scsi_status & STATMASK) == CHECK_COND)
          err = psmp->class_code;
    return err;
}
```

C.2.7 rfbrqerr.c

```
/****************************************************/
/* rfbrqerr(): Checks the status accompanied to the */
/*             accepted buffer request message.     */
/****************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "rf2500.h"

int rfbrqerr(brqp)
RCV_BRQ   *brqp;
{
    int     err;

    err = brqp->error & 0x3F;

    if (err != RET_OK)
    {  printf("Buffer ReQuest Status Message:\n");
       printf("ProtoId %x\t Error    %x\n", brqp->protoid, brqp->error);
       printf("Retries %x\t SCSI St  %x\n", brqp->retries, brqp->scsi_status);
       printf("Status  %x\t ClassCd  %x\n", brqp->status_type,
brqp->class_code);
       printf("Segment %x\t SCSI Fl  %x\n", brqp->segment, brqp->scsi_flags);
       printf("infob3  %x\t infob4   %x\n", brqp->infob3, brqp->infob4);
       printf("infob5  %x\t infob6   %x\n", brqp->infob5, brqp->infob6);
       printf("exlen   %x\n", brqp->exlen);
    }
    if (err == EE_SCSIERR)
       if ((brqp->scsi_status & STATMASK) == CHECK_COND)
          err = brqp->class_code;
    return err;
}
```

C.2.8 *rfinit.c*

```
/**************************************************/
/*                                                */
/* rfinit() - Rimfire 2500 initialization routine */
/*                                                */
/**************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "ifxvisi.old"
#include "rf2500.h"

extern int  ics_read();
extern void ics_write();
extern void mpc_init();
extern void adma_init();
extern int  mpc_send_uns();
extern void rfmkuns();
extern int  rferror();
extern BDINFO   board_info;
extern UNS_MSG  uns_buf;
static int disco = 0;

/* how long (looping) until controller reset should complete */
#define MAXWAIT     (16000)   /* wait few seconds */

int rfInit ()
{
   unsigned int   slot, reg;
   int            tmp, i, rec, timer, err;
   char           hwver[5], fwver[6], fwasm[10];
   unsigned char  fwdate[9];

   slot = SLOT_NO;
   mpc_init();
   adma_init();

   /* Now let's see if it passed BIST. */
   if (ics_read(slot, HD_SUP_LEVEL) <= 0)   /* What support level? */
      return (IFXEDVNTFOUND);

   /* Get the slave status register. */
   timer = MAXWAIT;
   do
   { tmp = ics_read(slot, HD_SLV_STATUS);
     if ((tmp & SS_IN_PROGRESS) != SS_IN_PROGRESS)
        break;
   } while (--timer);   /* wait for BIST to complete */
```

```
if (tmp)
   return (IFXEDVNTREADY);

/* Check the master status. */
tmp = ics_read(slot, HD_MSTR_STATUS);
if ((tmp & 0x3F) != HD_MS_REBOOT)
   return (IFXEDVNTREADY);

/* Check the general status. */
board_info.bd_udf_rec = UD_REC;
rec = board_info.bd_udf_rec;
if (ics_read(slot,rec) != UNIT_DEF_RECORD)
   return (IFXEDVNTREADY);
if (ics_read(slot, rec + UD_GEN_STATUS) != UDGS_POR_COMPLETE)
   return (IFXEDVNTREADY);

/* Then, print out the revision levels. */
/* First, the Hardware Version. */
for (i = 0; i < 4; i++)
    hwver[i] = ics_read(slot, rec + i + 14);
hwver[4] = '\0';
/* Secondly the Firmware Version. */
rec = FIRM_REC;
if (ics_read(slot,rec) != FIRMWARE_RECORD)
   return (IFXEDVNTREADY);
for (i = 0; i < 5; i++)
    fwver[i] = ics_read(slot, rec + i + 9);
fwver[5] = '\0';
/* Read the Firmware Assembly # */
for (i = 0; i < 8; i += 2)
{   tmp = ics_read(slot, rec + (i>>1) + 2);
    fwasm[i] = (((tmp >> 4) & 0xF) + '0');
    fwasm[i+1] = ((tmp & 0xF) + '0');
}
fwasm[8] = '\0';
/* Read the Firmware Date */
for (i = 0; i < 9; i += 3)
{   tmp = ics_read(slot, rec + (i/3) + 6);
    fwdate[i] = (((tmp >> 4) & 0xF) + '0');
    fwdate[i+1] = ((tmp & 0xF) + '0');
}
fwdate[2] = fwdate[5] = '/';
fwdate[8] = '\0';

/* Now, initialize the board. */
reg = board_info.bd_udf_rec + UD_CTLR_INIT;
tmp = ics_read(slot, reg);
board_info.bd_hostid_rec = HID_REC;
rec = board_info.bd_hostid_rec;
```

```
    if (ics_read(slot,rec) != HOST_ID_RECORD)
       return (IFXEDVNTREADY);


    if (tmp == 0)        /* ID´s are Initialized... */
    {
        reg = board_info.bd_udf_rec + UD_PORTID;
        board_info.port_id = ics_read(slot, reg+1) << 8;
        board_info.port_id |= ics_read(slot, reg);
        reg = board_info.bd_hostid_rec + HID_HOSTID;
        board_info.host_id = ics_read(slot, reg+1) << 8;
        board_info.host_id |= ics_read(slot, reg);
    }
    else
    {   reg = board_info.bd_hostid_rec + HID_HOSTID;
        ics_write(slot, reg, RF2500_HOST_ID & 0xFF);
        ics_write(slot, reg+1, (RF2500_HOST_ID >> 8) & 0xFF);
        board_info.host_id = RF2500_HOST_ID;
        reg = board_info.bd_udf_rec + UD_PORTID;
        ics_write(slot, reg, RF2500_PORT_ID & 0xFF);
        ics_write(slot, reg+1, (RF2500_PORT_ID >> 8) & 0xFF);
        board_info.port_id = RF2500_PORT_ID;
    }


    /* initialize the adaptor itself */
    bzero((char *)&uns_buf, sizeof(UNS_MSG));
    uns_buf.cmd_buf.adpcm.targetid = RF2500_ID;
    uns_buf.cmd_buf.adpcm.tbd[1] = OWN_ID;
    uns_buf.cmd_buf.adpcm.tbd[2] = DSC_PAR;
    uns_buf.cmd_buf.adpcm.cmd = AD_INITP;
    rfmkuns (UNS_TYPE);
    err = mpc_send_uns(&uns_buf,UNS_STD_LEN);
    if (err == RET_OK)
       err = rferror();
    if (err != RET_OK)
       return (err);


    /* Check the general status register. */
    tmp = ics_read(slot, board_info.bd_udf_rec + UD_GEN_STATUS);
    if (tmp != UDGS_INIT_COMPLETE)
       return (IFXEDVNTREADY);


    timer = 1000000L;
    while (timer--);  /* Give the devices some time to reset themselves. */
    return (RET_OK);
}
```

C.2.9 *rflpinit.c*

