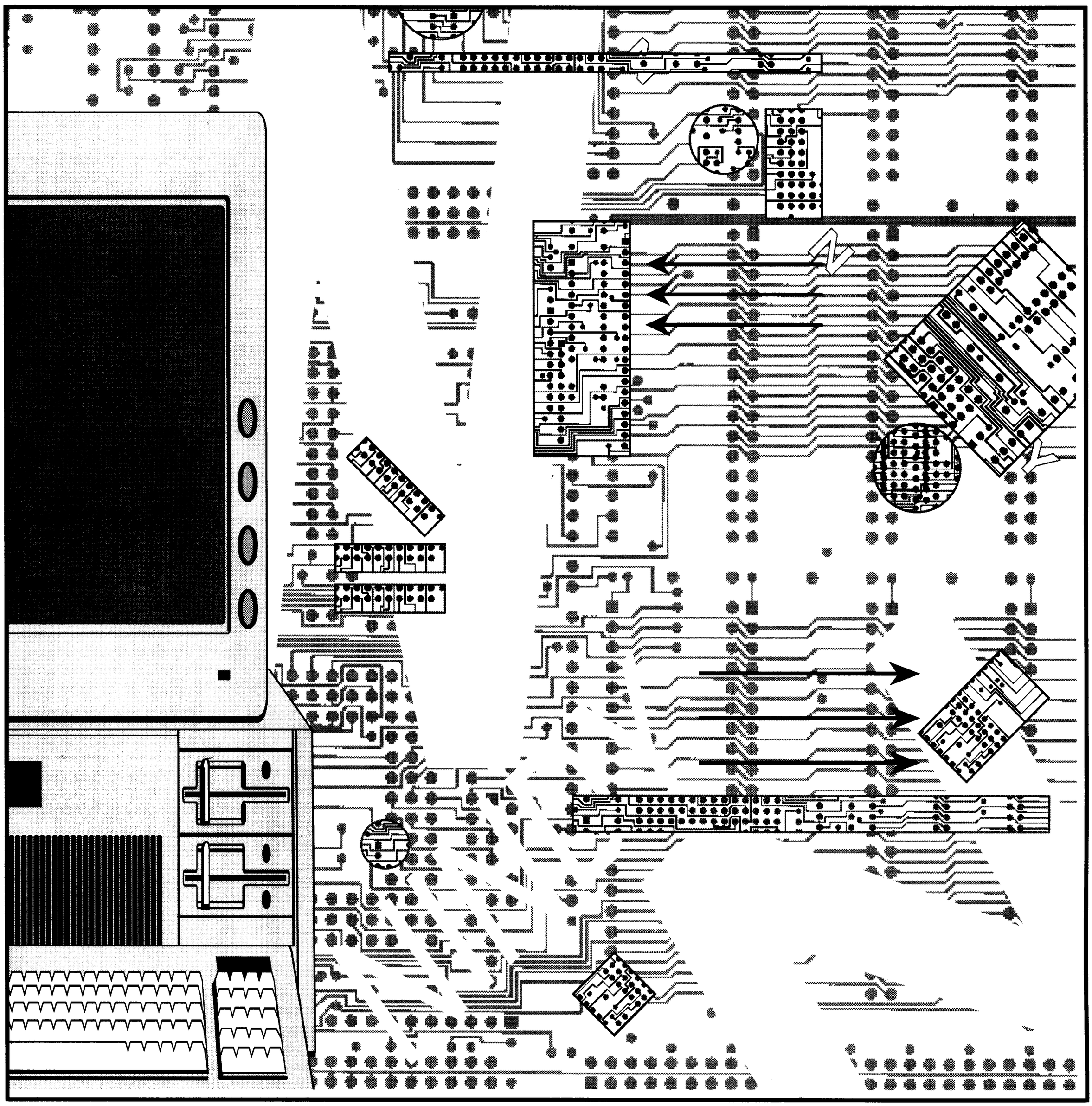
intel® NINDY User Manual



Order Number: 270874-001

Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Flash Programming Algorithm
Copyright © 1988-1990, Intel Corporation, all rights reserved.

NINDY User Manual
Copyright © 1990, Intel Corporation

Intel hereby grants you permission to copy, modify, and distribute this software and its documentation. Intel grants this permission provided that the above copyright notice appears in all copies and that both the copyright notice and this permission notice appear in supporting documentation. In addition, Intel grants this permission provided that you prominently mark as not part of the original any modifications made to this software or documentation, and that the name of Intel Corporation not be used in advertising or publicity pertaining to distribution of the software or the documentation without specific, written prior permission.

Intel Corporation does not warrant, guarantee or make any representations regarding the use of, or the results of the use of, the software and documentation in terms of correctness, accuracy, reliability, currentness, or otherwise; and you rely on the software, documentation and results solely at your own risk.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

376™	Im®	iSBC®	PC BUBBLE™
386™	iMDDX™	iSBX™	Plug-A-Bubble™
386SX™	iMMX™	iSDM™	PROMPT™
486™	Insite™	iSXM™	Promware™
Above™	Intel®	KEPROM™	QueX™
BITBUS™	intel®	Library Manager™	QUEST™
COMMputer™	intelBIOS™	MAPNET™	Quick-Erase™
CREDIT™	I2ICE™	MCS®	Quick-Pulse Programming™
Data Pipeline™	Intelevision™	Megachassis™	Ripplemode™
ETOX™	inteligent Identifier™	MICROMAINFRAME™	RMX/80™
GENIUS™	inteligent Programming™	MULTIBUS®	RUPI™
i®	Intellec®	MULTICHANNEL™	Seamless™
Λ™	Intellink™	MULTIMODULE™	SLD™
ICE™	iOSP™	ONCE™	SugerCube™
iCEL™	iPAT™	OpenNet™	UPI™
iCS™	iPDS™	iOTP™	VLSiCEL™
iDBP™	iPSC®	InBoard™	4-SITE™
iDIS™	iRMK™	Intel376™	Intel486™
iLBX™	iRMX®	Intel386™	Intel Certified™

and the combinations of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

Multibus® is a patented Intel bus.

First Printing April, 1990

Table of Contents

Introduction

Chapter 1 — Features

Chapter 2 — Reconfiguring

Overview for Reconfiguring NINDY 2-3
Details for Reconfiguring NINDY 2-5
Hardware Specific File Descriptions 2-7

Chapter 3 — Compilation

Compiling NINDY 3-2
Compiling an Application Program for Running on NINDY 3-7

Chapter 4 — Command Syntax

Chapter 5 — NINDY Theory of Operation

System Initialization for the 80960Kx 5-1
System Initialization for the 80960Cx 5-7
Faults 5-10
Interrupts 5-12
Stacks 5-14
Execution 5-17
Debug Methods 5-21
Breakpoints 5-22
Trace Events 5-24
Following The Trail 5-25
Monitor Features 5-27
Download 5-28
Download to Flash 5-30
Memory Display 5-31

Appendix A — NINDY File Descriptions

Appendix B — C-Library Calls

List of Figures

2.1 – Memory Map for the QT960 2-2
2.2 – Memory Map for the EV80960CA 2-2
2.3 – Structure for Putting the Pieces of NINDY Together 2-6
2.4 – Call Sequence for I/O 2-13
3.1 – Interleaved EPROMs 3-3
5.1 – Minimum System Configuration for the 80960Kx 5-2
5.2 – NINDY 80960Kx System Configuation 5-3
5.3 – Assembly Code For NINDY Kx System Configuration 5-4
5.4 – Fault Paths 5-11
5.5 – Stack Switch 5-15
5.6 – Stack Progression 5-16
5.7 – Memory Allocation for QT960 RAM 5-18
5.8 – Trace Bit Locations 5-21
5.9 – Use and Recovery of Steps for Software Breakpoints 5-22
5.10 – Stack Trail 5-26
5.11 – NINDY File Map 5-27
5.12 – Host-cable-Target Board 5-28
5.13 – Downloading to Memory 5-29

Chapter 1
Features

NINDY has several features designed to make debugging easier. Many of these features take advantage of features in the 80960 architecture and the processor specific implementations. There are several forms of memory display, including display byte, short, word, double word, triple word, and quad word. These display forms take advantage of the different size load instructions in the 80960 architecture. There are also displays of real, long real, and extended real floating point numbers in memory. Modification of bytes and words in memory are supported, but there is no support for modification of a floating point number. Memory locations may also be disassembled. Display and modification of registers is supported (no modification of floating point registers or special function registers though).

The monitor is designed for program execution, with single step capabilities, and several types of instruction tracing (return, supervisor, call, and branch instructions). Up to two hardware breakpoints can be set. The use of hardware breakpoints allows breakpoints in EPROM or Flash memory to be set. The breakpoint, trace and single step instructions take advantage of features in the 80960 processor. Software breakpoints are also available to expand the number of possible breakpoints.

NINDY communicates through a UART, either physical or emulated. Two types of object file download are supported, download to RAM and download to Flash EPROM. The download protocol used is Xmodem, a public domain program found in many terminal emulation and communications packages. The interface was designed to operate at the rate of 9,600 baud, but in theory, it will operate at any baud rate acceptable to the host. It has been tested up to a rate of 38,400 baud. User changeable baud rate is another NINDY feature.

There is a software board reset command. For the QT960 this is implemented using the 82380 peripheral. This peripheral also supplies the benchmark timer, Flash EPROM programming timer, and DMA functionality. The last three functions are part of the NINDY "libqt.a" user library. For the EV80960CA, the timing features are supplied by an 82c54 peripheral timer. The hardware specific library for the EV80960CA is called "libevca.a". The "libnin.a" library provides support for the standard I/O (stdio) calls for console input and output. The standard I/O support allows the use of C functions like printf and scanf which talk to the console. There is no file access (open, close, fprintf) support for serial communication implementations, although the low level hooks exist for their implementation.

Chapter 2
Reconfiguring

Your custom hardware is probably not the same system setup as any one of the monitor configurations provided. We provide for you the source code for the low level system library, "libnin.a", as well as source for two board specific libraries as examples for writing your own system and board specific libraries. We provide startup code for the Cx and Kx, and main monitor source so you can customize NINDY to have the features you require in your environment. Link/load directive files are provided as examples of how lnk960 knows what to link where.

Both the QT960 and the EV80960CA use the 82510 serial communications chip. The QT960 library, "libqt.a" has user callable routines which allow the user to take advantage of the 82380 peripheral chip, QT960 board features, and Flash EPROM. These routines are found in a directory called LIBQT. The EV80960CA uses an 82c54 timer, the code for which is in the "libevca.a" library, found in the directory LIBEVCA. To simplify the porting of this monitor to other hardware, the hardware dependent routines are placed together into directories called QT960 and EVCA. Only the hardware dependent routines and libraries should need to be rewritten for porting.

The system libraries are provided in a directory called LIBNIN. These files provide a run time interface for C through stdio library calls to the console I/O. The file manipulation routines are not implemented, however, because file manipulation across a serial link at 9600 baud is just not practical, and requires specific host software to communicate with. The design of NINDY is to hook to many different host interfaces. There are hooks in place to implement the file I/O routines, which you may choose to implement along with your own host software interface.

The linker directive files, "qtrom.ld" and "qtnindy.ld", set the QT960 memory as shown in Figure 2.1, and the files, "evrom.ld" and "evnindy.ld", set the EV80960CA memory as shown in Figure 2.2.

For a complete listing of NINDY source files, refer to Appendix A. Changes may be made to add other features or make enhancements to the existing monitor by modifying files in the base directory or architecture specific (Kx or Cx) directories.

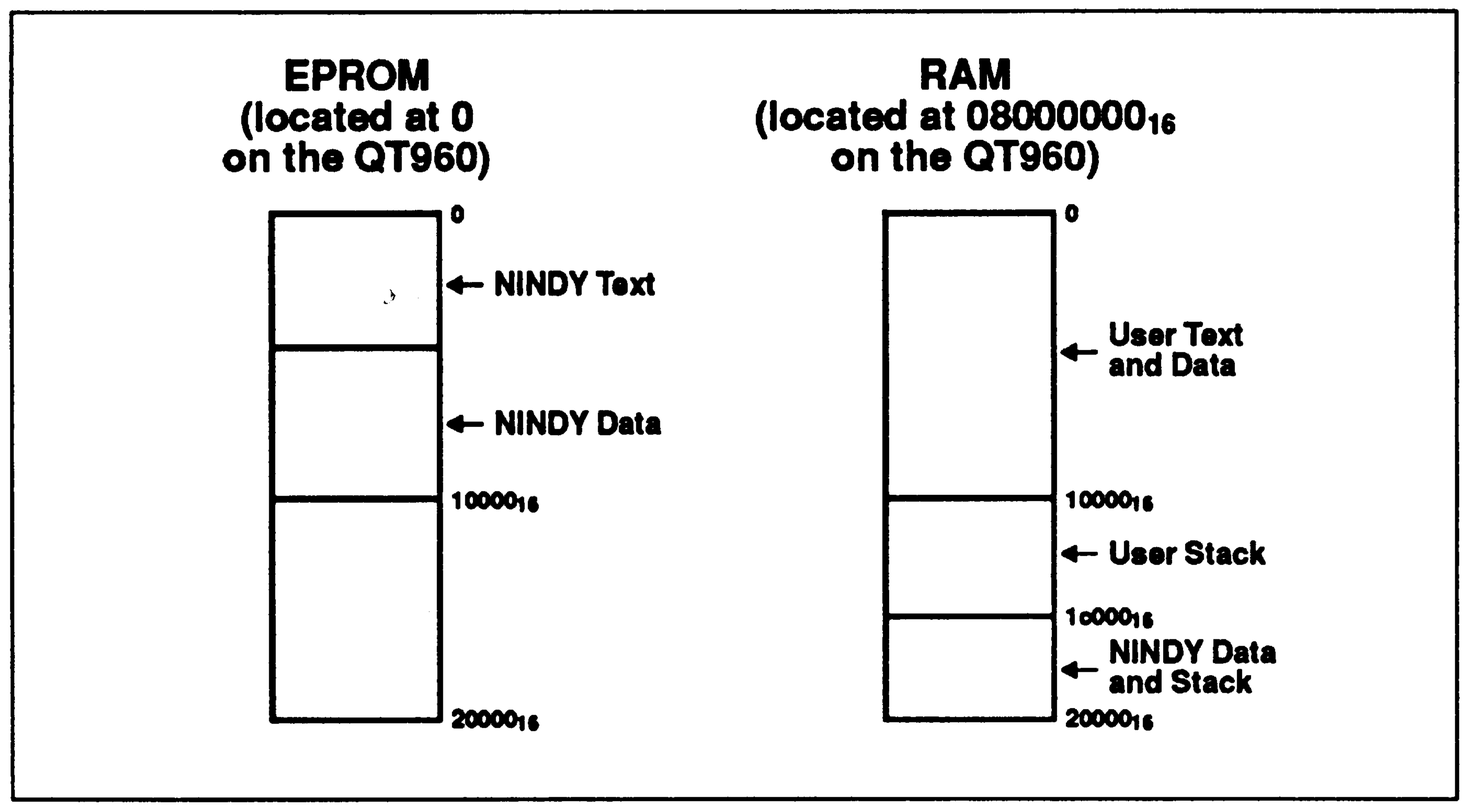


Figure 2.1 *— Memory Map for the QT960*

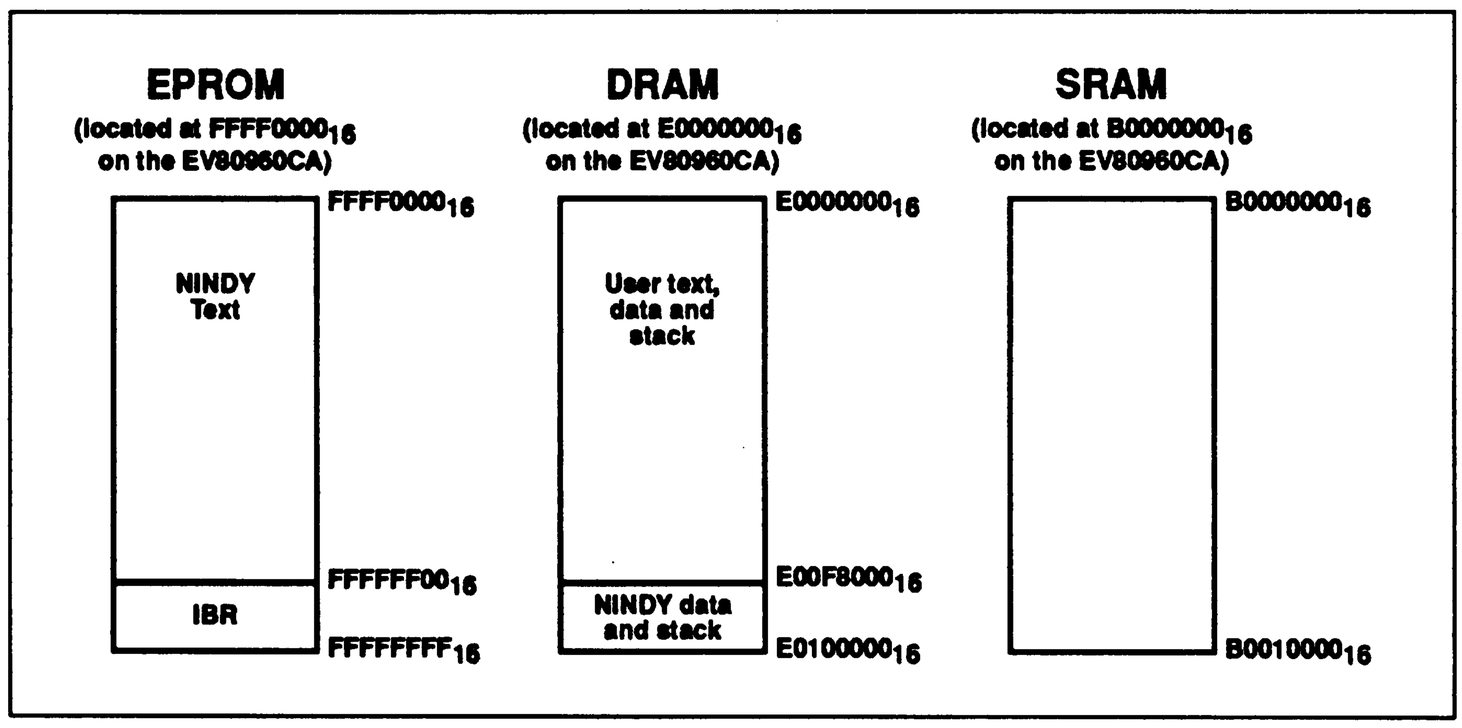


Figure 2.2 *— Memory Map for the EV80960CA*

Overview for Reconfiguring NINDY

Note *— The batch files supplied with NINDY assume that ic960 v1.2-6 has been installed (the command "ic960 -V" produces the version number 1.2-6). Each NINDY directory contains a subdirectory NEW, which contains batch files or in the case of the library directories, ".ld" files. The batch and ".ld" files are designed for ic960 v2.0, but have not been validated against the official release, as it was not available at NINDY release. If you are using ic960 versions greater than v1.2-6, copy the contents of the NEW directories into their respective base directories.*

The NINDY structure is broken down into a base directory, a CX or KX directory, and a board specific directory QT960 or EVCA. This allows maximum flexibility in reconfiguring NINDY for any type of 80960 processor running on any board configuration. The files which need to be reconfigured for different hardware are also minimized.