```
/****************************************************/
/*                                                  */
/* rfFloppyInit(): Initializes the floppy device.  */
/*                                                  */
/****************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "ifxvisi.old"
#include "rf2500.h"

extern UNS_MSG  uns_buf;
extern BRQ_MSG  brq_buf;
extern dev_param  flpdiskparm;
extern char iobuf[MEMSIZE];
extern void rfmkuns();
extern int  rferror();
extern int  rfbrqerr();
extern int mpc_send_uns();
extern int send_data();
extern int recv_data();

int rfFloppyInit ()
{
    register char *bp;
    register int  datalen, bsize, nblk, err;
    register int write_prot;
    mode_sel    *ms;
    page_5      *p5;
    page_20     *p20;
    read_cap    *rc;
    inq_data    *inqdat;

    bp = &iobuf[0];
    write_prot = 0;     /* Assume not write protected. */

    /* Issue a 'test unit ready' cmd to check if the eevib0 is present */
    bzero((char *)&uns_buf, sizeof(UNS_MSG));
    uns_buf.cmd_buf.scpcm.scb[0] = SC_READY;
    uns_buf.cmd_buf.scpcm.targetid   = FLP_TARGET;
    uns_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
    rfmkuns (UNS_TYPE);
    err = mpc_send_uns(&uns_buf,UNS_STD_LEN);
    if (err == 0)
       err = rferror();
    if (err != RET_OK)
       return (err);
```

```
/* Issue an 'Inquiry' command. */
bzero((char *)&uns_buf, sizeof(UNS_MSG));
datalen = sizeof (inq_data);
uns_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
uns_buf.cmd_buf.scpcm.scb[0] = SC_INQUIRY;
uns_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
uns_buf.cmd_buf.scpcm.scb[4] = datalen;
inqdat = (inq_data *) bp;
bzero((char *) inqdat, datalen);
rfmkuns (UNS_TYPE);
err = recv_data(&uns_buf,UNS_STD_LEN,bp,datalen);
if (err == 0)
   err = rfbrqerr(&uns_buf);
if (err != RET_OK)
   return (err);

/* Issue 'mode select' command */
bzero((char *)&brq_buf, sizeof(BRQ_MSG));
brq_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
brq_buf.cmd_buf.scpcm.scb[0] = SC_SELMODE;
brq_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
brq_buf.cmd_buf.scpcm.scb[4] = 44;
ms = (mode_sel *)bp;
bzero((char *)ms, sizeof(mode_sel));
ms->medium_type = FLP_TYPE;
ms->blk_des_len = 8;
ms->density_code = 0;
ms->nblk[0] = 0;     /* zero = all the blocks */
ms->nblk[1] = 0;
ms->nblk[2] = 0;
ms->blklen[0] = BYTE2(FLP_BLKSIZE);
ms->blklen[1] = BYTE1(FLP_BLKSIZE);
ms->blklen[2] = BYTE0(FLP_BLKSIZE);

/* Page 5H - floppy disk configuration */
p5 = (page_5 *)ms->vend_uniq;
bzero((char *)p5, sizeof(page_5));
p5->page_code = 5;
p5->page_length = 22;  /* Don't count length or code. */
p5->dsr = SWAB(3000);
p5->hd_st_dly = 1;
p5->on_dly = 10;
p5->off_dly = 40;
p5->hd_ld_dly = 1;
p5->xfer_rate = SWAB(0x1F4);  /* 500 Kbit/s */
p5->nheads = 2;
p5->nbps = SWAB(FLP_BLKSIZE);
```

```
p5->spt = 15;   /* AT compatible. */
p5->ncyls = SWAB(80);
p5->s_wpre = SWAB(255);
p5->s_rwc = SWAB(255);
p5->ssn_s0 = 1;
p5->ssn_s1 = 1;

/* Page 20H - Vendor unique */
/* Start page 20 at the end of the page 5. */
p20 = (page_20 *)&p5[1];
bzero((char *)p20, sizeof(page_20));
p20->page_code = 0x20; /* Fixed Value */
p20->page_length = 6; /* Fixed Value */
p20->post_index = 0; /* If default is 0, length is 32 bytes
            (single density) or 62 bytes (double density) */
p20->inter_sector = 0; /* If default is 0, length is 33 bytes
        (single density) or 63 bytes (double density) */
p20->tverify = 0; /* Setting 0 to 1 causes adapter to verify seeks */
p20->tsteps = 1;  /* Setting 1 causes adapter to read 96 TPI drive.
                     The tsteps value mutiplied by the number of cylinders

                     (set in page_5) must be less than 254 */
rfmkuns (BRQ_TYPE);
err = send_data(&brq_buf,UNS_STD_LEN,bp,44);
if (err == 0)
   err = rferror();
if (err != RET_OK)
   return (err);

/* First, let's get the mode sense information. */
bzero((char *)&uns_buf, sizeof(UNS_MSG));
datalen = sizeof(mode_sel);
ms = (mode_sel *)bp;
uns_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
uns_buf.cmd_buf.scpcm.scb[0] = SC_SENMODE;
uns_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
uns_buf.cmd_buf.scpcm.scb[4] = datalen;
bzero((char *)ms, datalen);
rfmkuns (UNS_TYPE);
err = recv_data(&uns_buf,UNS_STD_LEN,bp,datalen);
if (err == 0)
   err = rfbrqerr(&uns_buf);
if (err != RET_OK)
   return (err);

write_prot = (ms->byte2 & 0x80);
/* Get the block size in case there is one */
bsize = 0;
```

```
    bsize = (ms->blklen[0] << 16)  |   (ms->blklen[1] <<  8)(ms->blklen[2]);
    nblk = 0;
    nblk = (ms->nblk[0] << 16) |  (ms->nblk[1] << 8) | (ms->nblk[2]);
    if (bsize == 0)
       return IFXEBADBFSIZE;
    flpdiskparm.nblk = nblk;
    flpdiskparm.blklen = bsize;

    /* Issue 'read capacity' command */
    bzero((char *)&uns_buf, sizeof(UNS_MSG));
    datalen = sizeof (read_cap);
    uns_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
    uns_buf.cmd_buf.scpcm.scb[0] = SC_RDCAP;
    uns_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
    rc = (read_cap *)bp;
    bzero((char *)rc, datalen);
    rfmkuns (UNS_TYPE);
    err = recv_data(&uns_buf,UNS_STD_LEN,bp,datalen);
    if (err == 0)
       err = rfbrqerr(&uns_buf);
    if (err != RET_OK)
       return (err);

    flpdiskparm.nblk = (rc->nblk[0] << 24) |
                       (rc->nblk[1] << 16) |
                       (rc->nblk[2] <<  8) |
                       (rc->nblk[3]);
    flpdiskparm.nblk++;
    flpdiskparm.blklen =
        (rc->blklen[0] << 24) | (rc->blklen[1] << 16)
             | (rc->blklen[2] <<  8) | (rc->blklen[3]);
    if (bsize == 0)
       return IFXEBADBFSIZE;

    /* NOW...test for floppy existence by doing a rezero. */
    bzero((char *)&uns_buf, sizeof(UNS_MSG));
    uns_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
    uns_buf.cmd_buf.scpcm.scb[0] = SC_REZERO;
    uns_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
    rfmkuns (UNS_TYPE);
    err = mpc_send_uns(&uns_buf,UNS_STD_LEN);
    if (err == 0)
       err = rferror();
    if (err != RET_OK)
       return (err);

    /* Open is successful */
    return (RET_OK);
}
```

C.2.10 *rfread.c*

```
/***************************************************/
/* rfread(): Reads sectors from the floppy device. */
/***************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "ifxvisi.old"
#include "rf2500.h"
extern void rfmkuns();
extern int  rferror();
extern int recv_data();
extern UNS_MSG  uns_buf;
extern dev_param   flpdiskparm;

int rfRead(strt_sect,count_bytes,iobuf,actual)
int  strt_sect, count_bytes;
char *iobuf;
int  *actual;
{
    dword   blklen;
    int     err;

    *actual = 0;
    blklen = flpdiskparm.blklen;
    bzero((char *)&uns_buf, sizeof(UNS_MSG));
    uns_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
    uns_buf.cmd_buf.scpcm.scb[0] = SC_READ;
    uns_buf.cmd_buf.scpcm.scb[1] = (FLP_DEV<<5 | (BYTE2(strt_sect) & 0x1F));
    uns_buf.cmd_buf.scpcm.scb[2] = BYTE1(strt_sect);
    uns_buf.cmd_buf.scpcm.scb[3] = BYTE0(strt_sect);
    uns_buf.cmd_buf.scpcm.scb[4] = count_bytes / blklen;
    rfmkuns (UNS_TYPE);
    err = recv_data(&uns_buf,UNS_STD_LEN,iobuf,count_bytes);
    if (err == RET_OK)
       err = rfbrqerr(&uns_buf);
    if (err == RET_OK)
       *actual = count_bytes;
       else  err = IFXEIOERR;
    return (err);
}
```

C.2.11 rfwrite.c

```
/***************************************************/
/* rfwrite(): Writes sectors to the floppy device. */
/***************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "ifxvisi.old"
#include "rf2500.h"
extern void rfmkuns();
extern int  rferror();
extern int send_data();
extern BRQ_MSG  brq_buf;
extern dev_param   flpdiskparm;

int rfWrite(strt_sect,count_bytes,iobuf,actual)
int  strt_sect, count_bytes;
char *iobuf;
int  *actual;
{
    dword    blklen;
    int      err;

    *actual = 0;
    blklen = flpdiskparm.blklen;
    bzero((char *)&brq_buf, sizeof(BRQ_MSG));
    brq_buf.cmd_buf.scpcm.targetid = FLP_TARGET;
    brq_buf.cmd_buf.scpcm.scb[0] = SC_WRITE;
    brq_buf.cmd_buf.scpcm.scb[1] = (FLP_DEV<<5 | (BYTE2(strt_sect) & 0x1F));
    brq_buf.cmd_buf.scpcm.scb[2] = BYTE1(strt_sect);
    brq_buf.cmd_buf.scpcm.scb[3] = BYTE0(strt_sect);
    brq_buf.cmd_buf.scpcm.scb[4] = count_bytes / blklen;
    rfmkuns (BRQ_TYPE);
    err = send_data(&brq_buf,UNS_STD_LEN,iobuf,count_bytes);
    if (err == RET_OK)
       err = rferror();
    if (err == RET_OK)
       *actual = count_bytes;
       else  err = IFXEIOERR;
    return (err);
}
```

C.2.12 *rfcntl.c*

```
/****************************************************************/
/* rfcntl(): Several control operations on the floppy device. */
/****************************************************************/

#include "compiler.h"
#include "vrtxvisi.h"
#include "ifxvisi.old"
#include "rf2500.h"
extern void rfmkuns();
extern int  rferror();
extern int  rfbrqerr();
extern int mpc_send_uns();
extern UNS_MSG  uns_buf;

int rfCntl(opcode,pl)
int opcode;
char *pl;
{
    int     err;
    switch (opcode)
    {
            case IFXOFMTTRK:          /* Format track */
                 bzero((char *)&uns_buf, sizeof(UNS_MSG));
                 uns_buf.cmd_buf.scpcm.scb[0] = SC_FORMAT;
                 uns_buf.cmd_buf.scpcm.targetid    = FLP_TARGET;
                 uns_buf.cmd_buf.scpcm.scb[1] = FLP_DEV<<5;
                 rfmkuns (UNS_TYPE);
                 err = mpc_send_uns(&uns_buf,UNS_STD_LEN);
                 if (err == RET_OK)
                    err = rferror();
                 if (err != RET_OK)
                    err = IFXEIOERR;
                 break;

            default:
                 err = IFXENOTIMP;
                 break;
    }
    return err;    /* Return status code to IFX */
}
```

C.2.13 *rfmkuns.c*

```
/*****************************************************/
/* rfmkuns(type):                                    */
/* Construct one of the 2 unsolicited message types, */
/* according to the 'type' parameter.                */
/*                                                   */
/*****************************************************/
#include "rf2500.h"
extern UNS_MSG  uns_buf;
extern BRQ_MSG  brq_buf;
extern BDINFO   board_info;
int    trnsc_no = 0;


void rfmkuns (type)
char type;
{
    trnsc_no += 1;
    if (trnsc_no == 256)
       trnsc_no = 1;
    if (type == 0x00)
    {  uns_buf.dst_addr   = RF2500_HOST_ID;
       uns_buf.src_addr   = CPU_HOST_ID;
       uns_buf.msg_type   = type;
       uns_buf.prot_id    = 0x02;
       uns_buf.tx_ctl     = 0;
       uns_buf.dst_port   = (board_info.host_id << 8) + board_info.port_id;
       uns_buf.src_port   = CPU_PORT_ID;
       uns_buf.trnsc_id   = trnsc_no;
       uns_buf.trnsc_ctl  = 0x01;
       uns_buf.protoid    = RF2500_PROTOID;
    }
    else
    {  brq_buf.dst_addr   = RF2500_HOST_ID;
       brq_buf.src_addr   = CPU_HOST_ID;
       brq_buf.msg_type   = type;
       brq_buf.liaison    = 0;
       brq_buf.prot_id    = 0x02;
       brq_buf.tx_ctl     = 0;
       brq_buf.dst_port   = (board_info.host_id << 8) + board_info.port_id;
       brq_buf.src_port   = CPU_PORT_ID;
       brq_buf.trnsc_id   = trnsc_no;
       brq_buf.trnsc_ctl  = 0x01;
       brq_buf.protoid    = RF2500_PROTOID;
    }
}
```

C.3 MBII Level Services

This section contains MBII low-level service routines. In order to provide simple services that can be tested easily, these services work in a polling mode.

C.3.1 *mpc.inc*

```
;---------------------------------------------------------------------------
;
; contents: mpc addresses, commands and specific message structures
;
;---------------------------------------------------------------------------

; MPC ports offsets:
MPC_MDATA    EQU 10h
MPC_MCMD     EQU 1Ch
MPC_MSTAT    EQU 00h
MPC_MRST     EQU 00h
MPC_MCTL     EQU 0Ch
MPC_MERR     EQU 14h
MPC_MSOCMP   EQU 20h
MPC_MSICMP   EQU 24h
MPC_MSOCAN   EQU 20h
MPC_MSICAN   EQU 24h
MPC_MCON     EQU 08h
MPC_MID      EQU 04h
MPC_ICADRL   EQU 30h
MPC_ICADRH   EQU 34h
MPC_ICDATA   EQU 3Ch

; identity of the participating agents
CPU_HOST_ID      EQU   01h
RF2500_HOST_ID   EQU   05h

; mpc commands
MPC_RESET            EQU 00h ; MPC RESET VALUE
MPC_TX_START         EQU 00h ; MPC Start command
MPC_RET_OK           EQU 00h ;
MPC_ERROR_RETRIEVER  EQU 00h ;
MPC_RET_ERROR        EQU 01h ;
MPC_RECV_DATA        EQU 02h ; test value

; bit masks for the message status register MPC_MSTAT
MPC_INITDONE     EQU 80h
MPC_SICMP    EQU 10h
MPC_SOCMP    EQU 08h
MPC_XMTERR   EQU 04h
MPC_RCVNE    EQU 02h
MPC_XMTNF    EQU 01h
```

```
; bit definitions for the message control register MPC_MCTL
MPC_XMTIE   EQU 01h
MPC_RCVIE   EQU 02h
MPC_ERRIE   EQU 04h
MPC_SOCIE   EQU 08h
MPC_SICIE   EQU 10h