To configure a board with an 80960KB processor, you would need to compile all of the base directory files, all of the KX directory files, and then copy the files from the hardware specific directory which best matched your hardware (let's assume it's QT960) to your own hardware directory (let's call it NEW). You would then need to rewrite the files in NEW, renaming them to "new_io.c", "new_dl.c", "new_data.c", and "new_hw.c". The basic functionality for these files is described in the next section, along with proper stubs for functions which cannot be duplicated on your hardware. If you are using a pc, change the MAKE.bat file in NEW to reflect your new file names and change the target of the arc960 to make a mini-library named "new.a".

Configuring NINDY for a board with an 80960CA processor is similar to configuring NINDY for the 80960KB processor. You will need to compile all of the base directory files, all of the CX directory files, and then copy the files from the hardware specific directory which best matched your hardware (let's assume it's EVCA) to your own hardware directory (let's call it NEW). You would then need to rewrite the files in NEW, renaming them to "new_io.c", "new_dl.c", "new_boot.s", and "new_hw.c". The basic functionality for these files is described in the next section, *Details for Reconfiguring NINDY*, along with proper stubs for functions which cannot be duplicated on your hardware. Change the MAKE.bat file in NEW to reflect your new file names and change the target of the arc960 to make a mini-library named "new.a".

You will also need to build a "newnindy.ld" and "newrom.ld" to copy to your I960BASE/LIB directory. You can use "qtnindy.ld" and "qtrom.ld" or "evnindy.ld" and "evrom.ld" as your guidelines. You need to alter the lines in the bottom portion of the files "newnindy.ld" and "newrom.ld". The output of the rom960 utility should be changed from "qt0.hex" to "new0.hex", from "qt1.hex" to "new1.hex" etc. if you are using "qtrom.ld", and "evca.hex" should be changed to "new.hex" if you are using "evrom.ld". You should then change the addresses for both ".ld" files to reflect the address space on your new board. For more information on configuring the ".ld" file for your hardware see the *80960 Utilities User's Guide* (order number 408143). Chapters on both lnk960 and rom960 are contained in that document.

The batch file QTMAKE.bat or EVCAMAKE.bat can be copied to NEWMAKE.bat, and all references to "qt960" or "evca" can be then changed to "new". The NEWMAKE.bat file will change directories to NEW, build "new.a", change directories to KX and build "kx.a", or change directories to CX and build "cx.a", and then build the base files into your newly reconfigured NINDY.

You might have noticed there are libraries associated with NINDY. "libnin.a" is used by all NINDY versions and handles I/O read/write functionality. It does not need to be reconfigured for different hardware. The other library, "libevca.a" or "libqt.a", deals with hardware specific functionality such as benchmarking timers, Flash programming (on the QT960 board), or any other functions which may be present on your hardware which you might want to call from a library. The only "standard" functions which you may want to copy are the bentime() routines. You would then make a directory LIBNEW. Copy the files from the closest hardware equivilant library directory (let's pick LIBQT for our hardware specific library example), and change "time_qt.c" to "time_new.c" changing all relevant information inside the files also. The "libevca.a" library also has a non-interrupting benchmark timer which may be called instead of the interrupting timer. That timer is found in noint_tm.c in the LIBEVCA directory. When you have all of the library functions written which you want to include, change your make.bat file to reflect what is in your directory, making sure the target of arc960 is "newlib.a". You might want to be sure that the included files use the right addresses for your board. Include files which are shared by more than one directory are found in the INCLUDE directory. Include files which are used by only one directory are found in the directory in which they are used. Be sure to copy "newlib.a" to your I960BASE\LIB directory when you are done.

The "crt*nin.o" files will work with all configurations of NINDY, and do not need to be reconfigured for different hardware. They are not however, compatible with NINDY versions earlier than V1.4.

Details For Reconfiguring NINDY

There are several choices which you will need to make when configuring NINDY for custom hardware. Do you want numerics (floating point)? Are you configuring for the Cx architecture or the Kx architecture? What does your hardware look like?

NINDY may be configured for a KA, KB, or CA chip. The following functions are designed to use floating point for the KB and future numerics architectures.

Floating point specific functions:

```
float.s
fp.c
fp_flt.s
```

For the KA, CA and future non-numerics chips, you would use the following functions designed for use with no floating point.

Same functions, without floating point are in:

```
no_fp.c
nofp_flt.s
```

NINDY has files specifically for the Kx architecture, and ones specifically for the Cx architecture (with the same functionality in both).

Kx specific functions are in:

```
kx_break.c
kx.s
kx_ftbl.s
kx_init.s
```

Cx specific functions are found in:

```
cx_break.c
cx.s
cx_ftbl.s
cx_init.s
```

The hardware specific files will need to be changed for custom hardware. Two versions of custom files are provided (QT960 and EV80960CA), and a third is discussed for comparison.

The EV80960CA has EPROM at $ffff0000_{16}$, DRAM at $e0000000_{16}$, SRAM at $b0000000_{16}$, an 82C54 timer at $dfe00000_{16}$, and two 82510 UART ports, one at $dfc00000_{16}$ and one at $dfd00000_{16}$. It is a stand-alone board which communicates via serial cable.

Files with functions common to all NINDY boards, but are implemented specific to the EV80960CA:

```
copymem.s
evca_dl.c
evca_io.c
evca_hw.c
```

The Cx implementations also have a boot file ("ev_boot.s" for the EV80960CA) which is hardware specific, but its function is needed for every Cx NINDY implementation.

The QT960 has EPROM at 00000000_{16}, Flash EPROM at 10000000_{16}, SRAM at 08000000_{16}, an 82510 UART at 28000000_{16} and an 82380 peripheral (DMA, Interrupts, Timers) at 18000000_{16}. It is also a stand-alone board which communicates via serial cable. The QT960 is an 80960Kx board.

Files with functions common to all NINDY boards, but are implemented specific to the QT960:

```
copymem.s
qt_dl.c
qt_io.c
qt_hw.c
flash.c
```

The iLASER III board is a PC add-in card which communicates via shared memory on the PC. It has DRAM located at 00000000_{16}, and an 8254 timer located at 800000_{16}. The iLASER III is an 80960Kx board.

Files with functions common to all NINDY boards, specific to the iLASER III (not provided, but discussed for comparison):

```
nocpymem.c
laser_dl.c
laser_io.c
laser_hw.c
```

For the iLASER III implementation of NINDY, "nocpymem.c" is substituted for "copymem.s" as a stub. Since on this board, the 960 boots out of RAM (there is no EPROM to boot from on the iLASER III board) there is no need to copy the data structures to RAM using "copymem.s".

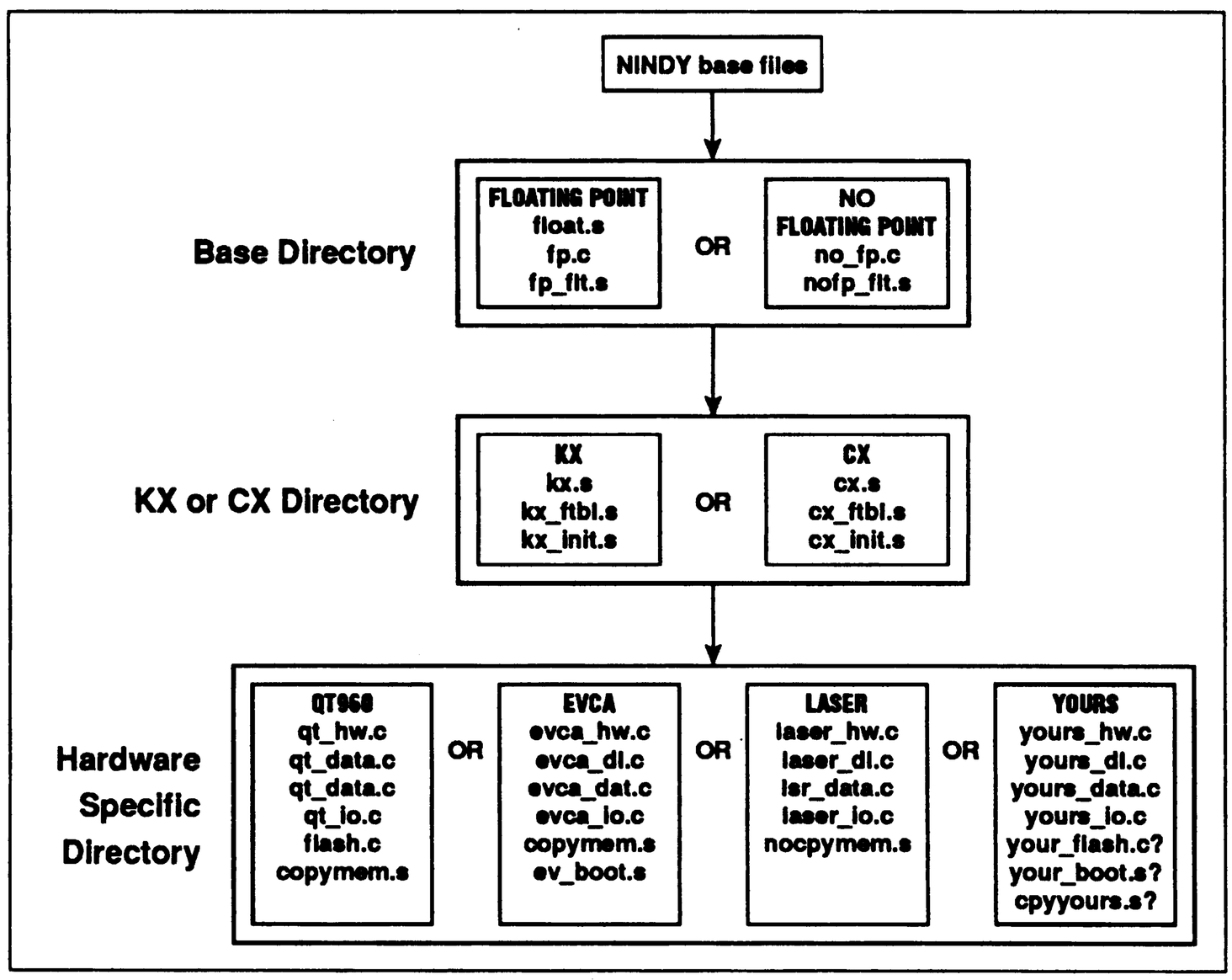


Figure 2.3 *— Structure for Putting the Pieces of NINDY Together*

Hardware Specific File Descriptions

The following describes in detail the necessary functions which make up the hardware specific section of NINDY. If your hardware does not support the functions, or you do not want these functions of NINDY, you may make stubs of some of the functions. After the description of each function is a description of what the proper stub for that function would be. All constants are defined in the files "defines.h" and "qtcommon.h" (for the QT960 implementation) found in the INCLUDE directory.

Flash EPROMs have been mentioned several times. Flash EPROMs are electrically erasable and programmable EPROMs. Flash is useful for testing boot code and other non-volitile code which would usually be burned into EPROM, however Flash memory is easily erasable and programmable like RAM. The QT960 evaluation board has Flash memory as one of its features. It uses the Intel 28F256 devices and the programming algorithms in NINDY are for the Intel family of Flash memory. As mentioned, Flash memory does require special programming and erasure algorithims and the ones in NINDY are taken from the Application Note AP-316, *Using the 28F256 Flash Memory for In-System Reprogrammable Nonvolatile Storage* (order number 292046), and modified to also program larger Flash devices using the algorithms found in the *28F010 1024 K (128 K x 8) CMOS Flash Memory* data sheet (order number 290207). *The 28F256 256 K (32 K x 8) CMOS Flash Memory* data sheet may also be of interest to you (order number 290158).

FILE "xxx_hw.c" (and "flash.c")

FUNCTION board_test()

This function calls a boot-time board confidence test. For the QT960 board, this function calls a suite of confidence tests which test the major functions of the board. The suite includes tests to read and write SRAM, read EPROM, read Flash EPROM, test the UART, test the interrupt registers in the 82380, test the timers in the 82380, test the LED's, and test the programmable wait state generator registers on the board. This function is implemented in the file "qt_hw.c".

As a stub, this function should return VOID. This function is called during initialization from the file "xxx_init.s". This function is a stub in the file "evca_hw.c".

FUNCTION check_flash(int type)

This function takes an argument, type, which is the value returned from init_flash(). Optionally, the type variable may be read from the identifier location on the Flash EPROM (see a Flash data sheet for more information on the identifier command). If the Flash is not readable or not present (based on the value at the identifier location), it returns ERROR (-1). If the Flash is present it uses the size information found in the identifier bits to read from the beginning to the first programmed word, and from the end backwards to the last programmed word. It then reports either that the Flash space is blank, or that it is programmed between the first and last programmed locations, and what the size is of the total programmable space. This function is implemented in "qt_hw.c".

As a stub this function should return VOID and may print out a "not implemented" message. This function is the implementation of the "cf" command. This function is a stub in the file "evca_hw.c".

FUNCTION disable_ints()

This function disables all external interrupt sources on the board by programming the peripheral chips to known states. The 80960 is not able to mask all interrupts (they automatically get posted to the interrupt table, just not serviced), so all possible external interrupt sources should be programmed to a known state to avoid spurious interrupts. Any interrupting timers are programmed to count to zero and stop, any interrupt controllers are programmed to known interrupt levels or masked off, and any interrupting UART's are programmed to a known state. This function is implemented in both "qt_hw.c" and "evca_hw.c".

As a stub, this function should return VOID. This function is called during initialization from the file "xxx_init.s".

FUNCTION down_flash(int monitor_flag)

This function takes an argument, monitor_flag, which is used to monitor whether a program is being executed or not. The variable is then passed along to download() if Flash is present on the board. If there is no Flash present on the board, this command should be left as a stub, since this function is only called from the download to Flash EPROM, "df", command. This function is implemented in the file "qt_hw.c".

As a stub, this function should return VOID and may print out a "not implemented" message. This function is the implementation of the "df" command. This function is a stub in the file "evca_hw.c".

FUNCTION erase_flash()

This function erases the Flash EPROMs as specified by the algorithm in the Flash Application Note (see above). It returns ERROR (-1) if an error in erasing occurs. This function is implemented in the file "flash.c".

As a stub, this function should return VOID and may print out a "not implemented" message. This function is the implementation of the "ef" command. This function is a stub in the file "evca_hw.c".

FUNCTION init_flash()

This function initializes the variables used for programming Flash. It then reads the identifier location found on the Flash EPROM and uses the size information found there to return the size of the programmable space. If there is no Flash present, or the Flash is unreadable, ERROR (-1) is returned. See a Flash data sheet for more information regarding Flash EPROMs. This function is implemented in the file "flash.c".

As a stub, this function should return ERROR (-1). This function is called by many of the Flash programming functions and is important to implement if Flash is to be used. This function is a stub in the file "evca_hw.c".

FUNCTION reset()

This function resets the board. It may be necessary to turn off free running timers or other peripherals in this function. This function may be connected to a hardware reset on the board, or may be a software re-init (go back to the beginning). The "hwqt.c" file uses a reset in the 82380 to reset the 960 chip. The "hwevca.c" and "hwlaser.c" go back to the initial reinit sysctl and reinit IAC respectively found in "cx_init.s" and "kx_init.s".

As a stub, this function should return VOID and may print out a "not implemented" message. This function is the implementation of the "rs" command.

The NINDY download function understands only executable files in the Common Object File Format (COFF) which is produced by the Intel lnk960 linker. It uses the location information found in the COFF header to figure out where to place the data being downloaded. The download protocol used by NINDY is Xmodem, a public domain data transfer protocol. The host terminal sends 128 byte packets containing the COFF executable file to NINDY, and NINDY takes those packets and reassembles them into the COFF header, text section, data section, bss (uninitialized data) section, and any other sections specified in the header. The download consists of two basic parts then, the Xmodem protocol which checks the incoming packet for a correct checksum and responds accordingly, and the COFF parser which places the contents of the packet where it belongs. The Xmodem part (receive_xmodem()) runs in a loop reading packets until an end of file is reached. Receive_xmodem() then calls parse_coff() for every good packet it receives. Parse_coff must then be reentrant, keeping track of where it ended in parsing the previous packet so the current packet is parsed correctly. Parse_coff() uses a state machine (a big switch statement) to keep track of its state. The initial case is P_INIT (parses out the COFF header information), the second case is P_SECT (parses out the COFF section information), the third case is P_RAWDATA (parses the raw data into the respective sections and programs it into memory), and the last section is P_DONE (nothing left to do).