; message types definitions
MPC_BROADCAST    EQU    01h
MPC_UNSOLICITED  EQU    00h
MPC_BUFF_REQUEST EQU    24h
MPC_BUFF_GRANT   EQU    35h

; Unsolicited message structure
unsol_msg  STRUC
   dst_id       DB  ?
   src_id       DB  ?
   msg_type     DB  ?
   request_id   DB  ?
   data         DB  24  DUP  (?)
   length_0_u   DB  ?
   length_1_u   DB  ?
   length_2_u   DB  ?
   res_data_u   DB  ?
unsol_msg  ENDS

; Sent Buffer Request message structure
buff_request_msg  STRUC
   dst_id_r     DB  ?
   src_id_r     DB  ?
   msg_type_r   DB  ?
   request_id_r DB  ?
   res_data_r   DB  ?
   length_0_r   DB  ?
   length_1_r   DB  ?
   length_2_r   DB  ?
   data_r   DB  24  DUP  (?)
buff_request_msg   ENDS

; Accepted Buffer Request message structure
buff_request_sts  STRUC
   dst_id_rs       DB  ?
   src_id_rs       DB  ?
   msg_type_rs     DB  ?
   request_id_rs   DB  ?
   res_data_rs     DB  ?
   length_0_rs     DB  ?
   length_1_rs     DB  ?
   length_2_rs     DB  ?
   prot_id_rs       DB  ?
   tx_ctl_rs        DB  ?
   dst_port_rs      DW  ?
   src_port_rs      DW  ?
   trnsc_id_rs      DB  ?
   trnsc_ctl_rs     DB  ?
   protoid_rs       DB  ?
   error_rs         DB  ?
   resv1_rs         DB  ?
   resv2_rs         DB  ?
   retries_rs       DB  ?
   resv3_rs         DB  ?
   scsi_status_rs   DB  ?
   status_type_rs   DB  ?
   class_code_rs    DB  ?
   segment_rs       DB  ?
   scsi_flags_rs    DB  ?
   infob3_rs        DB  ?
   infob4_rs        DB  ?
   infob5_rs        DB  ?
   infob6_rs        DB  ?
   exlen_rs         DB  ?
buff_request_sts    ENDS

; Buffer Grant message structure
buff_grant_msg    STRUC
   dst_id_g DB  ?
   src_id_g DB  ?
   msg_type_g   DB  ?
   request_id_g DB  ?
   not_used_g   DB  ?
   liaison_id_g DB  ?
   duty_cycle_g DB  ?
   length_0_g   DB  ?
buff_grant_msg ENDS

; definitions of mpc_tx_status codes
MPC_TX_FREE EQU 00h; no transmission was active
MPC_TX_UNS  EQU 01h; unsolicited message has been transmitted
```

C.3.2 adma.inc

```
;---------------------------------------------------------------
;
; File:   adma.inc
;
; 82258 - Advanced Direct Memory Access Coprocessor
;
; Related Board iSBC 386/116
;
; The channel assignement is the following:
; Channel 2 -- solicited message input (MPC-IDREQ)
; Channel 3 -- solicited message output (MPC-ODREQ)
;
; Symbols for the adma_handler
;
;------------------------------------------------------------------

ADMA_IN_CHANNEL   EQU  2
ADMA_OUT_CHANNEL  EQU  3

; General Registers:
ADMA_GCR  equ  0200h; General Command Register
ADMA_GSR  equ  0204h; General Status Register
ADMA_GMR  equ  0208h; General Mode Register
ADMA_GBR  equ  020Ah; General Burst Register
ADMA_GDR  equ  020Ch; General Delay Register

; Channel Registers:
ADMA_IN_CPRL   equ  02A0h; Command Pointer Register Low for channel 2
ADMA_IN_CPRH   equ  02A2h; Command Pointer Register High for channel 2
ADMA_OUT_CPRL  equ  02E0h; Command Pointer Register Low for channel 3
ADMA_OUT_CPRH  equ  02E2h; Command Pointer Register High for channel 3
ADMA_IN_CSR    equ  0290h;
ADMA_OUT_CSR   equ  02D0h;