FILE "xxx_dl.c"

Note *— NINDY I/O is polled not interrupt driven.*

FUNCTION check_download()

This function checks the start address of each of the sections to be downloaded (given in the COFF header) against the memory map for the board, and assigns each section a FLASH (0) or RAM (1) flag (since Flash memory needs to be programmed differently) to the global array downtype[], for use later in down_bytes(). Check_download() is called after the COFF header has been downloaded (the first section in a COFF executable). Check_download() returns ERROR (-1) if a section is assigned a FLASH flag and Flash is not programmable, otherwise it returns 0. This function is implemented with Flash in the file "qt_dl.c", and without Flash in the file "evca_dl.c".

If Flash is not present on the board all sections should be assigned a RAM flag. This function is called from the download() function and is important for implementing the download facilities.

FUNCTION down_bytes(unsigned char *inbuffer, int size, int sect)

This function takes a pointer to a buffer of bytes to be downloaded, the number of bytes to be downloaded, and which section of the COFF file we are downloading from as arguments. It then checks the global array downtype[] (indexed by the COFF section variable) flag to see if we are writing to Flash or RAM (this flag was set in the function check_download()). If the section flag is FLASH, it calls the Flash programming function, download_flash(), to program the buffer into the Flash EPROMs. If the section flag is RAM it simply copies the buffer to the proper address in RAM. It returns ERROR (-1) if the Flash programming causes an error, otherwise it returns 0. This function is implemented with Flash in "qt_dl.c", and without Flash in "evca_dl.c".

If Flash is not present on the board, this function should simply copy the buffer to the proper address in RAM. This function is called by the download() function and is important for implementing the download facilities.

FUNCTION parse_init()

This function sets the global variable parser_state to P_INIT (0) for use by the download function parse_coff(). If Flash is present on the board, it sets global variables fltype and flsize used for Flash programming based on the value returned by the function init_flash(). Parse_init() returns ERROR (-1) if it is called by the "df" command and Flash is not programmable. This function is implemented with Flash in "qt_dl.c", and without Flash in "evca_dl.c".

If Flash is not present on the board, this function should simply set the variable parser_state = P_INIT. This function is required for the download facilities to function properly.

FILE "xxx_data.c"

This file contains board specific data used by other routines to identify the hardware. The constant string, "BOARDNAME", must have the first three characters be the processor type in capital letters (KA, KB, MC, or CA) followed by a space. The second and last word of the string must be the board name. The is used in monitor.c for the sign on message, but will be parsed in future versions for other uses. There is also a constant in the file which is used to tune the xmodem packet timeout to about two seconds.

This file is essintial for resolving the constants it declares.

FILE "xxx_io.c"

FUNCTION buffco(unsigned char ch)

This function takes a character to be output as an argument. It checks the communication port first for an XOFF (CTRL-S) character waiting. If XOFF is found, it polls the communication port waiting until XON (CTRL-Q) is received before sending out the character. If no XOFF is found, it outputs the character immediately. This function is implemented in both "qt_io.c" and "evca_io.c".

This function is essential for character output to slow terminals (slow terminals often send XOFF characters to slow down input).

FUNCTION change_baud(int baud)

This function takes a new baud rate as an argument. It reprograms the UART to the new baud rate. This function is implemented in both "qt_io.c" and "evca_io.c".

As a stub, this function should return VOID and may print out a "not implemented" message. This function is the implementation of the "ba" command. This function is not implemented in "laser_io.c" since there is no serial port on the pc board and therefore baud rate has no meaning. All of the I/O for the iLASER III board is done through a shared memory window to the pc.

FUNCTION unsigned char ci()

This function polls the communication port waiting for a character to become available. It then returns the character.

This function is essential for character input, and is implemented for all boards.

FUNCTION co(unsigned char ch)

This function takes a character as an argument. It checks the communication port for a ready to transmit signal, then outputs the character. It does NOT check for XON-XOFF (this function is used for raw transmission).

This function is essential for binary character transfer, and is implemented for all boards.

FUNCTION csts()

This function checks the communication port status for an available character. It returns 0 if nothing is there, and returns non-zero if there is a character ready.

This function is used for character ready by the I/O functions, and is implemented for all boards.

FUNCTION init_console(int baud)

This function takes the default baud rate as an argument. It initializes the UART for Input/Output at the default (9600 for the standard NINDY) baud, or if there is no UART, sets up the port to which it will communicate (shared memory for the iLASERIII board). It also sets the global variables input_ptr = input_buffer for use by other I/O functions.

This function is called during initialization from the file "xxx_init.s". This function is implemented in one form or another on all boards.

FILE "xxx_boot.s"

This file has no executable code in it. It is the configuration file for a Cx board. It sets up the memory regions for the Cx, so it will be somewhat different for every board. See the *80960CA User's Manual* for information on how to configure it specifically for your board.

FILE "xxx_io.c" revisited:

The following functions are part of the file I/O routines. They require custom host software to communicate. The function for these routines is to set up and pass a structure containing the pertainent variables through to the host software, and return the correct information passed back from the host software. The host software then takes the command and structure, and implements the command on the host side. These functions merely serve as a channel for passing variables for the host software to do the work. These functions are implemented in the file "laser_io.c" and are tied to a custom host program which receives the structures and commands through the pc's shared memory.

All of these functions may be stubbed out by returning ERROR (-1). They are only used when custom host software is present to implement the functions. These functions are stubs in the files "qt_io.c" and "evca_io.c".

The flow of information from your application program through to the host software can be confusing, so the following example shows the steps taken by an fprintf() (printf() to a file) call. See Chapter 5, *NINDY Theory of Operation* for more information on system library calls. Figure 2.4 illustrates the call sequence.

1) Your program calls fprintf() - part of libcXX.a, the high level library.
2) fprintf() calls write() - part of libnin.a, the low level library.
3) IF the file is STDOUT or STDERR
   - write() calls buffco() through a system call to NINDY
   - buffco() writes to the console from NINDY on the board.
4) ELSE the file is on the host
   - write() calls f_write() through a system call to NINDY
   - f_write() talks to host software
   - host software writes to file

FUNCTION f_access()

FUNCTION f_close()

FUNCTION f_creat()

FUNCTION f_lseek()

FUNCTION f_open()

FUNCTION f_read()

FUNCTION f_systemd()

FUNCTION f_stat()

FUNCTION f_time()

FUNCTION f_unmask()

FUNCTION f_unlink()

FUNCTION f_write()

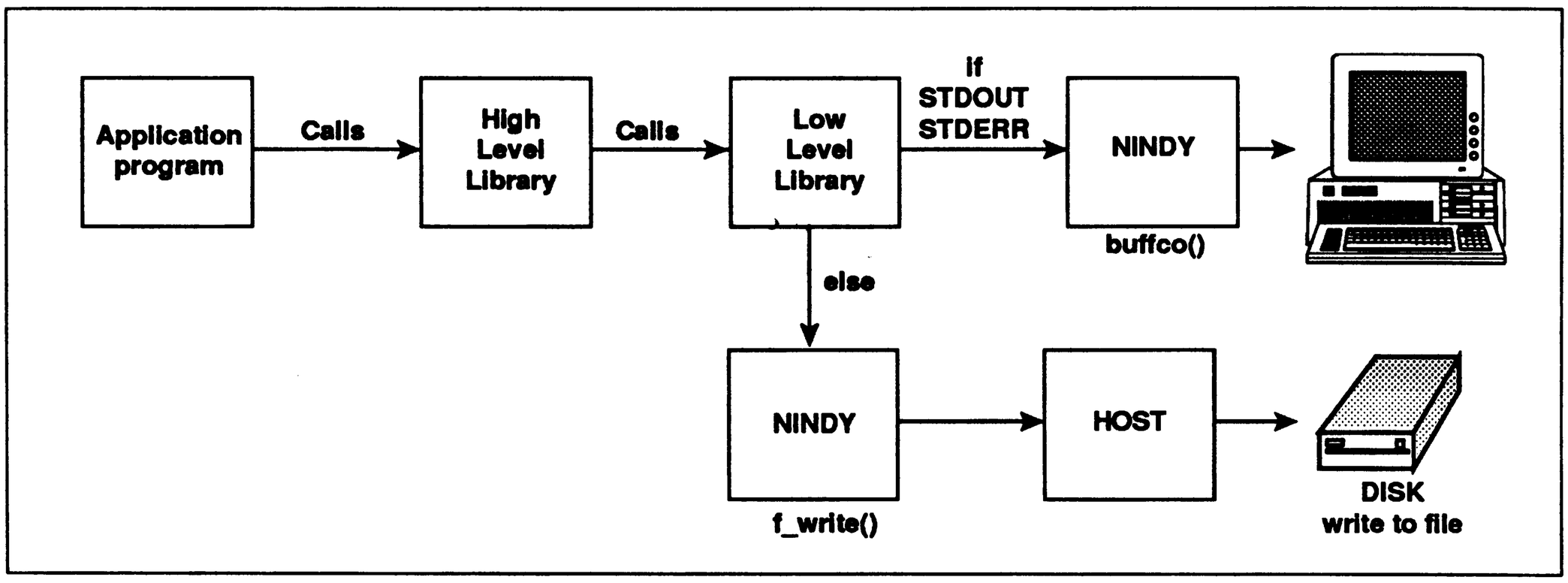


Figure 2.4 *— Call Sequence for I/O*

Chapter 3
Compilation

This section is a summary of how to compile and link files for burning NINDY into EPROM, downloading NINDY into Flash (for the QT960 board), downloading files to run on NINDY in RAM, and downloading files to run on NINDY in Flash (on the QT960 board). The tools used for this purpose are the DOS version 1.2-6 of the Intel 80960 C compiler (ic960) and Intel 80960 utilities (lnk960, asm960, rom960, arc960). Some differences may be found for other versions or other tools hosts.

Provided with the NINDY source are batch files for making the different pieces of a binary NINDY. QTMAKE.bat compiles the pieces needed for burning NINDY into ROMs on the QT960 board. FLMAKE.bat compiles the pieces needed for making NINDY downloadable to Flash on the QT960 board. MKLIB.bat creates the libraries connected with NINDY. MKTEST.bat compiles the test program "hello.c" for running out of RAM on the QT960 board, for downloading to and running out of Flash on the QT960 board, for running out of RAM on the QT960 board using a KA, and for running on the EV80960CA board for comparison. EVCAMAKE.bat is a batch file which makes a CA version of NINDY. There are two more batch files, INIT.bat and INSTALL.bat. INIT.bat copies the libraries and include files originally shipped with NINDY, to their respective places among the I960BASE tools. INIT.bat should be called first before running any of the batch files mentioned above (it should only need to be called once though). INSTALL.bat copies your newly made libraries and include files to their respective places among the I960BASE tools. INSTALL.bat should be called if MKLIB.bat is not used to create new libraries (MKLIB.bat does the work of INSTALL.bat automatically).

Compiling NINDY

To compile NINDY for running in ROMs on the QT960 board:

1) Change directories to the QT960 directory. The batch file, MAKE.bat is provided which compiles the files in the directory and archives them into a library, "qt.a" for linking. To compile them by hand, you would type:

```
ic960 -c -R -O2 -AKB -I..\include *.c *.s
arc960 -sr qt.a copymem.o flash.o qt_hw.o qt_dat.o qt_dl.o qt_io.o test_cal.o test_int.o test_asm.o tests.o test_tim.o
```

This command instructs the compiler to leave the files unlinked (-c), prepare the code for ROMs (-R), use optimization where possible (-O2), compile for the KB architecture (-AKB), and check the directory ..\INCLUDE for any include files not found in the current directory or in the I960BASE\INCLUDE directory. The archiver, arc960, then creates a file, "qt.a" from all of the object files created by the compiler. "qt.a" will be used later.

2) Next, change directories to ..\KX. A batch file is provided which compiles the files and creates a library "kx.a" for use later. To compile by hand, use the following lines:

```
ic960 -c -R -O2 -AKB -I..\include *.c *.s
arc960 -sr kx.a kx_break.o kx.o kx_ftbl.o
```

The switches and outcome are similar to the above descriptions for the QT960 directory.

3) Finally change directories to the base directory (..) again. We will not use the batch files in this directory since they would redo steps 1 and 2. To compile by hand, you type the following:

```
ic960 -c -R -O2 -AKB -Iinclude *.c *.s
```

There are a few assembly files which have pre-processor directives in them and need special consideration. These will not run straight through the compiler and come out with the correct results. "i_handle.s" is one of those files and must be re-compiled as follows:

```
copy i_handle.s i_handle.c
ic960 -E i_handle.c | asm960 -o i_handle.o
del i_handle.c
```

The assembly file must first be copied to a .c file extension to run through the pre-processing phase of the C compiler. When we are done, the C version of the file is removed. The -E option on the compiler sends the file through the C pre-processor and outputs to standard output. It is then piped to the assembler for assembly.

4) We now have the files compiled which are needed for NINDY. Unfortunately there are too many file names to fit on one link line. A file is provided, "basefile" which contains the names of all of the files common to every NINDY version. We do an incremental link to link these files together into one file so they will fit on the final link command line.

```
lnk960 -rF -o basefile.o basefile
```

The -rF switch tells the linker to link the files together, but keep the relocation information for later use. The -o switch renames the output file "basefile.o".

5) The final link state is next.

```
lnk960 -o qtrom lib\qtrom.ld kx\kx_init.o basefile.o fp.o float.o fp_flt.o kx\kx.a qt960\qt.a -lckb
```

The -o switch names the output file "qtrom". We are using "qtrom.ld" from the .\LIB directory (If you change "qtrom.ld" in LIBQT or CRT, be sure you use your new version here). We link in the initialization file, "kx_init.o", from the KX directory, the "basefile.o" file (created in step 4), the floating point files, "kx.a" (created in step 2) from the KX directory, "qt.a" (created in step 1) from the QT960 directory, and "libckb.a" (-lckb).

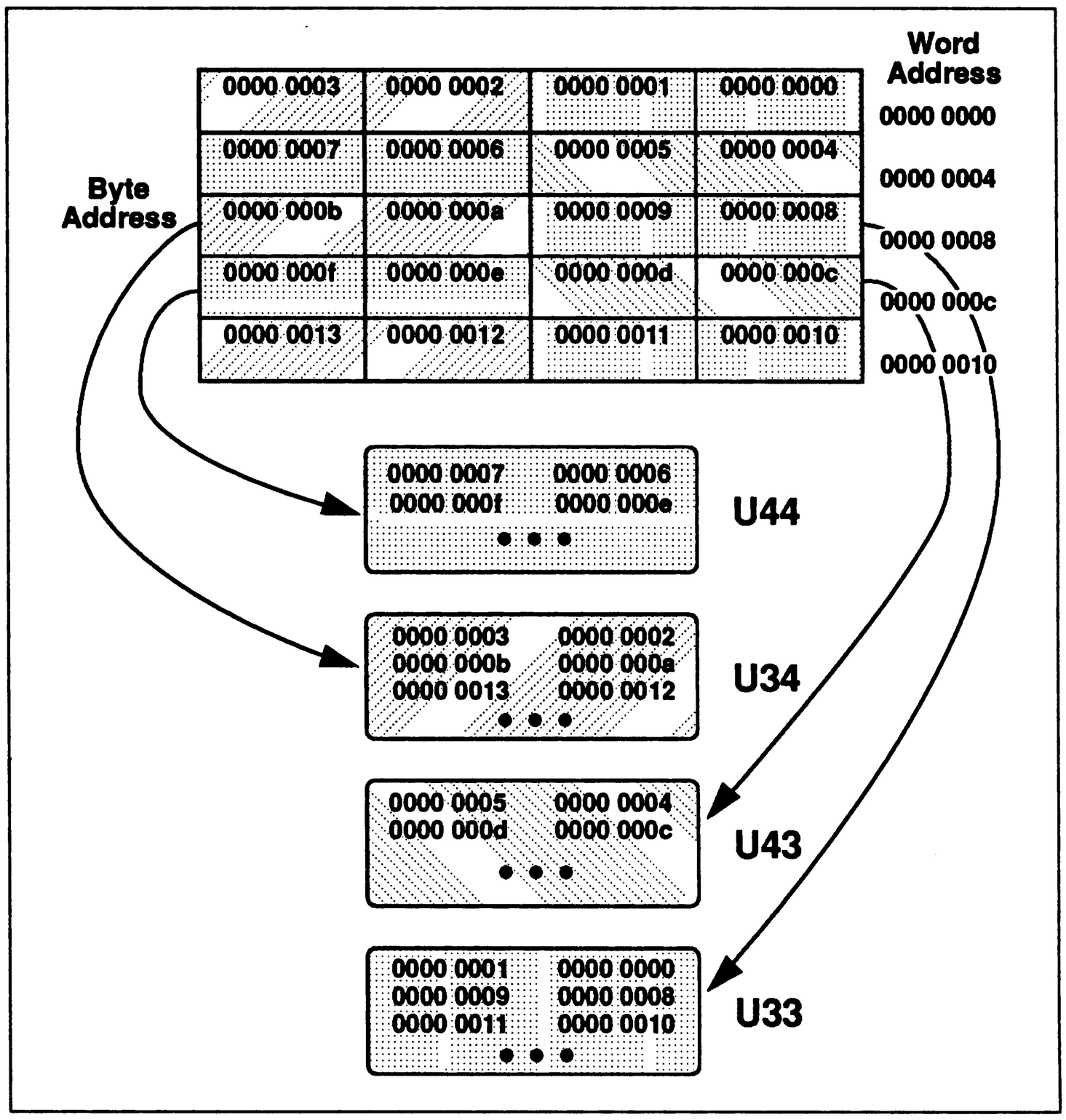


Figure 3.1 — *Interleaved EPROMs*

6) The "qtrom" file is then run through the rom960 utility:

```
rom960 .\lib\qtrom qtrom
```

"qtrom.ld" from the .\LIB directory will be used by rom960 for its commands (if you change this file in LIBQT or CRT, be sure you use your new version here). Rom960 produces several files. "image" is a raw image of NINDY without COFF headers. "out.00", "out.01", "out.02", and "out.03" are files produced when "image" is split into the four sections needed for the four ROMs. The files mentioned above are not needed again. They are temporary files used by the rom960 utility and may be removed. The files we do use are "qt0.hex", "qt1.hex", "qt2.hex", and "qt3.hex". These are the files in Intel HEX format which are used to burn the EPROMs.