; Command Block Structure
adma_cmd_block  struc
    cmd         dw  ? ; type 1 command
    src_addr    dd  ?
    dest_addr   dd  ?
    byte_count  dd  ?
    status      dw  ?
    stop        dw  ? ; type 2 command
    stop_data_1 dw  ?
    stop_data_2 dw  ?
adma_cmd_block  ends
```

C.3.3 mb2lib.asm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This module contains basic multibus II operations.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
.xlist
INCLUDE mpc.inc
INCLUDE vrtxvisi.inc
INCLUDE adma.inc
INCLUDE descrp.inc
.list

dseg    segment para use32 public 'data'
        assume ds:dseg

adma_in_cmd   adma_cmd_block  <>
adma_out_cmd  adma_cmd_block  <>
grant         buff_grant_msg  <>
endoftrns     DD    ?

dseg    ends

cseg  segment para use32 public 'code'
      assume cs:cseg

      public  ics_read, ics_write
      public  mpc_init, mpc_send_uns, mpc_recv_uns
      public  adma_init, adma_handler
      public  send_data, recv_data
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  val = ics_read (slot, reg);
; description:  procedure which reads a target register through the
;               interconnect space.
; input:        slot = The target slot number.
;               reg  = The target register number.
; output:       The required register content in register eax.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
ics_read  proc  near

      push  ebp
      mov   ebp,esp
      push  ebx
      mov   eax,[ebp+08h]
      shl   eax,11
```

```
        mov   ebx,[ebp+0Ch]
        shl   ebx,2
        add   eax,ebx
        pushfd
        cli
        out   030h,al
        mov   al,ah
        out   034h,al
        sub   eax,eax
        in    al,03Ch
        popfd
        pop   ebx
        pop   ebp
        ret

ics_read  endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  ics_write (slot, reg, val);
; description:  procedure which writes a value into a target register
;               through the interconnect space.
; input:        slot = The target slot number.
;               reg  = The target register number.
;               val  = The value to be written.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
ics_write  proc  near

        push  ebp
        mov   ebp,esp
        push  ebx
        mov   eax,[ebp+08h]
        shl   eax,11
        mov   ebx,[ebp+0Ch]
        shl   ebx,2
        add   eax,ebx
        pushfd
        cli
        out   030h,al
        mov   al,ah
        out   034h,al
        mov   eax,[ebp+014h]
        out   03Ch,al
        popfd
        pop   ebx
        pop   ebp
        ret
```

```
ics_write  endp
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  mpc_init ();
; description:  procedure which initializes MPC registers.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
mpc_init  proc  near

; reset MPC
      mov  al,MPC_RESET
      out  MPC_MRST,al

; poll until init done
mpc_init1:
      in   al,MPC_MSTAT
      and  al,MPC_INITDONE
      jnz  SHORT mpc_init1

; Configure MPC to: full message support:
;                   32 bit CPU,   16 bit DMA
      mov  al,89h
      out  MPC_MCON,al

; set message id for this host

      mov  al,CPU_HOST_ID
      out  MPC_MID,al

; wait for INITDONE to become 1
mpc_init2:
      in   al,MPC_MSTAT
      and  al,MPC_INITDONE
      jz   SHORT mpc_init2

      ret

mpc_init endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  adma_init ();
; description:  procedure which initializes the DMA coprocessor registers.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;;;;;;;;;;;;;;;
;
adma_init proc near

; program GMR with:  16 bit,  local mode,   normal channel 3,
;                    two cycle transfer,  all rotating priority
;                    interrupt disabled
```

```
      push  ax
      push  dx
      mov   ax,7F03h
      mov   dx,ADMA_GMR
      out   dx,ax

; program burst
      mov   ax,00h
      mov   dx,ADMA_GBR
      out   dx,ax

; program delay
      mov   ax,00h
      mov   dx,ADMA_GDR
      out   dx,ax
      pop   dx
      pop   ax
      ret

adma_init  endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  err = mpc_send_uns (msg,len);
; description:  procedure which sends an unsolicited message.
; input:        message pointer (48 bits)
;               message length
; output:       eax:  OK = RET_OK,  error != RET_OK;
;               If error, the message is retrieved from message error port
;               and written over the original message. The upper 4 bits of
;               byte 3 contain the error information.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
mpc_send_uns    proc  near

      push  ebp
      mov   ebp,esp
      push  es
      push  esi
      push  ecx

; poll for TX FIFO Empty
send_uns1:
      in   al,MPC_MSTAT
      and  al,MPC_XMTNF
      jz   SHORT send_uns1

; load the message to MDATA port
      mov  ecx,[ebp+010h]
      shr  ecx,2          ; len/4 moves
      push ecx
      cld                  ; clear direction flag
      les  esi,[ebp+8h]    ; the message address from the stack
send_uns2:
      lods dword ptr es:[esi]  ; eax <-- next 4 bytes of the message
      out  MPC_MDATA,eax
      loop send_uns2
      pop  ecx

; start transmission
      mov  al,MPC_TX_START
      out  MPC_MCMD,al

; poll for transmission status
send_uns3:
      in   al,MPC_MSTAT
      mov  ah,al
      and  ah,02h
      jnz  send_read
      and  al,MPC_XMTNF OR  MPC_XMTERR
      jz   SHORT send_uns3
      and  al,MPC_XMTNF
      jz   SHORT send_uns_error

; Success Exit
      pop  ecx
      pop  esi
      pop  es
      pop  ebp
      mov  al,MPC_RET_OK
      ret

send_read:
      mov   eax,[ebp+010h]
      push  eax
      les   eax,[ebp+08h]
      push  es
      push  eax
      call  mpc_recv_uns
      add   esp,0Ch

send_uns_error:
      push  edi
; retrieve the failed message from the error port
; first, dummy read to perpare the failed message
      in   al,MPC_MERR

; ecx = length in dwords from the receive loop
      les  edi,[ebp+8h]    ; the message address from the stack
```

```
send_uns4:
      in    eax,MPC_MERR    ; read 4 bytes of the message
      stos  dword ptr es:[edi]  ; original message <-- eax
      loop  send_uns4

; inform the MPC that the software has retrieved the messge in error
      mov  al,MPC_ERROR_RETRIEVED
      out  MPC_MERR,al

; exit
      pop  edi
      pop  ecx
      pop  esi
      pop  es
      pop  ebp
      mov  al,MPC_RET_ERROR
      ret

mpc_send_uns endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  mpc_recv_uns (msg,len);
; description:  procedure which reads an unsolicited message.
; input:        buffer pointer (48 bits)
;               message length
; output:       message, in the buffer provided by the caller.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
mpc_recv_uns    proc  near

      push  ebp
      mov   ebp,esp
      push  es
      push  edi
      push  ecx
      push  eax

; poll for Receive FIFO Not Empty
recv_uns1:
      in   al,MPC_MSTAT
      and  al,MPC_RCVNE
      jz   SHORT recv_uns1

; retrieve the received message from the data port
; first, read the length of the received message
      in   al,MPC_MDATA

; make a read loop into the message buffer space
      mov  ecx,[ebp+010h]
      shr  ecx,2                ; len/4 moves
      push ecx
      les  edi,[ebp+8h]         ; the message address from the stack
recv_uns2:
      in    eax,MPC_MDATA       ; read 4 bytes of the message
      stos  dword ptr es:[edi]  ; message buffer <-- eax
      loop  recv_uns2
      pop   ecx

; inform the MPC that the software has retrieved the received message
      in   al,MPC_MCMD

; exit
      pop  eax
      pop  ecx
      pop  edi
      pop  es
      pop  ebp
      ret

mpc_recv_uns endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  int = realaddr (addr)
; description:  procedure to compute the physical address from logical
;               selector:offset address. Logical addresses are supposed
;               GDT based.
; input:        addr - the address to be cinverted.
; output:       EAX the physical address
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
realaddr  proc near

      push    ebp
      mov     ebp,esp
      push    es
      push    edx
      les     eax,[ebp+08h]
      push    eax
      mov     eax,0
      mov     ax,es
      mov     dx,GDT_ALIAS_DES    ; set es:0 to point to GDT
      mov     es,dx

; get the base address from the descriptor indexed by dx
      mov     dx,ax
      shr     eax,3               ; get rid of unsignificant bits
      shl     eax,3
      mov     dh,es:[eax].bas_24_31
      mov     dl,es:[eax].bas_16_23
      shl     edx,16
      mov     dx,es:[eax].bas_0_15

; base address is in edx, add the given offset
      pop     eax                  ; restore offset
      add     eax,edx              ; address in eax
      pop     edx
      pop     es
      pop     ebp
      ret

realaddr  endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call:         adma_setup_output
; description:  procedure to setup ADMA output channel.
; input:        EAX = the physical address of a source buffer.
;               EDX = byte count.
; output:       none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
adma_setup_output  proc near

; save input twice for different pops
      push  eax
      push  edx
      push  eax
      push  edx

; stop any channel 3 operation
      mov  al,8Ch
      mov  dx,ADMA_GCR
      out  dx,al

; restore the buffer address and count
      pop  edx
      pop  eax

; the ADMA channel is set up with the following command:
;          Destination (MPC) Sync.
;          Source memory
;          Source increment
;          Source width 16 bits
;          Destination I/O
;          Destination noinc
;          Destination width 16 bits
```

```
      mov  adma_out_cmd.cmd,808Dh
      mov  adma_out_cmd.src_addr,eax     ; source address of output buffer
      mov  adma_out_cmd.dest_addr,0
      mov  adma_out_cmd.byte_count,edx
      mov  adma_out_cmd.status,0
      mov  adma_out_cmd.stop,0
      mov  adma_out_cmd.stop_data_1,0
      mov  adma_out_cmd.stop_data_2,0

; load command block pointer in ADMA
; first, get physical address of command block
      push ds
      mov  eax,offset adma_out_cmd
      push eax
      call realaddr
      add  esp,08h

; physical address in now in eax, load pointer here
      mov  dx,ADMA_OUT_CPRL
      out  dx,ax
      mov  dx,ADMA_OUT_CPRH
      shr  eax,16
      out  dx,ax

; start the adma channel 3 operation
      mov  al,8Ah
      mov  dx,ADMA_GCR
      out  dx,al
      pop  edx
      pop  eax
      ret

adma_setup_output  endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call:          adma_setup_input
; description:   procedure to setup ADMA input channel.
; input:         EAX = the physical address of the destination buffer.
;                EDX = byte count.
; output:        none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
adma_setup_input proc near

; save input twice for different pops
      push  eax
      push  edx
      push  eax
      push  edx
```

```
; stop any channel 2 operation
        mov  al,4Ch
        mov  dx,ADMA_GCR
        out  dx,al


; prepare the command block
        pop  edx
        pop  eax
; the ADMA channel is set up with the following command:
;               Source (MPC) Sync.
;               Destination memory
;               Destination increment
;               Destination width 16 bits
;               Source I/O
;               Source noinc
;               Source width 16 bits
        mov  adma_in_cmd.cmd,40D8h
        mov  adma_in_cmd.src_addr,0
        mov  adma_in_cmd.dest_addr,eax    ; dest. address of input buffer
        mov  adma_in_cmd.byte_count,edx
        mov  adma_in_cmd.status,0
        mov  adma_in_cmd.stop,0
        mov  adma_in_cmd.stop_data_1,0
        mov  adma_in_cmd.stop_data_2,0

; load command block pointer in ADMA
; first, get physical address of command block
        push ds
        mov  eax,offset adma_in_cmd
        push eax
        call realaddr
        add  esp,08h

; physical address in now in eax, load pointer here
        mov  dx,ADMA_IN_CPRL
        out  dx,ax
        mov  dx,ADMA_IN_CPRH
        shr  eax,16
        out  dx,ax

; start the adma channel 2 operation
        mov  al,4Ah
        mov  dx,ADMA_GCR
        out  dx,al
        pop  edx
        pop  eax
        ret
```

```
adma_setup_input  endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call:          adma_handler
; description:   set up the ADMA channel for the operation.
; input:         DX:EAX - data address
;                ECX    - data count
;                EBX    - data channel
; output:        none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
adma_handler   proc   near

      push  edx
      push  eax

; first, get the physical address of the output buffer
      call  realaddr

; prepare the byte count for the set up routine which expects it in EDX
      mov   edx,ecx

; find on which channel to set up
      cmp   ebx,ADMA_IN_CHANNEL
      je    SHORT set_in

set_out:
      call  adma_setup_output
      pop   edx
      pop   eax
      ret

set_in:
      call  adma_setup_input
      pop   edx
      pop   eax
      ret

adma_handler  endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:   err = send_data (uns,len,msg,length);
; description:   procedure which sends an unsolicited message + transfering
;                a large data buffer through the DMA.
; input:         unsolicited buffer prepared by the caller (48 bits)
;                the unsilicited message length
;                message pointer (48 bits)
;                message length
; output:        eax:  OK = RET_OK,  error != RET_OK;
;                If error, the message is retrieved from message error port
;                and written over the original message. The upper 4 bits of
;                byte 3 contain the error information.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
    public  send_data
send_data  proc   near

        push  ebp
        mov   ebp,esp
        push  es
        push  ebx
        push  ecx
        push  edx

        les   eax,[ebp+014h]
        mov   dx,es
        mov   ecx,[ebp+01Ch]   ;data count in bytes
        mov   ebx,ADMA_OUT_CHANNEL
        call  adma_handler


; get the data length and put it in message
        les   eax,[ebp+08h]
        mov   es:[eax].length_0_r,cl
        mov   es:[eax].length_1_r,ch
        shr   ecx,8
        mov   es:[eax].length_2_r,ch

; send the message, push its address
        mov   ecx,[ebp+010h]
        push  ecx
        push  es
        push  eax
        call  mpc_send_uns
        add   esp,0Ch
        cmp   al,MPC_RET_OK
        jnz   SHORT send_error

; Buffer Request sent succesfully, wait now for completion
send_wait_comp:
        in    al,MPC_MSTAT
        and   al,MPC_SOCMP
        jz    SHORT send_wait_comp

; transfer completed, read completion status
        in    al,MPC_MSOCMP
        and   al,0f0h        ; mask request id
```

```
      jnz  SHORT  send_error
      mov  eax,0
      jmp  SHORT  send_exit

send_error:
      mov   eax,1
send_exit:
      pop   edx
      pop   ecx
      pop   ebx
      pop   es
      pop   ebp
      ret

send_data endp

;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; call from C:  err = recv_data (uns,len,msg,length);
; description:  procedure which sends an unsolicited message which opens
;               a solicited message for receiving data through the DMA.
; input:        unsolicited buffer prepared by the caller (48 bits)
;               the unsilicited message length
;               message pointer (48 bits)
;               message length
; output:       eax:  OK = RET_OK,  error != RET_OK;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
    public  recv_data
recv_data  proc   near

      push  ebp
      mov   ebp,esp
      push  es
      push  fs
      push  ebx
      push  ecx
      push  edx
      push  edi
      push  esi
      lfs   esi,[ebp+014h]

; send the opening message, push its address.
; first, get the data length and put it in message
      mov   eax,[ebp+010h]
      push  eax
      mov   ecx,[ebp+01Ch]    ;data count
      les   eax,[ebp+08h]
      push  es
      push  eax
      mov   es:[eax].length_0_u,cl
      mov   es:[eax].length_1_u,ch
      shr   ecx,8
      mov   es:[eax].length_2_u,ch
      call  mpc_send_uns
      add   esp,0Ch

; receive the buffer request message, and check its status.
; push its address (the same buffer of the opening message).
next_brqt:
      mov   eax,[ebp+010h]
      push  eax
      les   eax,[ebp+08h]
      push  es
      push  eax
      call  mpc_recv_uns
      add   esp,0Ch
      cmp   es:[eax].msg_type_rs,MPC_BUFF_REQUEST
      jne   recv_error
      mov   cl,es:[eax].trnsc_ctl_rs
      mov   endoftrns,ecx
      cmp   cl,02h
      jz    SHORT dma_grant
      mov   cl,es:[eax].error_rs
      and   cl,03Fh
      cmp   cl,0
      jne   recv_error

; set up adma input channel, data address
; length is calculated from the received request
dma_grant:
      mov   ecx,0
      mov   cl,es:[eax].request_id_rs
      mov   edi,ecx
      mov   cl,es:[eax].length_1_rs
      mov   ch,es:[eax].length_2_rs
      shl   ecx,8
      mov   cl,es:[eax].length_0_rs
      mov   dx,fs
      mov   eax,esi
      mov   ebx,ADMA_IN_CHANNEL
      call  adma_handler
      add   esi,ecx

; prepare a Buffer Grant message to send to the requesting MPC
      mov   [grant].dst_id_g,RF2500_HOST_ID
      mov   [grant].src_id_g,CPU_HOST_ID
      mov   [grant].msg_type_g,MPC_BUFF_GRANT
      mov   [grant].request_id_g,0
      mov   ebx,edi
      mov   [grant].liaison_id_g,bl
      mov   [grant].duty_cycle_g,0CAh   ; 2 cycle, 25% bw
      mov   [grant].length_0_g,cl
      mov   [grant].not_used_g,0

; send the buffer grant
      push  8          ; buffer grant length
      push  ds
      mov   eax,OFFSET grant
      push  eax
      call  mpc_send_uns
      add   esp,0Ch
      cmp   al,MPC_RET_OK
      jnz   SHORT recv_error

; Buffer Grant sent succesfully, wait now for completion
recv_wait_comp:
      in    al,MPC_MSTAT
      and   al,MPC_SICMP
      jz    SHORT recv_wait_comp

; transfer completed, read completion status
      in   al,MPC_MSICMP
      and  al,0f0h   ; mask request id
      jnz  SHORT  recv_error
      mov  eax,0
      mov  ecx,endoftrns
      cmp  cl,00h
      jz   SHORT recv_exit
      jmp  next_brqt

recv_error:
      mov   eax,1
recv_exit:
      pop   esi
      pop   edi
      pop   edx
      pop   ecx
      pop   ebx
      pop   fs
      pop   es
      pop   ebp
      ret

recv_data endp

cseg    ends
    end
```