7) Burn the "qt*.hex" files into EPROMs. The QT960 evaluation board uses four 27C202's (16 x 16 K) which are interleaved (see Figure 3.1). The data is copied into the EPROMs starting at address 0 with no offset. With the power off, place "qt0.hex" in socket U33, place "qt1.hex" in socket U33, place "qt2.hex" in socket U43, and place "qt3.hex" in socket U44. Be sure the jumpers in the lower right corner of the QT960 board are jumpered: J22-J23 in and J24-J25 in. The board will now boot from the new EPROMs upon powerup.

To compile NINDY for running in Flash on the QT960 board:

1) See step 1 above.

2) See step 2 above.

3) See step 3 above.

4) See step 4 above.

5) The final link state is next.

```
lnk960 -o flrom lib\flrom.ld kx\kx_init.o basefile.o fp.o float.o fp_flt.o kx\kx.a qt960\qt.a - lckb
```

The -o switch names the output file "flrom". We are using "flrom.ld" from the LIB directory (If you change "flrom.ld" in LIBQT or CRT, be sure you use your new version here). We link in the initialization file, "kx_init.s" from the KX directory, the "basefile.o" file (created in step 4), the floating point files, "kx.a" (created in step 2) from the KX directory, "qt.a" (created in step 1) from the QT960 directory, and "libckb.a" (-lckb).

6) The "flrom" file is then run through the rom960 utility:

```
rom960 .\lib\flrom flrom
```

"flrom.ld" from the .\LIB directory will be used by rom960 for its commands (if you change "flrom.ld" in LIBQT or CRT, be sure you use your new version here). This locates "flrom" at location 10000000_{16} and locates the data immediately after the text.

7) The "flrom" file is now ready to download to Flash. Turn on the QT960 board. It should be booting out of the EPROM space. Use the command "cf" to check the Flash space to see if it is blank. Use "ef" to erase the Flash space if it is not blank. Use the "df" command to download the "flrom" file to Flash. Turn off the power to the QT960 board. Set the jumpers in the lower right hand corner of the QT960 board to J23-J26 in and J22-J24 in. Upon powerup the board will be running out of the Flash.

To compile NINDY for running in the ROM on the EV80960CA board:

1) Change directories to the EVCA directory. The batch file, MAKE.bat is provided which compiles the files in the directory and archives them into a library, "evca.a" for linking. To compile them by hand, you would type:

```
ic960 -c -R -O2 -ACA -I..\include *.c *.s
arc960 -sr evca.a copymem.o evca_dat.o evca_dl.o evca_io.o evca_hw.o
```

This command instructs the compiler to leave the files unlinked (-c), prepare the code for ROMs (-R), use optimization where possible (-O2), compile for the CA architecture (-ACA), and check the directory ..\INCLUDE for any include files not found in the current directory or in the I960BASE\INCLUDE directory. The archiver, arc960, then creates a file, "evca.a" from all of the object files created by the compiler. "evca.a" will be used later.

2) Next, change directories to ..\CX. A batch file is provided which compiles the files and creates a library "cx.a" for use later. To compile by hand, use the following lines:

```
ic960 -c -R -O2 -ACA -I..\include *.c *.s
arc960 -sr cx.a cx_break.o cx.o cx_ftbl.o
```

The switches and outcome are similar to the above descriptions for the EVCA directory.

3) Finally change directories to the base directory (..) again. We will not use the batch files in this directory since they would redo steps 1 and 2. To compile by hand, you type the following:

```
ic960 -c -R -O2 -ACA -Iinclude *.c *.s
```

There are a few assembly files which have pre-processor directives in them and need special consideration. These will not run straight through the compiler and come out with the correct results. "i_handle.s" is one of those files and must be re-compiled as follows:

```
copy i_handle.s i_handle.c
ic960 -E -ACA i_handle.c | asm960 -ACA_A -o i_handle.o
del i_handle.c
```

The assembly file must first be copied to a .c file extension to run through the pre-processing phase of the C compiler. When we are done, the C version of the file is removed. The -E option on the compiler sends the file through the C pre- processor and outputs to standard output. It is then piped to the assembler for assembly.

4) We now have the files compiled which are needed for NINDY. Unfortunately there are too many file names to fit on one link line. A file is provided, "basefile" which contains the names of all of the files common to every NINDY version. We do an incremental link to link these files together into one file so they will fit on the final link command line.

```
lnk960 -rF -o basefile.o basefile
```

The -rF switch tells the linker to link the files together, but keep the relocation information for later use. The -o switch renames the output file "basefile.o".

5) The final link state is next.

```
lnk960 -o evcarom lib\evcarom.ld cx\cx_init.o basefile.o nofp.o nofp_flt.o cx\cx.a evca\evca.a-lcca
```

The -o switch names the output file "evcarom". We are using "evcarom.ld" from the .\LIB directory (If you change "evcarom.ld" in LIBEVCA or CRT, be sure you use your new version here). We link in the initialization file, "cx_init.o", from the CX directory, the "basefile.o" file (created in step 4), the floating point stub files, "cx.a" (created in step 2) from the CX directory, "evca.a" (created in step 1) from the EVCA directory, and "libcca.a" (-lcca).

6) The "evcarom" file is then run through the rom960 utility:

```
rom960 .\lib\evcarom evcarom
```

"evcarom.ld" from the .\LIB directory will be used by rom960 for its commands (if you change this file in LIBEVCA or CRT, be sure you use your new version here). Rom960 produces an intermediate file, "image", a raw image of NINDY without COFF headers. "image" is a temporary file used by the rom960 utility and may be removed. The file we do use is "evca.hex", a file in Intel HEX format which is used to burn the EPROM.

7) Burn the "evca.hex" file into an EPROM. The EV80960CA evaluation board uses one 27512 (8 x 16k). The data is copied into the EPROM starting at address 0 with no offset. With the power off, place "qt0.hex" in socket U33, place "qt1.hex" in socket U33, place "qt2.hex" in socket U43, and place the EPROM into the socket. The board will boot from the new EPROM upon powerup.

Compiling an Application Program for Running on NINDY

Make sure you have run the INIT.bat file to copy all of the LIB and CRT files to their respective places in the I960BASE directory.

Provided with the NINDY source is a file "hello.c" which is a basic "hello world" program with the benchmark timer turned on. The benchmark timing is included as an example of how to time programs (normally you would not be timing the I/O for accurate results). It is also included to illustrate how to compile using all of the NINDY libraries. If you can get a program which prints a message to compile and run, you have 90% of the pieces working together. If you get timing working also, you have about 99% of everything working together.

To compile "hello.c" for RAM using KB architecture on the QT960 board:

```
ic960 -Tqtnindy -AKB -O2 -o hello hello.c -lqt
```

This command line tells the compiler to use the "qtnindy.ld" file found in the I960BASE\LIB directory for linker directives (-Tqtnindy), to compile it for the KB architecture (-AKB -this is the default if omitted), to use optimizations where possible (-O2), to rename the output "hello" (-o hello), and to link in the library "libqt.a" found in the I960BASE\LIB directory (-lqt). If the benchmark timing is not desired, the -lqt switch may be omitted and the timing related code commented out. The "qtnindy.ld" file automatically links in "crtnin.o" and "libnin.a" from the I960BASE\LIB directory.

To compile "hello.c" for RAM using KA architecture on the QT960 board:

```
ic960 -Tqtnindy -AKA -O2 -o hello.ka hello.c -lqtka
```

This command line tells the compiler to use the "qtnindy.ld" file found in the I960BASE\LIB directory for linker directives (-Tqtnindy), to compile it for the KA architecture (-AKA), to use optimizations where possible (-O2), to rename the output "hello.ka" (-o hello.ka), and to link in the library "libqtka.a" found in the I960BASE\LIB directory (-lqtka). If the benchmark timing is not desired, the -lqtka switch may be omitted and the timing related code commented out. The "qtnindy.ld" file automatically links in "crtnin.o" and "libnin.a" from the I960BASE\LIB directory.

To compile "hello.c" for Flash on the QT960 board:

```
ic960 -Tflash -R -AKB -O2 -o hello.fla hello.c -lqt
rom960 flash hello.fla
```

The first command line tells the compiler to use the "flash.ld" file found in the I960BASE\LIB directory for linker directives (-Tflash), to compile it for the KB architecture (-AKB), to use optimizations where possible (-O2), to rename the output "hello.fla" (-o hello.fla), and to link in the library "libqt.a" found in the I960BASE\LIB directory (-lqt). If the benchmark timing is not desired, the -lqt switch may be omitted and the timing related code commented out. The "flash.ld" file automatically includes "crtfnin.o" and "libnin.a" from the I960BASE\LIB directory. The second command line invokes the rom960 utility to relocate the data section immediately following the text section for downloading to Flash. It instructs the rom960 utility to use the "flash.ld" file for commands which is found in the I960BASE\LIB directory.

To compile "hello.c" for RAM using CA architecture on the EV80960CA board:

```
ic960 -Tevnindy -ACA -O2 -o hello.ca hello.c -levca
```

This command line tells the compiler to use the "evnindy.ld" file found in the I960BASE\LIB directory for linker directives (-Tevnindy), to compile it for the CA architecture (-ACA), to use optimizations where possible (-O2), to rename the output "hello.ca" (-o hello.ca), and to link in the library "libevca.a" found in the I960BASE\LIB directory (-levca). If the benchmark timing is not desired, the -levca switch may be omitted and the timing related code commented out. The "evnindy.ld" file automatically links in "crtnin.o" and "libnin.a" from the I960BASE\LIB directory.

Chapter 4
Command Syntax

NINDY parses the command line using only the first two characters for a valid command. The exceptions to this are '?' for help and '.' for repeat. All addresses are in hexadecimal and are accepted with either a leading 0x or without it. The default number of times displayed is 1 if the [#<times>] option is not given.

The following are the available commands with an example of the prompt, command line, and any message which might follow the command.

da[sm] [<address>][#<words>] — Disassemble an instruction beginning at the address given. If no address is given, the current instruction pointer is used.

```
=>da 8000000#2
08000000 : 5c601e01 mov 1, r12
08000004 : 6563028c modpc r12, r12, r12
```

di[splay] <address>[#<words>] — Display a whole word beginning at the address given.

```
=>di 8000000#2
08000000 : 5c601e01
08000004 : 6563028c
```

di[splay] <register> — Display one register, where register can be any register.

```
=>di pfp
pfp : 0801c500
```

db <address>[#<bytes>] — Display one or more bytes beginning at the address given. Also displays bytes as ASCII characters if printable.

```
=>db 8000006#2
08000006 : 63 c
08000007 : 65 e
```

ds <address>[#<shorts>] — Display one or more shorts beginning at the address given.

```
=>ds 8000000#2
08000000 : 1e01
08000002 : 5c60
```

dd <address>[#<times>] — Display one or more sets of double words at the address given. Note that this command invokes a two-word burst fetch on the 80960 burst bus.

```
=>dd 8000000#2
08000000 : 5c601e01
08000004 : 6563028c
08000008 : 5ca81615
0800000c : 8c083000
```

dt <address>[#<times>] — Display one or more sets of triple words beginning at the address given. Note that this command invokes a three-word burst fetch on the 80960 burst bus.

```
=>dt 8000000#2
08000000 : 5c601e01
08000004 : 6563028c
08000008 : 5ca81615
0800000c : 8c083000
08000010 : 080064e8
08000014 : 8c603000
```

dq address[#times] — Display one or more sets of quad words beginning at the address given. Note that this command invokes a four-word burst fetch on the 80960 burst bus.

```
=>dq 8000000#2
08000000 : 5c601e01
08000004 : 6563028c
08000008 : 5ca81615
0800000c : 8c083000
08000010 : 080064e8
08000014 : 8c603000
08000018 : 00001000
0800001c : 5908408c
```

fr <address>[#<times>] — Display one or more real (32 bit) floating point numbers beginning at the address given.

```
=>fr 8000000#2
08000000 : 2.5233e 17
08000004 : 0.000000e 0
```

fl[oat] <address>[#<times>] — Display one or more long real (64 bit) floating point numbers beginning at the address given.

```
=>fl 8000000#2
08000000 : 2.46506565e 180
08000008 : -0.10557101e -249
```

fx <address>[#<times>] — Display one or more extended real (80 bit) floating point numbers beginning at the address given.

```
=>fx 8000000#2
08000000 : Reserved Operand
0800000c : Reserved Operand
```

ba[ud] <rate> — Change the current baud rate to the one given. Choices are: 1200 2400 9600 19200 38400

```
=>ba 19200
```

do[wnload] — Download using Xmodem. If downloading to Flash, this will automatically erase Flash first.

```
=>do
Downloading
Receiving file <— Invoke local download here.
                  Bring the host side into the
                  foreground running Xmodem and
                  download the file.
- Download complete -
Start address is :8000000
```

go [<address>] — Begins execution of program at address given. If no address is given the default is read from information in the downloaded file or the current IP is used. Also used as a continue from breakpoint.

```
=>go 10000000
```

st[ep] [<address>] — Single step through program. NOTE: the address of the breakpoint is the address of the instruction which has JUST FINISHED EXECUTING. The RIP and IP will contain the instruction to be executed.

```
=>st 8000000
*** Breakpoint at :08000000
Type(s) : Single-step
08000000 : 5c601e01 mov 1, r12
```

tr[ace] [br|ca|re|su [on|off]] —Turn on or off one of the following trace options: br[anch], ca[ll], re[turn] or su[pervisor]. If on/off is not given, the status of that option is displayed.

```
=>tr br on
Branch trace on
Call trace off
Return trace off
Supervisor trace off
```

br[eak] <address> —Set instruction break. If no address is given, all of the current instruction breakpoints are displayed. To delete an instruction breakpoint, use the 'de' command.

```
=>br 8000000
```

de[lete] <address> — Delete the specified instruction breakpoint.

```
=>de 8000000
No instruction breakpoints set
```

re[gisters] — Dump contents of registers.

```
=>re

g0 : 00000000 g1 : 00000000 g2 : 00000000 g3 : 00000000
g4 : 00000000 g5 : 00000000 g6 : 00000000 g7 : 00000000
g8 : 00000000 g9 : 00000000 g10: 00000000 g11: 00000000
g12: 00000000 g13: 00000000 g14: 00000000 fp : 0801c540

pfp: 0801c500 sp : 0801c580 rip: 08000004 r3 : 00000000
r4 : 00000000 r5 : 00000000 r6 : 00000000 r7 : 00000000
r8 : 00000000 r9 : 00000000 r10: 00000000 r11: 00000000
r12: 00000000 r13: 00000000 r14: 00000000 r15: 00000000
pc : 001f0003 ac : 3b001001 tc : 00000000 ip : 00005d00
fp0: 0.000000e 0
fp1: 0.000000e 0
fp2: 0.000000e 0
fp3: 0.000000e 0
```

mo[dify] <register> — Modify a register (eg. r4). Reg can NOT be floating point register. NOTE: be careful when modifying the fp, pfp, sp, pc, ac, or tc. ALSO NOTE: the register set is invalid until after the application program has begun execution.

```
->mo r10
r10 : 00000000 :
```

mo[dify] <address>[#<words>] — Modify one or more words in memory.

```
=>mo 801c0000
0801c000 : 00000002 :
0801c000 : 00000100 :
```

mb <address> — Modify a byte in memory (does not read byte first).

```
=>mb 2800000c
2800000c :
```

. — Repeat previous command.

```
=>di g0
g0 : 00000000
=>.
di g0
g0 : 00000000
```

cf — Check if Flash is blank. If not blank, prints out a message stating the first and last addresses at which it is programmed and the total size of the Flash EPROMs.

```
=>cf

Flash is programmed between 0x10000000 and 0x10005820

Size is 0x20000
```

df — Download to Flash EPROMs using Xmodem. Fatal error messages will be printed out if the Flash EPROM will no longer program. This option does NOT erase Flash first before downloading. This allows multiple programs to be fit into the Flash address space.

```
=>df

Downloading to Flash

Receiving file <— Invoke local download here.
                  Bring the host side into the
                  foreground running
                  Xmodem and download the file.

- Download complete -
Start address is :10000000
```

ef — Erase Flash EPROM. Error messages will be printed if the Flash will no longer erase.

```
=>ef

Programming zeros
Erasing Flash
```

ve[rsion] — Print out version header.

```
=>ve
    NINDY monitor for the 80960XX BOARD
         Version 2.06

    Copyright (c) 1989, 1990, Intel Corporation
```

rs — Resets the board.

```
=>rs
    NINDY monitor for the 80960XX BOARD
        Version 2.06

    Copyright (c) 1989, 1990, Intel Corporation
```

he[lp] [<command>]

? [<command>] — Gives help for optional 'command'. If no 'command' is specified or unknown 'command', print short version of all commands.

```
=>he rs

rs
```

— Resets the board.

fi[ll] <address1> <address2> <data> — Fill memory from address1 to address2 inclusive with word of data value. If address1 = address2 then one word is filled at address1

```
=>fi 8000000 800000c a5a5a5a5
```

bd [<address>] — Set data breakpoint at the address given. This only applies to 80960Cx versions which have hardware data breakpoints. If no address is given all of the current data breakpoints are displayed. To delete a data breakpoint use the 'ed' command.

```
=>bd 801c000
```

ed <address> — Erase the specified data breakpoint.

```
=>ed 801c000
    No data breakpoints set.
```

Chapter 5
NINDY Theory of Operation

This chapter attempts to explain the source code which is supplied, and why we implemented features as we did. The source code is intended to be an example of processor initialization, fault handling, stack switching, use of the 80960 debug features, interrupt handling, and general programming of the 80960. Each of these blocks will be covered in this chapter. Sometimes we implemented features in a more complex way than may be necessary. Our reasoning behind the chosen method, and an example of an alternate way to do things is provided.

System Initialization for the 80960Kx

The first thing you will want to to do is bring the processor up to a working state so that it can execute programs. Although you will probably want to do more, you may want to know what minimum initialization is required. The minimum requirements for the 80960Kx are as follows:

- 8 checksum words located at address 0_{16}
- System Address Table (SAT). This table is located on a 4 K byte boundary and is 176 bytes long. It is often located at address 0_{16} with the checksum embedded in its first 8 words.
- Processor Control Block (PRCB). This table is located on a quad word boundary and is 172 bytes long.
- System Procedure Table (SPT). This table is located on a quad word boundary and is a minimum of 48 bytes long.
- Supervisor Stack Pointer
- Interrupt Stack Pointer

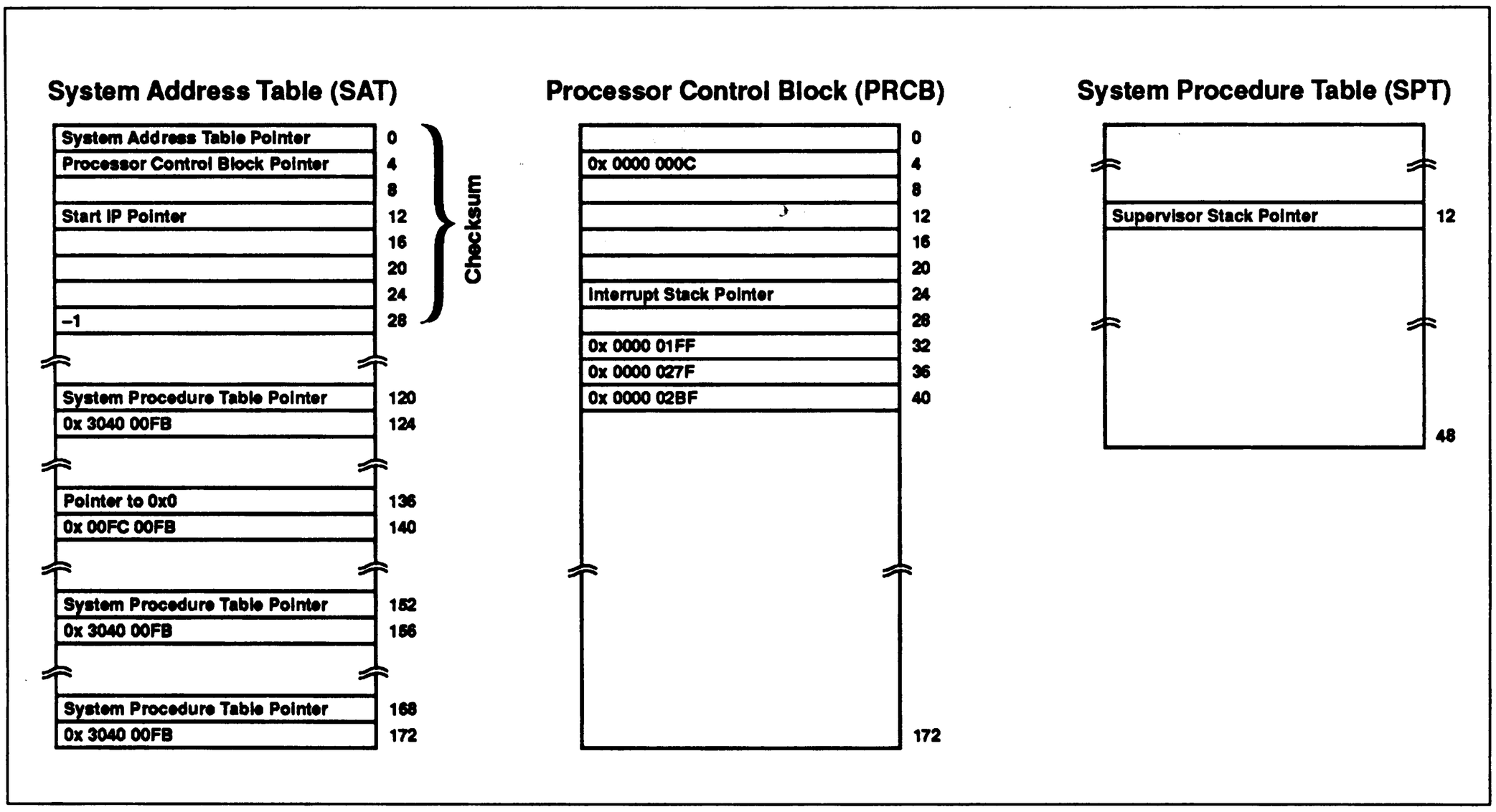


Figure 5.1 *— Minimum System Configuration for the 80960Kx*

The minimum configuration described above will bring the processor out of reset and execute code. You may locate all of the tables in the minimum configuration, in ROM. The minimum processor data structures do not supply legitimate fault and interrupt tables. Therefore interrupt or fault events will cause unpredictable behavior. To use this approach successfully, you must be absolutely certain that the processor will not be interrupted or it will try to access a nonexistent interrupt table trying to post the interrupts. A similar problem exists for faults. If the processor faults for any reason it will try to access a nonexistent fault table while trying to handle the fault. For further explanation of the contents of these initialization elements see the *80960KB Programmer's Reference Manual*, Chapter 7.

In order for you to fully utilize the features of the 80960, namely the ability to receive interrupts and faults without unpredictable results, several things must be added to the minimum configuration. The stacks must be located in RAM. An interrupt table must be added (and located in RAM if you want pending interrupts to be posted). A fault table (and optional fault procedure table) must be added (see following section on FAULTS). The processor (which starts up in an interrupted state) needs to be changed to executing state and restarted with the new PRCB in RAM. If your application program will be running in user mode, a user stack must also be added. NINDY's initializations for the 80960Kx are done in the "kx_init.s" and "copymem.s" files. NINDY's Kx interrupt table and fault table are located in the files "i_table.s" and "kx_ftbl.s" respectively.

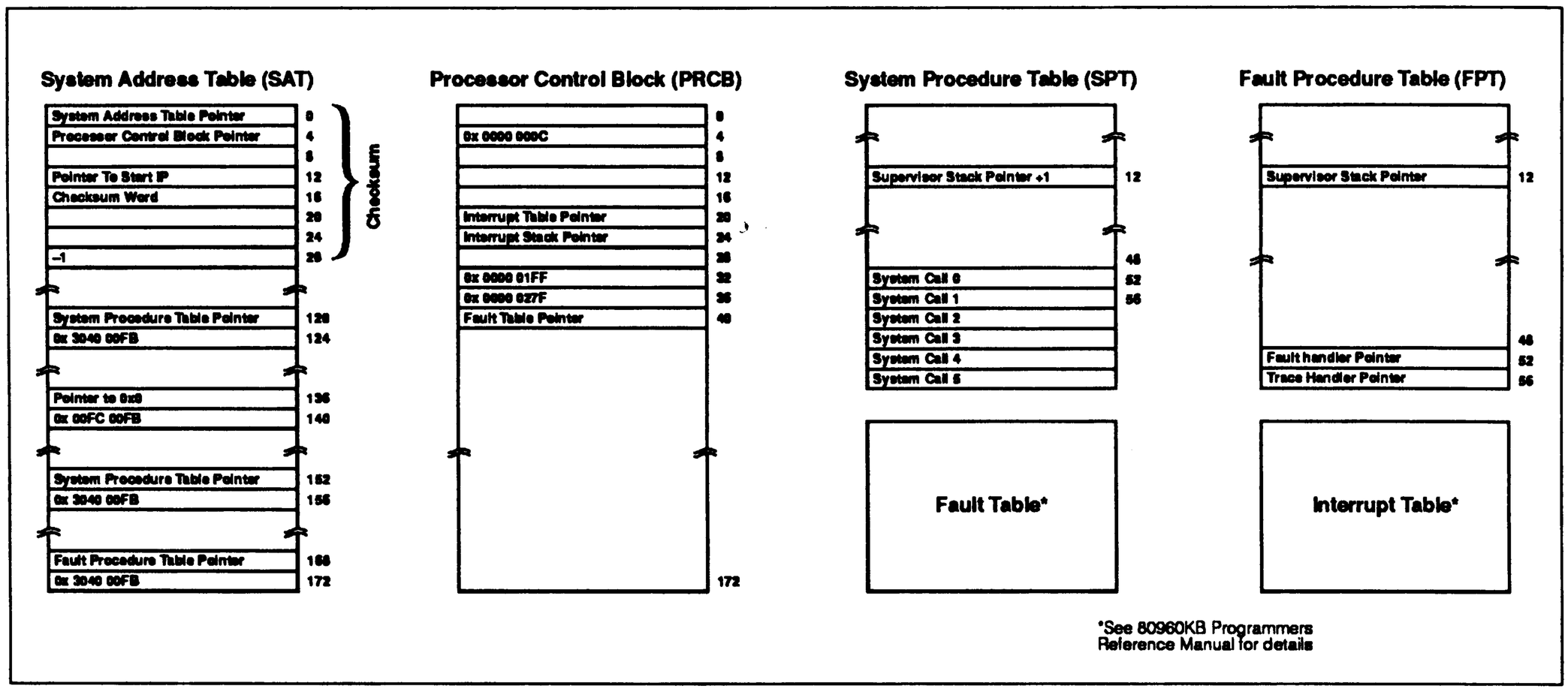


Figure 5.2 *— NINDY 80960Kx System Configuration*

The QT960 board begins execution from ROM on reset. The first thing which must be done when executing from ROM is to move the initialized data area (.data section) to RAM, so that any variables which will need to change as the program runs will actually change. NINDY's "copymem.s" file does the data copy.

The Processor Control Block (PRCB) contains information for transferring program control when the processor encounters a change of state (i.e. interrupt, fault, system call, etc). The contents of the PRCB are pointers to the other major data structures which the processor uses to handle these state changes. The PRCB is cached on chip at processor initialization, but needs to be located in RAM, to allow the processor to dump the resumption record there. The interrupt table also needs to be moved to RAM to allow posting of interrupts and substitution of the default interrupt handlers. NINDY uses a simple memory copy to move these structures to RAM, similar to the one it uses to copy the data section in "copymem.s". Once the new PRCB is in RAM, NINDY changes the interrupt table entry in the PRCB to reflect the fact that the interrupt table has a new address in RAM. The new interrupt table address is substituted for the old address in the PRCB. In a similar fashion, if you want to override the default fault handlers in your application program with local fault handler calls, you can move the fault table to RAM and replace its entry in the PRCB.

SAT

```
system_address_table:
    .word   system_address_table # SAT pointer
    .word   prcb_ptr             # PRCB pointer
    .word   0
    .word   start_ip             # Pointer to first IP
    .word   csl                  # calculated at link time
    .word   0                    # csl = -(segtab + PRCB + startup)
    .word   0
    .word   -1

    .space 88                    # reserve 88 bytes
    .word sys_proc_table         # initialization words
    .word 0x304000fb

    .space 8

    .word system_address_table
    .word 0x00fc00fb             # initialization words

    .space 8

    .word sys_proc_table         # initialization words
    .word 0x304000fb

    .space 8

    .word fault_proc_table       # initialization words
    .word 0x304000fb
```

SPT

```
    .align 6
sys_proc_table:
    .word   0                    # Reserved
    .word   0                    # Reserved
    .word   0                    # Reserved
    .word   (_trap_stack + 0x01) # Supervisor stack pointer
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   (_console_io + 0x2)  # console I/O routines
    .word   (_buffer_io + 0x2)   # file (buffered) I/O routines
    .word   (_lpt_io + 0x2)      # pc shared mem I/O routines
    .word   0                    # Preserved
    .word   0                    # Preserved
    .word   (_get_prob + 0x2)    # Return PRCB base
```

PRCB

```
#- initial PRCB
#
# - This is our startup PRCB. After initialization, this will
# - Be copied to RAM
    .align 6
prcb_ptr:
    .word   0x0          #  0 - reserved
    .word   0xc          #  4 - initialize to 0x0c
    .word   0x0          #  8 - reserved
    .word   0x0          # 12 - reserved
    .word   0x0          # 16 - reserved
    .word   intr_table   # 20 - interrupt table address
    .word   _intr_stack  # 24 - interrupt stack pointer
    .word   0x0          # 28 - reserved
    .word   0x000001ff   # 32 - pointer to offset zero
    .word   0x0000027f   # 36 - system procedure table pointer
    .word   fault_table  # 40 - fault table
    .word   0x0          # 44 - reserved
    .space 12            # 48 - reserved
    .word   0x0          # 60 - reserved
    .space 8             # 64 - reserved
    .word   0x0          # 72 - reserved
    .word   0x0          # 76 - reserved
    .space 48            # 80 - scratch space (resumption)
    .space 44            # 128 - scratch space ( error)
```

FPT

```
    .align 6
fault_proc_table:
    .word 0                                # Reserved
    .word 0                                # Reserved
    .word 0                                # Reserved
    .word _trap_stack                      # Supervisor stack pointer
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word 0                                # Preserved
    .word (_switch_stack_on_fault + 0x2)   # Fault Handler
    .word (_switch_stack_on_fault + 0x2)   # Trace Handler
```

Figure 5.3 *— Assembly Code for NINDY Kx System Configuration*

The processor continues to use the old PRCB for state changes until otherwise notified. To notify the processor of the new PRCB now located in RAM, NINDY issues an internal reinitialize IAC telling the processor to use the new PRCB. The reinitialize IAC is set up as follows:

```
reinitialize_iac:
.align 4
.word 0x93000000              # reinitialize IAC message
.word system_address_table    # address of system address table
.word prcb_ram                # use newly copied PRCB address
.word start_again_ip          # start at new address
```

The first word sets the IAC message to be a reinitialize IAC, the second word points to the address of the SAT (in case it is changed), the third word points to the new RAM based PRCB, and the last word points to the address at which to resume execution when the IAC is finished. The new start address instructs the processor where to begin execution after establishing its new structures.

NINDY issues the IAC with the following instructions:

```
iac:
lda 0xff000010, g5            # address for the IAC
lda reinitialize_iac, g6      # address of the IAC message
synmovq g5, g6                # send it off
```

For more information on IAC's see the *80960KB Programmer's Reference Manual*, Chapter 13.

Previously, the processor was running at priority 31 and interrupts were being "posted" to ROM, so all but priority 31 interrupts were essentially being masked off. Now that we have moved the interrupt table to RAM, lower priority interrupts are being posted for later use. If there are any possible interrupt sources on the target board you should disable or initialize them to a known state at this point. The 80960Kx processor comes out of reset in an interrupted state of priority 31, meaning that only interrupts of priority 31 will be serviced, and the processor runs off of the interrupt stack. To get out of the interrupted state, NINDY sets up a simulated interrupt record on the stack and does a return. The return from interrupt bit is set in the previous frame pointer (pfp). In the simulated interrupt record the desired process controls and arithmetic controls are set up. The return (from interrupt) loads the process control and arithmetic control registers from the interrupt record. The code segment for this process is shown below (taken from "kx_init.s"):

```
# Before call to main, we need to take the processor out
# of the "interrupted" state. In order to do this, we will
# execute a call statement, then "fix up" the stack frame
# to cause an interrupt return to be executed.
#

ldconst   64, g0
addo      sp, g0, sp # bump up stack to make room for
                     # simulated interrupt frame
call      fix_stack  # routine to turn off int state

# we have reserved area on the stack before the call to this
# routine. We need to build a phony interrupt record here
# to force the processor to pick it up on return. Also, we
# will take advantage of the fact that the processor will
# restore the pc and ac to its registers
#

fix_stack:
flushreg
or        pfp, 7, pfp        # put interrupt return code into pfp
ldconst   0x1f0002, g0
st        g0, -16(fp)        # store contrived pc
ldconst   0x3b001000, g0     # set up arith. controls
st        g0, -12(fp)        # store contrived ac
ret
```

The process controls are set so that NINDY is running at priority 31 with tracing turned off. We chose to run NINDY at priority 31 so that any unplanned interrupts would not get through. Tracing is turned off in NINDY, however it need not be (see section on FAULTS for explanation of why). The arithmetic controls are set up so that all of the floating point and integer fault masks are set. The fault masks allow the processor to continue executing when questionable arithmetic results are obtained. The burden of checking the arithmetic fault flags is then upon you. Another option is for you to leave the arithmetic faults unmasked and handle the resulting faults in fault handlers.

The initialization of the 80960Kx is complete. After you initialize any peripheral chips on the board, your application code can be started up. We leave NINDY running in supervisor mode on the supervisor stack. If you run your application in user mode, be sure there is a user stack on which the application can run.

System Initialization for the 80960Cx

The minimum requirements for the 80960Cx are as follows:

- 12 word Initialization Boot Record (IBR) located at address $ffffff00_{16}$.
- Processor Control Block (PRCB). This table is located on a quad word boundary and is 40 bytes long.
- System Procedure Table (SPT). This table is located on a word boundary and is a minimum of 48 bytes long.
- Control Table located on a quad word boundary. This table is 96 bytes long.
- Interrupt Table located on a word boundary.
- Interrupt Stack Pointer.
- Supervisor Stack Pointer.