Glossary

absolute pathname. See **pathname, absolute.**

address, external. The address used by a Direct Memory Access (DMA) device in a dual-ported memory system.

address, internal. The address used by the processor in a dual-ported memory system.

advisory lock. See **lock, advisory.**

application buffer. A user-supplied buffer that holds application data. IFX transfers this data between the device and the application buffer.

asynchronous I/O. An I/O operation that executes in parallel with the calling task.

attribute byte. A byte located in the directory entry for a file that defines certain file characteristics, such as read/write permission, volume label, subdirectory, and archive bits.

beginning-of-file. The zero position. See **end-of-file.**

BIOS Parameter Block (BPB). An MS-DOS structure that describes the disk format. It specifies the sector size, number of sectors per cluster, root directory size, starting sector number (for a partition), and other disk format information. BIOS is an abbreviation for *basic input/output system*.

Board Support Package (BSP). An interface program between Ready Systems' components (VRTX32, RTscope, IFX, TPX, TNX, etc.) and the target microprocessor board.

boot sector. The first sector on a volume that contains the BIOS Parameter Block (BPB) and signature bytes. It is read from the disk by the ROM bootstrap program in systems that load from disk. See **volume** and **BIOS Parameter Block.**

buffering. A method for speeding up or reducing the number of I/O operations, using an area of memory to hold data temporarily. See **circular buffering** and **disk buffering.**

byte-oriented RAM disk. See **RAM disk, byte-oriented.**

Cache Manager. See **Disk Buffer Cache Manager.**

Circular Buffer Device Manager. Handles the circular buffer of stream I/O. As a lower-level manager, it serves as the underlying device for the Line Editor Device Manager.

circular buffering. A method of speeding up serial I/O operations using two queues. The input queue holds characters that have been received by the device but not yet read by the application. The output queue holds characters that have been written by the application but not yet transmitted by the device.

client task. A task that requests an I/O operation to be performed by a server task. See **server task.**

cluster. A fixed number of sectors grouped together to form a unit (the smallest unit of allocation) for storing data. The number of sectors per cluster is determined by the disk type and is established when the disk is formatted. Clusters are managed by the File Allocation Table (FAT). See **File Allocation Table (FAT).**

codespace. The memory area that contains a component's starting address.

Component Vector Table (CVT). A structure that integrates Ready Systems software components into a single operating system. The CVT tells VRTX32 which components are present in the system and where they are located.

current position. A descriptor's file location. Reads and writes take place beginning at this point in the file. After the read or write operation the current position is updated to the previous position plus the length of data that was transferred.

cylinder. A conceptual object consisting of all tracks within a disk pack (multiple surfaces) that have the same track number. If each surface contains *n* tracks, there are *n* cylinders.

descriptor. A data structure that defines the characteristics of a connection between a task and an I/O object. See **I/O object.**

descriptor, private. A descriptor that is only accessible by the task that opened it.

descriptor, public. A special type of descriptor used with the Ada run-time system. Public descriptors are accessible by all tasks.

Descriptor Control Block (IFXCCB). A data structure associated with each descriptor. It is used by pathname device drivers.

descriptor ID. A small nonnegative integer that references a descriptor. It is returned when a file is opened.

Device Control Block (IFXDCB). A data structure associated with each IFX device that is used to maintain device specific information. A pointer to the device control block is passed to the device driver when it is called.

Device Driver Control Block (IFXDDCB). A data structure associated with each IFX device driver. It is used for activating a device driver.

device driver. A function IFX calls to handle device installation, device removal, read, write, and I/O control operations.

device manager. A type of device driver that translates high-level I/O requests to simpler operations.

device name. A user-assigned name that specifies a mounted volume or installed device.

device-specific function code. See **function code, device-specific.**

Direct Memory Access (DMA). A technique of transferring data directly between memory and a peripheral without CPU involvement. The CPU is only used to set up the transfer.

directory. A list of files, their attributes, and their locations on the storage media. A system for organizing files into a hierarchical structure. It may contain subdirectories.

directory, root. The top-level directory in a hierarchy of directories. See also **subdirectory.**

directory, tree-structured. A directory hierarchy that begins with the root directory. All directories, with the exception of root, have a parent.

directory, working. A designated directory IFX uses as the starting point in determining all relative pathnames. Each task may have a different working directory.

Disk Buffer Cache Manager. A component of the Disk I/O System that maintains a cache of recently used data to reduce I/O operations to disk.

disk buffering. A method of speeding up disk I/O operations by reading more sectors of data into a memory buffer than what is requested by the current I/O operation. The next request for a sector of data can be retrieved from the buffer.

dismounting a volume. The process of informing IFX that a volume is no longer available for disk management or file I/O operations. It removes the association between the disk drive and the media residing in the drive. See **mounting a volume.**

dual-ported memory. Memory that can be accessed by both a processor and DMA.

elevator algorithm. A specific algorithm design that alternately selects I/O requests in increasing order by starting sector number, then in decreasing order by ending sector number.

end-of-file. The position immediately following the last byte in the file.

event flag. A bit within an event flag group that changes to reflect the occurrence of a specified event. See **event flag group.**

event flag group. A global, long-word (32-bit) structure in VRTX32 Workspace. Each of the 32 bits in the event flag group is an event flag. See **event flag.**

exclusive lock. See **lock, exclusive.**

extent. A collection of contiguous clusters. The number of clusters in an extent depends on the physical volume and the user's request for space allocation. Associated records are not necessarily stored contiguously. See **cluster.**

external address. See **address, external.**

file. A collection of related data sets that are used as a unit.

File Allocation Table (FAT). A section of the disk that stores information on disk-file space usage. It contains an entry for each cluster that specifies whether the cluster is assigned, unassigned, or marked as bad. See **Media Format Byte**.

file lock. See **lock, file**.

filename. A string used to uniquely identify a file. It consist of a main part of up to eight characters, optionally followed by a period and an extension of up to three characters. The filename is the last component of a file's pathname. See **pathname**.

file, working. A temporary collection of data sets that is destroyed once the data is utilized or transferred to another form.

File Manager. A Disk I/O System component that reads and writes media compatible with MS-DOS version 4.0.

format. The physical separation between sectors.

formatting. The process of writing the format pattern and initial data to each sector of the disk.

formatting, logical. Imposes a logical structure of files and directories that prevents data from being accessed by name. The MS-DOS volume manager is responsible for doing logical formatting.

formatting, physical. Writes the timing information for each sector onto the disk. It destroys any old data by overwriting it with a meaningless pattern. The disk driver is responsible for doing physical formatting.

function code. A code that specifies a particular IFX operation to be performed.

function code, device-specific. Function codes that are unique to one device type.

function code, generic. Function codes that are applicable to all device types.

function code, optional. Function codes that may or may not be implemented by the driver.

function code, pathname. Function codes whose first parameter is a pathname string. Ordinary device drivers do not need to implement pathname function codes.

function code, required. Function codes that must be implemented by the driver.

interleave factor. A number that controls the sectors numbering within a track. For performance reasons, the sectors order is not their physical order. The interleave factor–1 is the number of physical sectors that separates two consecutive sector numbers. See **sector interleaving**.

internal address. See **address, internal**.

Interrupt Service Routine (ISR). User-supplied code that is activated by a hardware device interrupt.

I/O object. A file, device, or volume.

ISR, receiver. An interrupt service routine that handles interrupts for incoming characters.

ISR, transmitter. An interrupt service routine that handles interrupts when the transmitter is ready to accept another character.

line buffer. A memory area used in line-editing mode only to build an input line based on the special characters received.

line-editing mode. An IFX feature used for high-speed binary communication between terminals. Line-editing mode functions include echoing, erase character, erase word, erase line, conversion of newline to carriage-return linefeed, and simulation of tab stops.