The minimum configuration described above will bring the processor out of reset and execute code. The minimum configuration does not include a fault table, which you will need to add if the processor is going to fault for any reason. Unpredictable behavior will occur if the processor tries to access a nonexistant fault table while trying to handle a fault. For further explanation of the contents of these initialization elements see the *80960CA User's Manual.*

NINDY's initialization for the Cx is handled in the files "cx_init.s" and "copymem.s". The IBR and Control tables are specific to the hardware, but an example is found in "ev_boot.s". The interrupt and fault tables are found in "i_table.s" and "cx_ftbl.s" respectively.

The EV80960CA board begins execution from ROM on reset. The first thing which must be done when executing from ROM is to move the initialized data area (.data section) to RAM, so that any variables which will need to change as the program runs will actually change. NINDY's "copymem.s" file does the data copy.

The Processor Control Block (PRCB) contains information for transferring program control when the processor encounters a change of state (i.e. interrupt, fault, system call, etc). The contents of the PRCB are pointers to the other major data structures which the processor uses to handle these state changes. The PRCB is cached on chip at processor initialization, but needs to be located in RAM to allow changes to the structures. The interrupt table also needs to be moved to RAM to allow posting of interrupts and substitution of the default interrupt handlers. NINDY uses a simple memory copy to move these structures to RAM, similar to the one it uses to copy the data section in "copymem.s". Once the new PRCB is in RAM, NINDY changes the interrupt table entry in the PRCB to reflect the fact that the interrupt table has a new address in RAM. The new interrupt table address is substituted for the old address in the PRCB. In a similar fashion, if you want to override the default fault handlers in your application program with local fault handler calls, you can move the fault table to RAM and replace its entry in the PRCB.

The processor continues to use the old PRCB for state changes until otherwise notified. To notify the processor of the new PRCB now located in RAM, NINDY issues a reinitialize sysctl telling the processor to use the new PRCB. The reinitialize sysctl is issued as follows:

```
_reinit_sysctl:
ldconst   0x300, r4
ldconst   start_again_ip, r5
ldconst   _prcb_ram, r6
sysctl    r4, r5, r6
```

At boot time, the processor was running at priority 31 and we have all but the Non-Maskable Interrupt (NMI) masked. If there are any possible interrupt sources on the target board you should disable or initialize them to a known state at this point, and unmask the interrupts if desired. The 80960Cx processor comes out of reset in an interrupted state of priority 31, meaning that only interrupts of priority 31 or the NMI will be serviced, and the processor runs off of the interrupt stack. To get out of the interrupted state, NINDY sets up a simulated interrupt record on the stack and does a return. The return from interrupt bit is set in the previous frame pointer (pfp). In the simulated interrupt record the desired process controls and arithmetic controls are set up. The return (from interrupt) loads the process control and arithmetic control registers from the interrupt record. The code segment for this process is shown below (taken from "cx_init.s"):

```
# Before call to main, we need to take the processor out
# of the "interrupted" state. In order to do this, we will
# execute a call statement, then "fix up" the stack frame
# to cause an interrupt return to be executed.
#

ldconst   64, g0
addo      sp, g0, sp          # bump up stack to make room for
                              # simulated interrupt frame
call      fix_stack           # routine to turn off int state

# we have reserved area on the stack before the call to this
# routine. We need to build a phony interrupt record here
# to force the processor to pick it up on return. Also, we
# will take advantage of the fact that the processor will
# restore the pc and ac to its registers
#

fix_stack:
flushreg
or        pfp, 7, pfp         # put interrupt return code into pfp
ldconst   0xd81f0002, g0      # the upper bits are for the
                              # microcode interrupt return
st        g0, -16(fp)         # store contrived pc
ldconst   0x3b001000, g0      # set up arith. controls
st        g0, -12(fp)         # store contrived ac
ret
```

The process controls are set so that NINDY is running at priority 31 with tracing turned off. We chose to run NINDY at priority 31 so that any unplanned interrupts would not get through.

The initialization of the 80960Kx is complete. After you initialize any peripheral chips on the board, your application code can be started up. We leave NINDY running in supervisor mode on the supervisor stack. If you run your application in user mode, be sure there is a user stack on which the application can run.

Faults

Two main types of faults exist on the 80960, debug faults and generic "operation" faults. A generic fault is an error which occurs when the processor is executing instructions and comes across something illegal or exceptional. The other type of fault, the debug fault, is a planned fault such as what occurs when a breakpoint is reached which has been set. A debug fault also can occur if one of the tracing modes is set and the condition is met. A fault is the processor's way of halting execution. The processor internally handles a debug fault the same way as it handles a generic fault.

In order to handle faults, the processor must have access to a fault table. The fault table is referenced by the processor every time a fault occurs. The fault table has entries for either local fault handler routines, or supervisor fault handler routines. The supervisor fault handlers are accessed through the SAT which in turn looks up the correct supervisor procedure table that points to the fault handler routines. The fault table used by NINDY is found in "kx_ftbl.s" for the 80960Kx or "cx_ftbl.s" for the 80960Cx.

NINDY uses a fault procedure table (FPT) to reference the system fault handlers for the 80960Kx. The FPT has tracing turned off in the trace bit of the table. You may use the SPT for fault handler calls, however NINDY's SPT has the trace bit turned on to allow system calls to be traced. Tracing is fine for generic type faults, but for debug faults an infinite trace fault loop could occur. Given the scenario that tracing is turned on, the application which faulted has tracing turned on and the fault is a trace fault, the trace fault handler will be faulting infinitely through itself. NINDY uses a separate FPT to avoid this infinite trace fault scenario.

The 80960Cx handles faults slightly differently than the 80960Kx. The Cx was designed so that tracing is automatically turned off for all faults. Because of this, there is no need for the FPT. NINDY accesses its default fault handlers for the Cx through the SPT.

Local fault handlers directly referenced from the fault table are an alternative to the supervisor fault handlers described above. The advantage of using the supervisor route is it isolates the fault handler from the faulting application. For faults other than trace faults, isolation may be of no advantage to you, other than providing a user/supervisor stack switch. For trace faults, isolation is necessary.

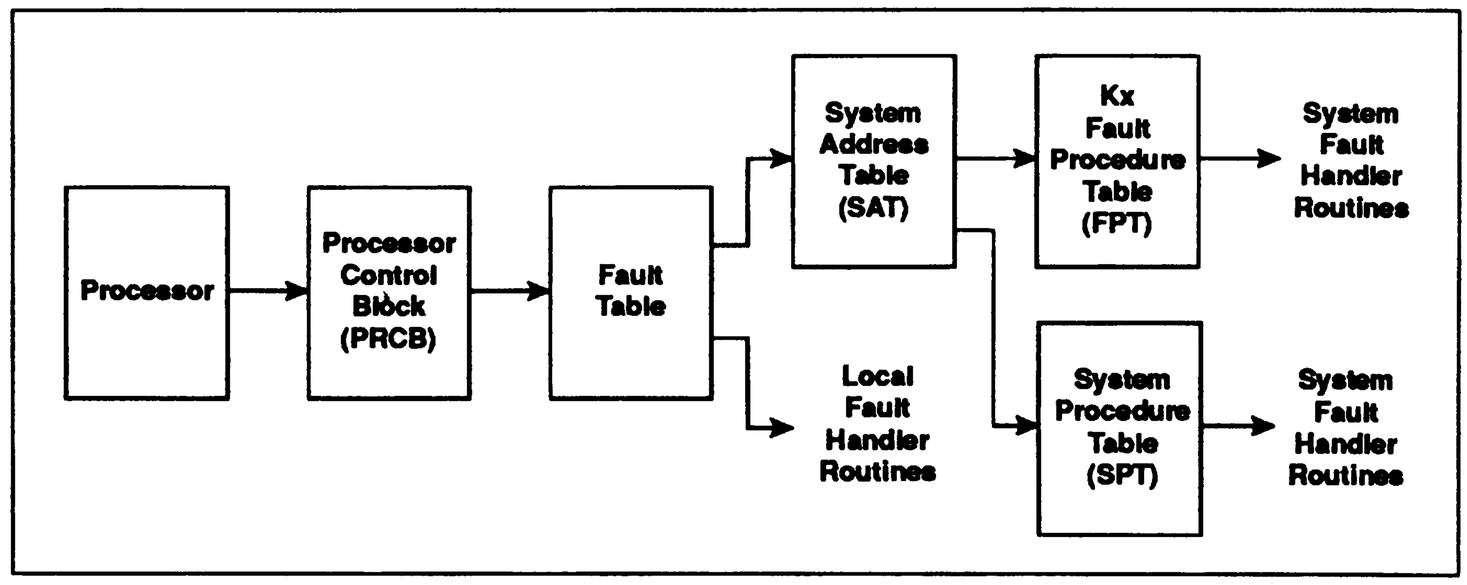


Figure 5.4 *— Fault Paths*

Supervisor fault handler Fault Table entry:

```
.word   2              # supervisor call, procedure 0
.word   0x2bf          # Fault Procedure Table (Kx only)
```

If you use a System Procedure Table for faults instead of the FPT:

```
.word   6              # supervisor call, procedure 1
.word   0x27f          # System Procedure Table
```

Local fault handler Fault Table entry:

```
.word   local_handler # local call, address of local handler
.word   0             # save this space, but it isn't used
```

Originally we turned off tracing in NINDY and used local fault handlers. This solved the infinite fault problem for tracing through NINDY for the Kx, but the problem was still there for any program running on top of NINDY with tracing turned on. We changed to using supervisor fault handler calls through the FPT and the trace fault problems were solved. When the Cx came along the problem was solved for us. Theoretically, tracing can now be turned on for all NINDYs to allow NINDY to debug itself without getting into an infinite trace fault loop. We chose to leave NINDY untraceable however.

Interrupts

The 80960Kx and Cx handle most of the overhead for interrupts in hardware. When the processor receives an interrupt the following actions are taken:

1) The stack is switched to the interrupt stack
2) A fresh set of local registers is allocated for the interrupt handler
3) The "state" in which the processor was running at the time of the interrupt is saved
4) The processor switches to supervisor mode for the interrupt handler
5) The interrupt handler is called.

The processor allows interrupt vectors 8 - 255, which translate to priority levels 1 - 31. For the 80960Cx, a Non-Maskable Interrupt (NMI) is also allowed. Priority 31 is the highest interrupt possible and is unmaskable for the Kx. For the Cx, all but the NMI may be masked. A special type of interrupt, an Interagent Communication (IAC), is a priority 31 interrupt on the Kx. Only interrupts which are a higher priority than the program currently being executed (or interrupt being serviced) will be serviced by the processor. All interrupts of lower or equal priority will be posted in the interrupt table by the processor for future servicing.

To simplify things, NINDY runs at priority 31 so only priority 31 interrupts (and NMI's for the Cx) will be serviced while the monitor is executing. The priority of your application running on top of NINDY determines which interrupts are serviced while your application is executing. You can set the priority in the Process Controls Register by using a modpc instruction. The following code segment sets the program priority to zero:

```
ldconst   0x001f0000, g0
ldconst   0, g1
modpc     g1, g0, g1
```

A protection fault will occur if the modpc is executed from user mode, so be sure your program is in supervisor mode when changing priority. NINDY runs applications in supervisor mode by default. For more information on how the 80960Kx handles interrupts internally see Chapter 8 of the *80960KB Programmer's Reference Manual*, or Chapter 6 of the *80960CA User's Manual*.

NINDY provides a default interrupt handler for each interrupt vector. The default handler simply executes a return. The processor automatically saves the local registers when entering the fault handler, however it is a good idea, especially if the fault handler calls another routine, to save the global registers also. You do not need to save the global registers if you know that they will not be used. The default interrupt handlers are found in "i_handle.s and int_rout.c", and the interrupt table which points to the handlers is found in "i_table.c".

When programming applications to run on NINDY, you may find it necessary to override one or more of the default interrupt handlers. The method for doing this is used in init_bentime(), found in "libqt\time_qt.c" or "libeva\time_ev.c". The timer bentime() uses a non-default interrupt handler to increment a value in memory every five milliseconds and init_bentime() sets this handler in place.

First, a system call locates the PRCB address. Since the PRCB points to the interrupt table, we find the address of the interrupt table from there. Once the interrupt table is found, we replace the correct entry with a pointer to the new interrupt handler routine. The following is the C code which replaces the default handler for entry 255 in the interrupt table with the timer interrupt routine timer_isr().

/* We need to located the interrupt table in order to enter our new vector into the table. We will do this by first finding the PRCB (Processor control block) base and then indexing into it to find the interrupt table. Once we have the interrupt table, we can index into it and put the vector into the appropriate spot. */

```
/* move PRCB to prcb pointer */
prcb = (unsigned int *) get_prcb ();

/* Interrupt table is here */
int_table = (unsigned int *) prcb [4];

/* set our timer vector */
int_table [256] = (unsigned int *) timer_isr;

** ** **

/* get_prcb is a system call 5 which simply returns
the base address of the PRCB */
        .globl _get_prcb
_get prcb:
        calls 5
        ret
```

Note that the int_table[] entry 256 is used to index vector 255. The interrupt table is set up such that an entry number does not correspond exactly to its vector number. Refer to the *80960KB Programmer's Reference Manual*, Chapter 8, or the *80960CA User's Manual*, Chapter 6, for more information on the interrupt table.

In a similar fashion, if you wanted to override a default fault handler, you could move the fault table to RAM and access it through the PRCB to replace a local entry in the fault table. This could be useful for writing specific application fault handlers.

Stacks

The 80960 recognizes three different types of stacks; the user stack, the supervisor stack, and the interrupt stack. NINDY uses two supervisor stacks; the nindy_stack and the trap_stack, and one interrupt stack, intr_stack. In addition to these three stacks, your program running on top of NINDY keeps its own stack space.

The processor automatically uses NINDY's interrupt stack out of reset, since it thinks it's servicing an interrupt. The interrupt stack is a supervisor mode stack since the processor automatically runs interrupt handlers in supervisor mode.

NINDY switches to the nindy_stack when it takes the processor out of its initial interrupted state (refer to section on SYSTEM INITIALIZATION). This is done at initialization by reloading the Frame Pointer and Stack Pointer as seen in this code segment from "xx_init.s".

```
lda     _nindy_stack, fp    # set up NINDY stack space
lda     -0x40(fp), pfp      # load pfp (just in case)
lda     0x40(fp), sp        # set up current stack ptr
```

The Previous Frame Pointer should not need to be loaded (it is hoped the previous frame will never be referenced), but the value is set just in case it is needed.

NINDY tries to run as much in its own stack space as possible. When a fault occurs the fault record and stack space for the fault handler are automatically put at the end of the faulting frame (see section on DEBUGGING). If the fault occurs in the application program (a breakpoint can cause this), the fault handler frame and subsequent NINDY debug frames will end up on the application's stack space. To keep NINDY running in its own space, NINDY switches the Stack Pointer (SP) immediately after reaching the fault handler. The SP now points to the trap_stack and the trap_stack takes over for the fault handler routine and subsequently called routines. The stack switch is done in a function called switch_stack_on_fault() found in both "fp_flt.s", and "nofp_flt.s".

The processor will be expecting to see the fault record immediately preceding the new fault handler frame on the trap_stack when it returns to running the application program on a return from fault. NINDY obligingly copies the fault record over to the trap_stack. The Previous Frame Pointer (PFP) of the new trap_stack frame is then pointed to the application program frame so that when the stack frames unwind on the return to the application program the old fault handler frame will be bypassed.

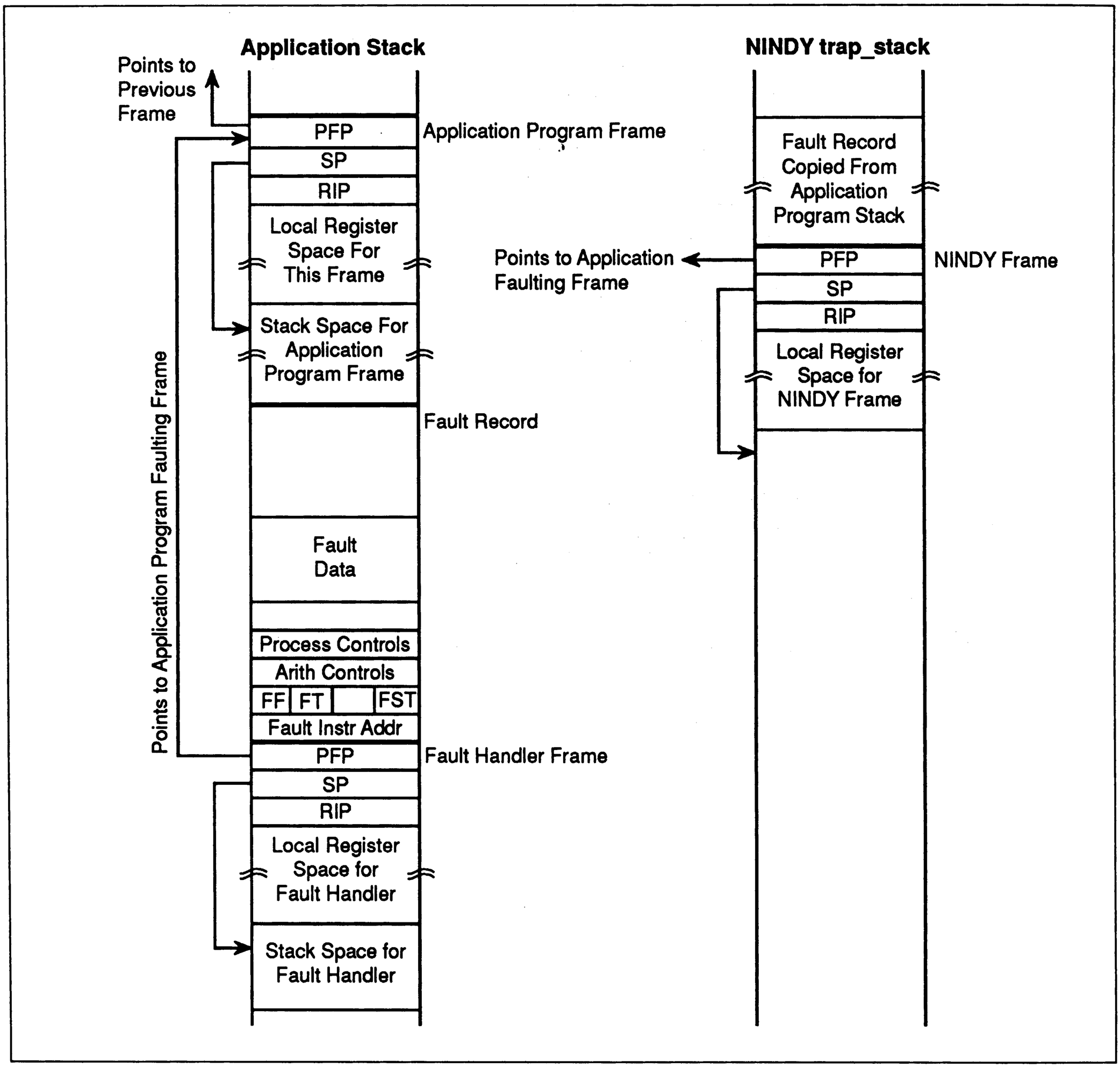


Figure 5.5 — *Stack Switch*

NINDY begins running after reset on the nindy_stack. To run an application program, NINDY switches the stack pointer to the application's stack space. This occurs before starting the application so that the application program always runs on its own stack space. When a breakpoint occurs, NINDY switches to the trap_stack to run NINDY's fault handlers and any debugging routines (display registers, disassemble memory, etc). Upon return to an application program, the trap_stack unwinds pointing back to the application's stack from which the application can now run. Exiting the application program essentially causes a fault in order to return to NINDY. Since any fault causes a stack switch to the trap_stack, even after the application program has completed, NINDY will continue to run off of the trap_stack space. Only a reset will bring NINDY back to the nindy_stack again.

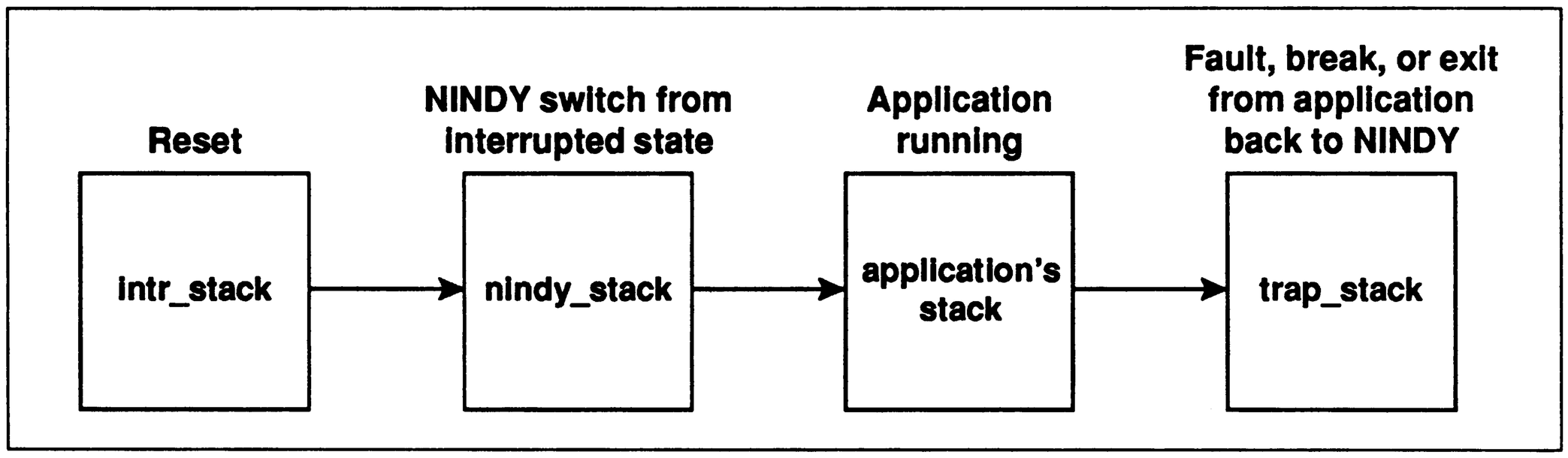


Figure 5.6 — *Stack Progression*

Execution

NINDY is designed to be transparent and non-intrusive to the user while running and debugging programs on top of it. When you are debugging a program under NINDY (setting breakpoints, displaying registers, displaying memory, single stepping), the processor actually goes back and forth between executing your application program and NINDY.

When your program is linked for running on NINDY, you link in three library files (belonging to NINDY) with your application program. One is "crtnin.o", one is "libqt.a" or "libevca.a", and the other is "libnin.a".

"libnin.a" is a library which provides the links between low level I/O routines in NINDY and the high level C library I/O calls like printf(). "libqt.a" is a library of hardware specific calls the user can make to benchmark timer routines, flash programming routines, and 82380 DMA routines when running on the QT960. "libevca.a" is a library of hardware specific routines the user can call for benchmark timer functions or other hardware functions on the EV80960CA. The "libnin.a" routines all result in systems calls through the SPT to routines in NINDY which handle the actual functions. The NINDY half of the functions are located in a file called "qt_io.c" or "evca_io.c". Calls which use STDIN, STDOUT, and STDERR are implemented in this file, however file I/O routines return ERROR (-1), and are implemented as stubs. You can find explanations of the library routines in Appendix B, C Library Calls.

The run time initialization for your program is provided by "crtnin.o". Your application program is sandwiched between a startup routine and an exit routine. The startup routine zeros the uninitialized data area (.bss section), enables tracing for the application program, sets the heap area, sets the stack pointer to the end of the heap and calls main() from your application program. For ROM based programs "crtfnin.o" performs these functions along with copying the data section (.data) to RAM. The exit routine catches the program when it "falls off the end" if exit() is not called from main(). When a program falls off the end it hits an fmark with a special word after it. This special word signals to the fault handler that this was not an ordinary fmark, and the program should finish gracefully by returning to the monitor.

The heap space has a default allocation in the library "libnin.a" of 1000_{16} bytes. The actual file in that library which does the allocation is called "heapsize.s". The heap is used by routines such as malloc() for dynamically allocating space for variable use. The default RAM setup for a program running on NINDY is shown in Figure 5.7.

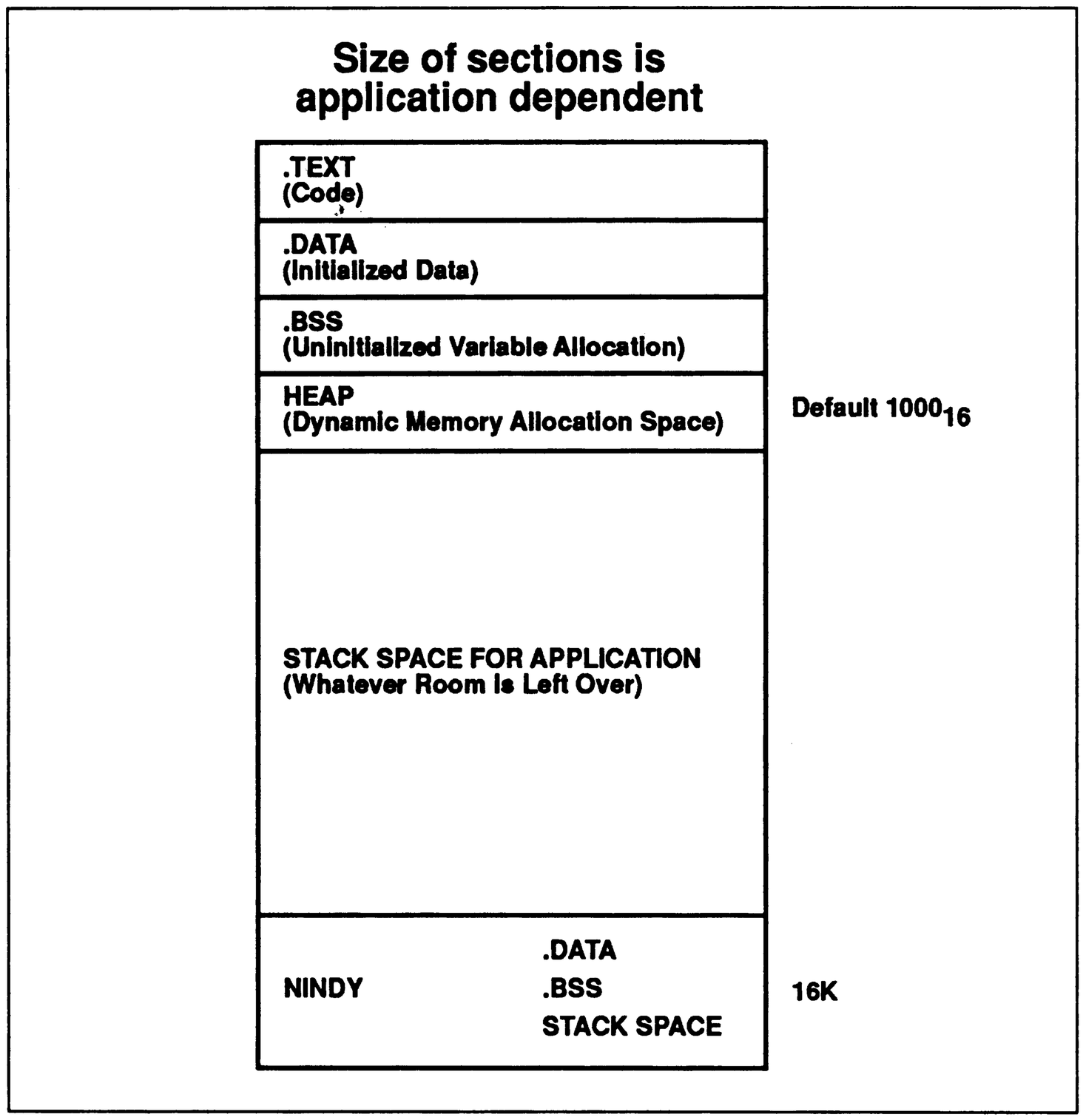


Figure 5.7 *— Memory Allocation for QT960 RAM*

If more than the default heap space is needed, two methods of overriding the default are possible. The first is to include a line of the form:

```
heap_size=XXXX;
```

on the first line of the linker specification ("qtnindy.ld" or other .ld file). You will get a linker warning, saying that the symbol "heap_size" is being redefined, but you can safely ignore it. The second method for overriding the default heap space allocation is for you to link in an assembler file containing the following two lines:

```
.globl    heap_size
.set      heap_size, XXXX
```

When you are debugging a program using NINDY and a breakpoint (or trace event) occurs, NINDY changes its prompt to signal that things beneath the surface are no longer as simple as you see them. Several things have happened. NINDY has switched to running off of its trap_stack (see section on STACKS). A flushreg from NINDY has caused the cached local register sets to be written to their respective stack frames, and NINDY has saved all of the registers away in memory so the current "state" is preserved.

A continue ("go" command) from the current instruction pointer tells NINDY to switch control back to your application program. This takes several steps which are transparent.. For NINDY to run your application program, it has to load all the registers which were saved in memory back into their correct locations, and start your application running with a return from fault. This fault return automatically switches the stacks back to your application stack (see section on STACKS in this chapter).

The I/O routines in "libnin.a" ("lib.c" and "libasm.s") use system calls to link user callable routines to NINDY routines. To illustrate a system call of an application program through the system procedure table, we will use an example program application(). The application program entry is set for a local call. The following SPT has entries for both supervisor and local programs:

```
sys_proc_table:
.word     0                          # Reserved
.word     0                          # Reserved
.word     0                          # Reserved
.word     (_trap_stack + 0x01)       # Supervisor stack pointer
.word     0                          # Preserved
.word     0                          # Preserved
.word     0                          # Preserved
.word     0                          # Preserved
.word     0                          # Preserved
.word     0                          # Preserved
.word     0                          # Preserved
.word     0                          # Preserved
.word     (_console_io + 0x2)        # calls 0
.word     (_buffer_io + 0x2)         # calls 1
.word     (_lpt_io + 0x2)            # calls 2
.word     (_switch_stack_on_fault + 0x2) # calls 3 (for the Cx)
.word     (_switch_stack_on_fault + 0x2) # calls 4 (for the Cx)
.word     (_get_prob + 0x2)          # calls 5
.word     _application               # calls 6 (no sup. bit set)
```

The first six system calls have been reserved for monitor compatability. They provide a uniform link between the application program and the board on which it is running. This allows a program linked with the "libnin.a" library to run on any target board running NINDY regardless of the target board configuration.

To use a system call for your own application, you would add it to the end of the SPT as has been done with application () above. Note that the supervisor bit has been set for the monitor routines, but application () has been left as a local call. You would then access the application by:

```
         calls       6       # assembly
or       asm ("calls 6");    /* C call */

instead of:
         call _application   # assembly
or       application();      /* C call */
```

For the record, console_io() is "calls 0", and buffer_io() is "calls 1". If the calling program is running in user mode, both console_io() and buffer_io() will switch to supervisor mode when executing (a stack switch to the supervisor stack indicated in the SPT will occur), but application() will continue to execute in user mode. If the calling program is running in supervisor mode, no change of state will occur for any of the system calls.

NINDY's file I/O calls pass arguments from the user program all the way down to NINDY through the system call to buffer_io(). By knowing the calling convention of the C compiler this is possible to do without actually passing the argument on every call. The Intel 80960 compiler passes arguments in global variables starting with g0. Knowing that the global variables are preserved between function calls, we can make function calls (such as the calls 1), without referencing the variables and still expect the variables to be in place when the final destination is reached (buffer_io()). We will use the file I/O call lseek() to illustrate how this works. Your application program calls lseek() which has three parameters, a file descriptor (fd), an offset (off), and an offset descriptor (od). The call would look like this:

```
lseek (fd, off, od);
```

where fd, off, and od have a valid value associated with them. This call is picked up by the library "libnin.a". The file "lib.c" containing the function lseek() links directly with your application program. lseek() calls a function files() with four parameters; the three parameters it was called with, plus an identifying parameter as the first parameter:

```
files (BS_LSEEK, fd, off, od);
```

files() is also in "libnin.a", but resides in the file "libasm.s", so "libasm.s" is now linked in with your application program. files() simply does a "calls 1". From the SPT above (a derivative of the "cx_init.s" SPT) you can see that a "calls 1" translates to a call to buffer_io(). buffer_io() expects up to 5 variables (it doesn't care if they exist as actual variables or not, it just loads in g0-g4). It uses the first variable, the identifying parameter which was stuck on the front of the files() call, to decide which NINDY function corresponds to the original library call. For the call lseek() the corresponding NINDY function is f_lseek(). The NINDY I/O functions are found in "qt_io.c". Any valid parameters which were passed in are then passed through to the NINDY function. buffer_io() looks something like this in C:

```
buffer_io (identifier, param1, param2, param3, param4)
int identifier;
int param1;
int param2;
int param3;
int param4;
{
switch (identifier) {
case BS_LSEEK:f_lseek(param1, param2, param3);
```

Param4 is not needed for lseek() and is a garbage value for this call, so is not passed on to f_lseek(). f_lseek() performs the desired task (usually signals a host function lseek(), or in the case of the QT960 and EV80960CA boards, just returns ERROR (-1)).

Your system call would not need to have as many levels of indirection as NINDY's I/O library calls have. A simple calls instruction from your application could get to your system call function directly. Because of NINDY's parameter passing and referencing several functions through one system call, its system calls are more complex.

Debug Methods

Debugging in an embedded control environment can be quite different than debugging in a reprogrammable environment. When you debug in a reprogrammable environment you use line numbers, symbolics and breakpoints on high level language statements to track errors. When debugging in an embedded control environment you use addresses and address maps of assembly language statements, registers and breakpoints on assembly instructions to track faults. NINDY has been designed to make debugging your program in the embedded environment easier. It uses features designed into the 80960 to trace through programs, and takes advantage of the trail which the processor leaves behind on a fault to inform you of what is happening.

Notice in Figure 5.8 there are three locations of trace bits on the Kx which turn on or off tracing and breakpoints. One is located in the Process Controls register, another is located in the Trace Controls register. There is also a bit in the SPT or FPT which controls tracing through system calls. For the Cx there is a fourth bit which must be turned on to allow tracing, a bit in the Fault Configuration Word of the PRCB.

Tracing is automatically turned off by the processor when servicing an interrupt, so it is never possible to trace through an interrupt handler. To debug an interrupt handler, you should call it explicitly from a program which has tracing turned on.