Line Editor Manager. A component of the Stream I/O System that handles echoing and editing of characters by an operator.

lock, advisory. Using *ifx_lockf* and *ifx_unlockf* to implement a locking policy that works only if all tasks follow a consistent lock protocol. Advisory locks do not prevent other tasks from reading or writing a file, only from obtaining the lock.

lock, exclusive. Prevents all other tasks from using a file until it is unlocked. See **lock, shared.**

lock, file. The use of semaphores or system calls to prevent two or more tasks from accessing the same file at the same time for I/O operations.

lock, nested. Embedding multiple locks on a file. Nested locks must all be made by the same task and specified for the same use (exclusive or shared).

lock, shared. Used to prevent other tasks from applying an exclusive file lock; however, it does not prevent other tasks from using the file. See **lock, exclusive.**

logical formatting. See **formatting, logical.**

mailbox. A user-defined pointer-size location in user read/write memory used to coordinate data transfer between IFX and a device driver.

master boot sector. A reserved area, usually the first section on the disk, that describes the disk's configuration. See **partition** and **partition table.**

Media Format Byte. The first byte of the FAT, used in early versions of MS-DOS to identify volume format. See **File Allocation Table (FAT).**

mounting a volume. Establishing a disk or partition as an available IFX volume by assigning it a logical name. See **dismounting a volume.**

named pipe. A first-in-first-out queue that is similar to the UNIX pipe.

nested lock. See **lock, nested.**

optional function code. See **function code, optional.**

parameter. A variable or constant that is passed to a function.

parameter list. A list that defines parameters required for a specified function.

partition. MS-DOS allows a hard disk to be divided into any number of logical disk drives that are called partitions. Each partition may be formatted to use different parameters. See **master boot sector.**

partition table. A part of the master boot sector that contains four 16-byte entries that describe a particular partition. See **partition.**

pathname. A string that identifies a file relative to the root directory or the current directory.

pathname, absolute. A string that begins with the root directory and explicitly specifies each subdirectory leading up to the final file.

pathname driver. A device driver or manager that implements a pathname function code.

pathname function code. See **function code, pathname.**

pathname, relative. A string that begins at the working directory and specifies files or subdirectories in terms of their relative location to that directory, leading to the final file.

physical formatting. See **formatting, physical.**

physical mode. Used for high-speed binary communication between computers. It does not give special treatment to any characters, but merely passes them through to the application.

private descriptor. See **descriptor, private.**

property function. A function that inquires or explicitly changes file properties.

public descriptor. See **descriptor, public.**

RAM disk. A pseudo-device implemented entirely by software, which emulates a real disk. The disk data is stored in random access memory (RAM) instead of magnetic media.

RAM disk, byte-oriented. A RAM disk that is addressable in byte units. A volume can be mounted directly on top of a byte-oriented RAM disk.

RAM disk, sector-oriented. A RAM disk that is addressable in sector units. A disk buffer cache must be mounted on top of a sector-oriented RAM disk. The volume is then mounted on top of the disk buffer cache.

receiver ISR. See **ISR, receiver.**

receiver serial device. See **serial device, receiver.**

reentrant device driver. A device driver that can service more than one device attached to a single controller board and several controller boards, if they are the same kind.

reentrant function. A function that can be used by more than one task at the same time.

relative pathname. See **pathname, relative.**

required function code. See **function code, required.**

reserved bytes. The ten extra bytes in a directory entry that are reserved for future MS-DOS use.

return code. An integer value that indicates the status of a completed operation, and that specifies the reason for any failure.

root directory. See **directory, root.**

RTscope. A real-time debugger and VRTX32 System Monitor for use with VRTX32-based software systems.

sector. Evenly divided subsections on a track that hold stored data. A sector is the smallest addressable space on a disk's media.

sector interleaving. A method of assigning physical sector numbers in an order that skips alternating sectors. See **interleave factor.**

sector number. An integer from zero to the total number of sectors minus one. Sector numbers are used to address a particular sector.

sector-oriented RAM disk. See **RAM disk, sector-oriented.**

Serial Control Block (IFXSCB). An internal IFX data structure. However, in order to improve performance of the serial devices, two of the structure's fields are used to pass service routine addresses from IFX to the driver.

serial device, receiver/transmitter. The receiver accepts an incoming bytestream over the communication medium, such as an RS-232 link or Centronics parallel interface. The transmitter sends a bytestream over the communication medium.

serial_receive_character. The receiver ISR calls this routine to transfer each character to the input buffer as the character is received from the USART.

serial_transmit_ready. The transmit ISR calls this routine to tell IFX the device is ready. The next character in the output buffer (if any) is returned to the ISR.

server task. A task that performs an I/O operation on behalf of a client task. See **client task.**

shared lock. See **lock, shared.**

signature byte. The last two bytes of the boot sector, which contain the value 0x55AA. See **boot sector.**

software trap. A special processor instruction used to enter the operating system (for example, VRTX32, or IFX).

status code. An integer value returned by each IFX function that indicates the disposition of the requested operation. This term is used interchangeably with return code. See **return code.**

subdirectory. All directories below the root directory.

synchronous I/O. An event or operation that does not return control to the caller until the physical I/O is complete.

timeout. The maximum period of time that a driver waits for an I/O operation to complete.

Index

READY SYSTEMS

Chapter 2

Calling Conventions

2.1 Introduction

All device drivers use a particular set of calling conventions to interact with a device. This chapter explains the guidelines for interfacing IFX device drivers and managers with your device. More detailed information for each device type, and how to install devices can be found in the remaining chapters of this manual.

IFX and the device driver follow the conventions described below.

- The device driver should only be called by IFX. Do not call the device driver directly from an application task or other user code.
- The device driver runs in the context of a task, as opposed to an interrupt handler. However, serial device drivers are devided into two parts: one part runs in a context of a task, and the other part is an ISR.
- IFX calls the device driver with interrupts enabled. If the device driver needs to disable interrupts, it should restore the interrupt status before returning.
- For the M68000 processor family, the device driver executes in supervisor mode. In particular, on the 68020 and 68030, the device driver uses the regular supervisor stack, as opposed to the interrupt stack.
- For the 80386, the device driver executes at privilege level 0.
- The device driver is allowed to call VRTX32 services, such as *sc_pend*, that suspend the caller.
- We do not recommend that the device driver call any IFX services. In particular, be aware of the potential for deadlock if the device driver calls an IFX service that involves another device.
- The device driver should return when it completes its operation or detects an error.

There are also rules that govern the use of parameters, registers, and status codes. These conventions are described in the following sections.

track. The path along which information is recorded on a magnetic disk. Tracks are organized in concentric rings and each track is divided into sectors. Any location on a track can be accessed directly without a sequential search.

transmitter driver. A routine that IFX calls to transmit the first character of a given output stream.

transmitter ISR. See **ISR, transmitter.**

transmitter serial device. See **serial device, transmitter.**

tree-structured directory. See **directory, tree-structured.**

update. To read and modify a record with current transaction information, then rewrite the same fixed-length record. Updates also set the current position back to access the same record twice. See **current position.**

virtual device. A conceptual device that possesses the characteristics of a real hardware device.

volume. A disk or partition in MS-DOS format. It consists of the boot sector, FAT, root directory, and data storage space.

volume label. A string of up to 11 characters used to identify the media. It is not necessarily the same as the volume name.

volume synchronization. Writing all data in the buffer cache to the disk.

VRTX32. A high-performance Versatile Real-Time Executive, silicon-software component for embedded microprocessors.

VRTX32 Workspace. A memory area that contains the system variables, task and int stacks, TCBs, control structures for queues and VRTX32-managed user memory, event flag groups, and semaphores.

working directory. See **directory, working.**

working file. See **file, working.**

workspace. The read/write memory a component uses during execution.