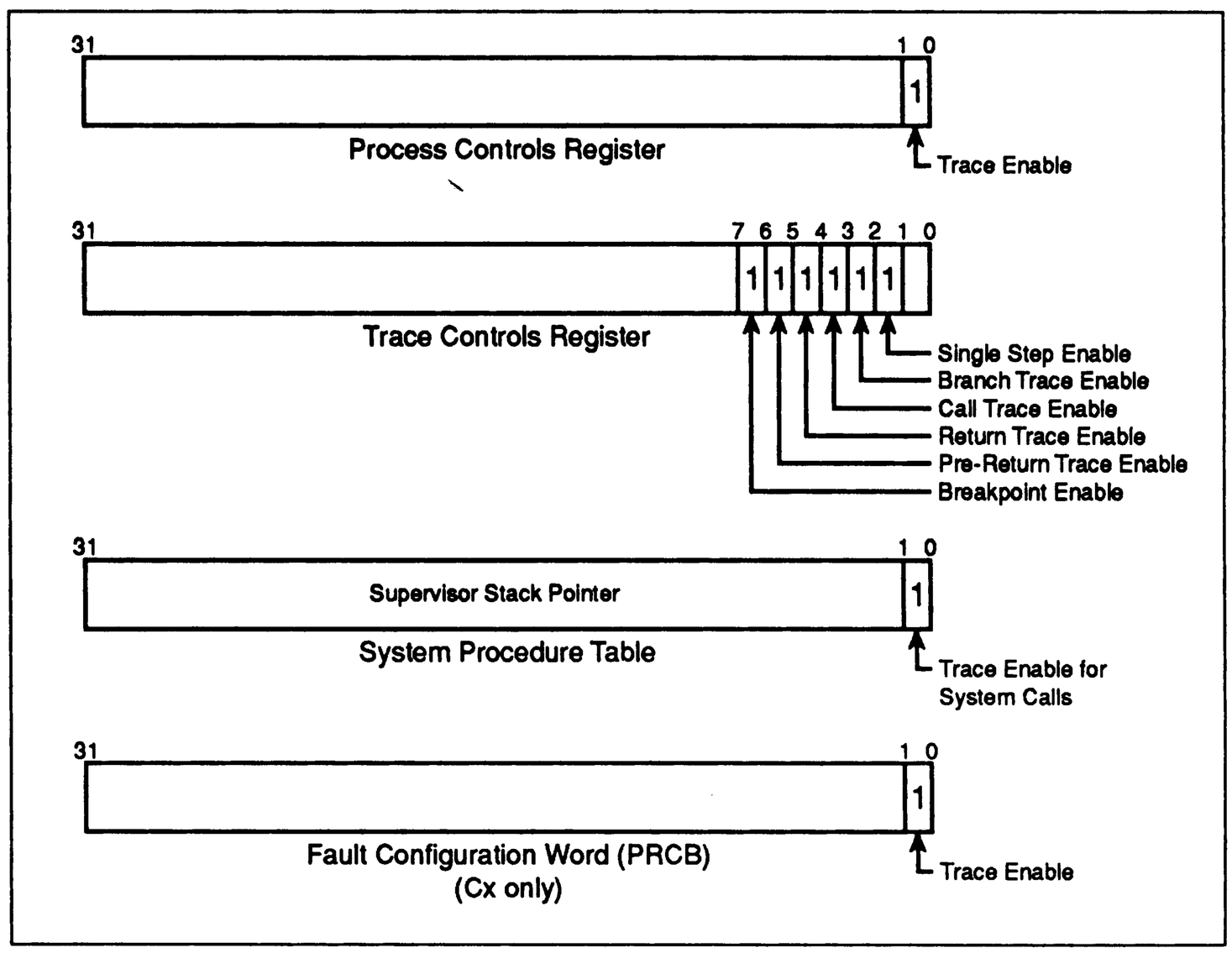


Figure 5.8 — *Trace Bit Locations*

Breakpoints

The most obvious debugging tool is the breakpoint. The 80960Kx allows two kinds of breakpoints, hardware and software instruction breakpoints. The 80960Cx allows hardware and software instruction breakpoints, as well as data breakpoints.

You can set a software breakpoint manually by replacing the instruction at which to break with a "mark" or "fmark" instruction. The mark instruction also requires the trace mode bit to be set in the trace controls register in order to break. The fmark instruction requires the process controls register trace bit to be on, but the trace controls register trace bit is not checked for the fmark instruction. Refer to Figure 5.9 for details on using software breakpoints.

Original code sequence

```
Address    Instruction    Disassembly
8000140 :  8C803000       lda 0x307,g0
8000144 :  00003070
8000148 :  090049F0       call 0x7A00
800014C :  59801901       subo 1, 0, g0
```

1. Replace instruction (first word of two word instruction) with fmark instruction.

```
Address    Instruction    Disassembly
8000140 :  66003e00       fmark
8000144 :  00003070       .word 0x00003070
8000148 :  090049F0       call 0x7A00
800014C :  59801901       subo 1, 0, g0
```

2. Run program.
3. When breakpoint occurs, replace fmark with original instruction (see original code sequence).
4. Subtract 4 from instruction pointer. (IP is at 80000144_{16} when breakpoint occurs, but needs to execute instruction at 8000140_{16}).
5. Single step through instruction.
6. When single step breakpointt occurs, replace code instruction with fmark instruction again.
7. Continue execution as desired.

Figure 5.9 *— Use and Recovery of Steps for Software Breakpoints*

To set hardware breakpoints, you must tell the processor your breakpoint address(es) using a set breakpoint register IAC on the Kx, or a load control register set 0 sysctl on the Cx. NINDY does this when either the "br" or the "de" (break or delete) command is executed. The breakpoint trace mode bit must also be set in the trace controls register, and for the Cx the trace bit must be set in the Fault Configuration Word of the PRCB. The IAC is issued as follows:

```
breakpt_iac:
.align 4
.word 0x8f000000        /* set up message type */
.word addr1             /* set up first breakpoint addr */
.word addr2 + 0x2       /* second breakpoint is disabled */
.word 0                 /* not used */

lda 0xff000010, g0      /* load local IAC address */
lda breakpt_iac, g1     /* load breakpt_iac message */
synmovq g0, g1          /* send IAC */
```

A breakpoint is set at addr1, but is disabled at addr2. If you want to delete both hardware breakpoints set up the set breakpoint register IAC with both breakpoint addresses disabled.

The load control register sysctl is issued as follows:

```
ldconst   _control_table, g4
ldconst   _addr1, g3
ldconst   _addr2, g2
st g3, (g4)             /* store address 1 at base of */
                        /* control table */
st g2, 4(g4)            /* store address 2 at second */
                        /* word of control table */
sysctl 0x400, 0, 0      /* issue load control register sysctl */
```

The advantages of using hardware breakpoints are that they can be set in ROM and require less recovery after the trace fault has occurred. Recovery from a hardware breakpoint is handled like any other trace event, but a software breakpoint requires replacing the fmark instruction with the original instruction word, backing the instruction pointer to the instruction, single stepping over the instruction, replacing the instruction with the fmark instruction again, and resuming execution. You should also note that a software breakpoint breaks BEFORE the marked instruction has executed (after the fmark has been executed) and a hardware breakpoint breaks AFTER the instruction has executed. Both types of breakpoints must have the trace bit(s) turned on (one in the process control register, and for the Cx, one in the Fault Configuration Word) to work.

Trace Events

The 80960 architecture allows six types of instruction traces in addition to the breakpoint trace. NINDY only uses five of these instruction trace types. The prereturn trace sets NINDY into a trace handling loop and is not implemented. The prereturn trace faults BEFORE the instruction is executed, whereas all of the other trace events fault AFTER the instruction is executed. NINDY's trace fault handler is set up to return to the instruction following the trace instruction. For a prereturn trace, the handler returns to the return instruction which has not yet executed, once again signaling a prereturn fault. In order to implement a "go" command with the prereturn trace, the trace fault handler would have to disable the prereturn trace after the first time it caused a fault, single step past the return instruction, reset the prereturn trace, and continue execution.

The instruction trace is better known as the single step. This trace mode breaks after every instruction executed.

The branch trace breaks every time a branch is taken. A conditional branch must take the branch for a branch trace to occur. Branch and link instructions do not cause a branch trace to occur.

Call traces are generated for any type of call or branch and link instruction. Note that the next instruction to be executed is the first instruction in the routine which was called.

Return traces are generated any time a return instruction is executed. No return trace is generated for a return from a branch and link.

The supervisor call trace will only work if your program is executing in user mode. NINDY by default leaves your program running in supervisor mode, so the processor does not recognize the supervisor call as a change of state making supervisor tracing appear not to work. If your program is running in user mode, supervisor tracing will work. A supervisor return to user mode will also trigger a supervisor trace event.

To enable any of these trace events, the Process Control register must have the trace bit set, and the Trace Control register must have the bit for the appropriate trace event set. Enabling and disabling the trace events is done by NINDY in the file "trace.c".

The trace fault handler in NINDY simply prints out which trace event(s) triggered the fault, and returns program control to NINDY. The NINDY debugging prompt appears to allow you to look at memory or registers and find out the state of the program at the time of the trace event.

Following the Trail

The 80960 leaves a fault record behind on the stack when a trace or generic fault occurs as shown in Figure 5.10. NINDY uses the fault record to print out the type of fault or trace event which has occurred, and the address for the next instruction to be executed after the fault (Return Instruction Pointer). NINDY's generic fault handler also prints out the beginning address of the fault record on the stack. The stack location is useful for tracing back through the stack frames to find out more about why and where the fault occurred. The fault record is always dumped on the current stack frame of the faulting instruction.

When a fault occurs, NINDY automatically flushes the local registers to their respective stack frames, and saves both the global and local registers from the faulting frame in a special register set area in memory (displayed by the NINDY command "re"). The register set area is actually two global variables named "register_set" and "fp_register_set" (for the KB). Flushing the registers insures that what is seen on the stack is valid. The local registers contain the previous stack frame pointer (PFP), the current stack pointer (SP), and the instruction to return to when that frame is done (RIP). By using the location of the fault record as being in the first stack frame, you can follow the stack frame links (PFP's) back to determine which routine contained the faulting instruction and from which routine it was called.

NINDY's trace and generic fault handlers are found in "fp_flt.s" or "nofp_flt.s", and "fault.c".

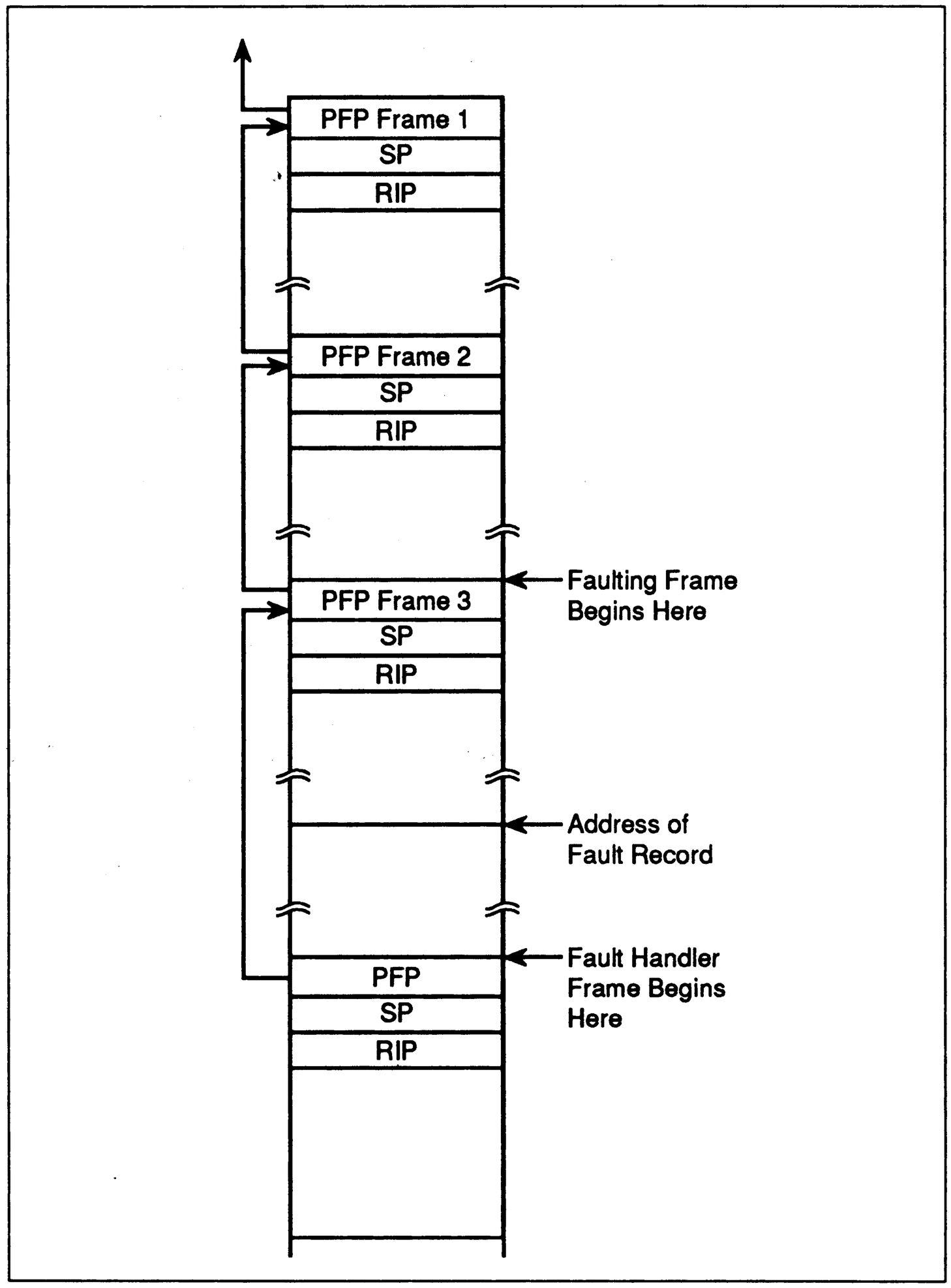


Figure 5.10 *— Stack Trail. Frame 3 is called from Frame 2. Frame 2 is called from Frame 1. RIP is the address of the instruction following the call.*

Monitor Features

NINDY as a debug monitor has several interesting features worth spending some time explaining. A few of these features are hardware dependent, but most are generic. These features will not be explained in great detail, but only as they apply to the NINDY monitor. See Figure 5.11 for a chart fo the files that form NINDY. All of the NINDY commands begin execution from the file "monitor.c".

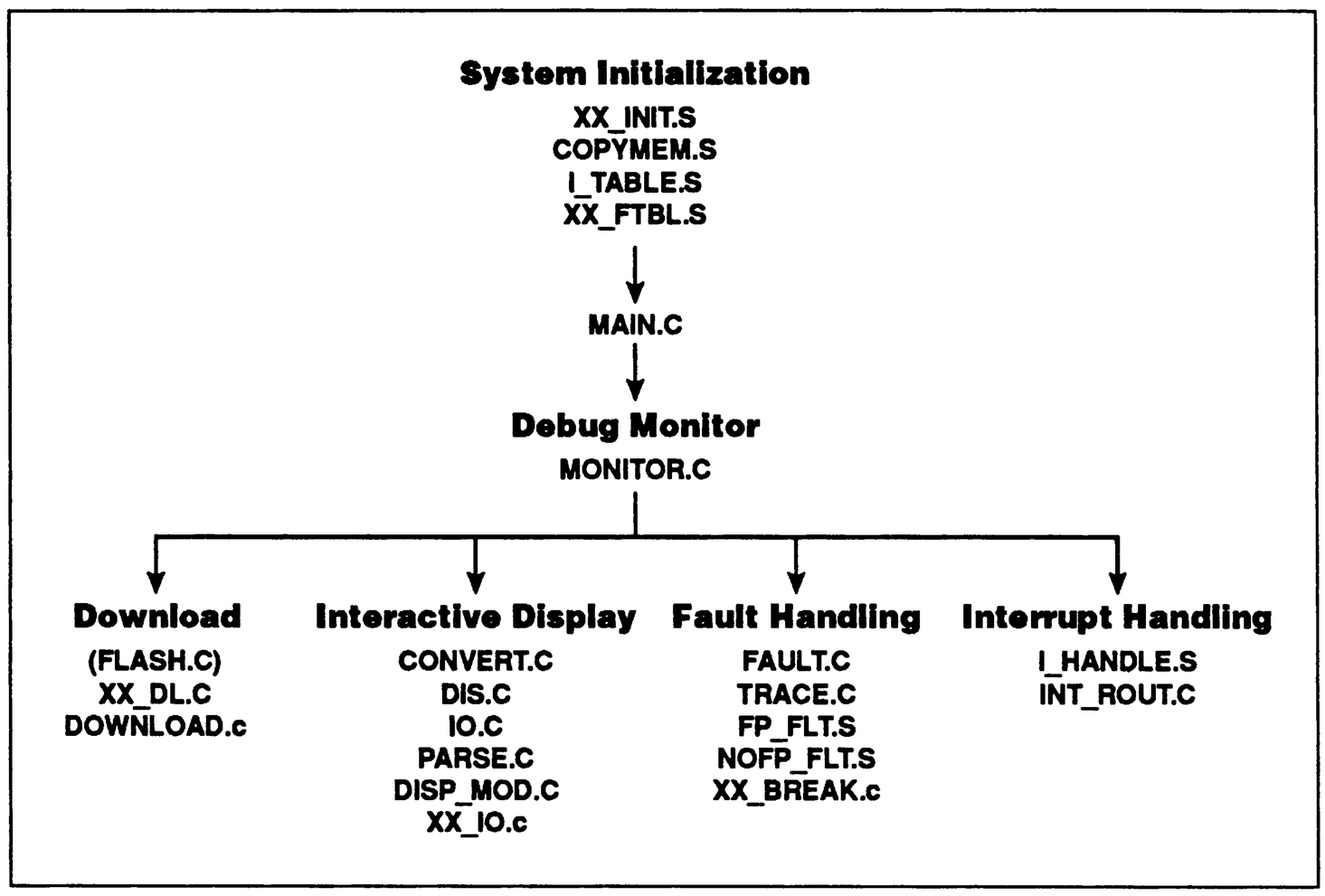


Figure 5.11 *— NINDY File Map*

Download

Data is transferred from the host terminal to the QT960 or EV80960CA board by means of the transfer protocol Xmodem. Xmodem sends packets of data (in our case 128 byte packets) across the RS-232 cable with a checksum at the end. See Figure 5.12.

NINDY is on the receiving end of the data transfer, stuffing the 128 byte packet into a buffer (global variable buff[]) shown in Figure 5.13. If the checksum at the end of the packet is correct on the receiving end, NINDY sends an acknowledge (ACK) signal back to the host terminal and processes the buffer. The transfer protocol is handled by NINDY in the file "download.c" by the function receive_xmodem().

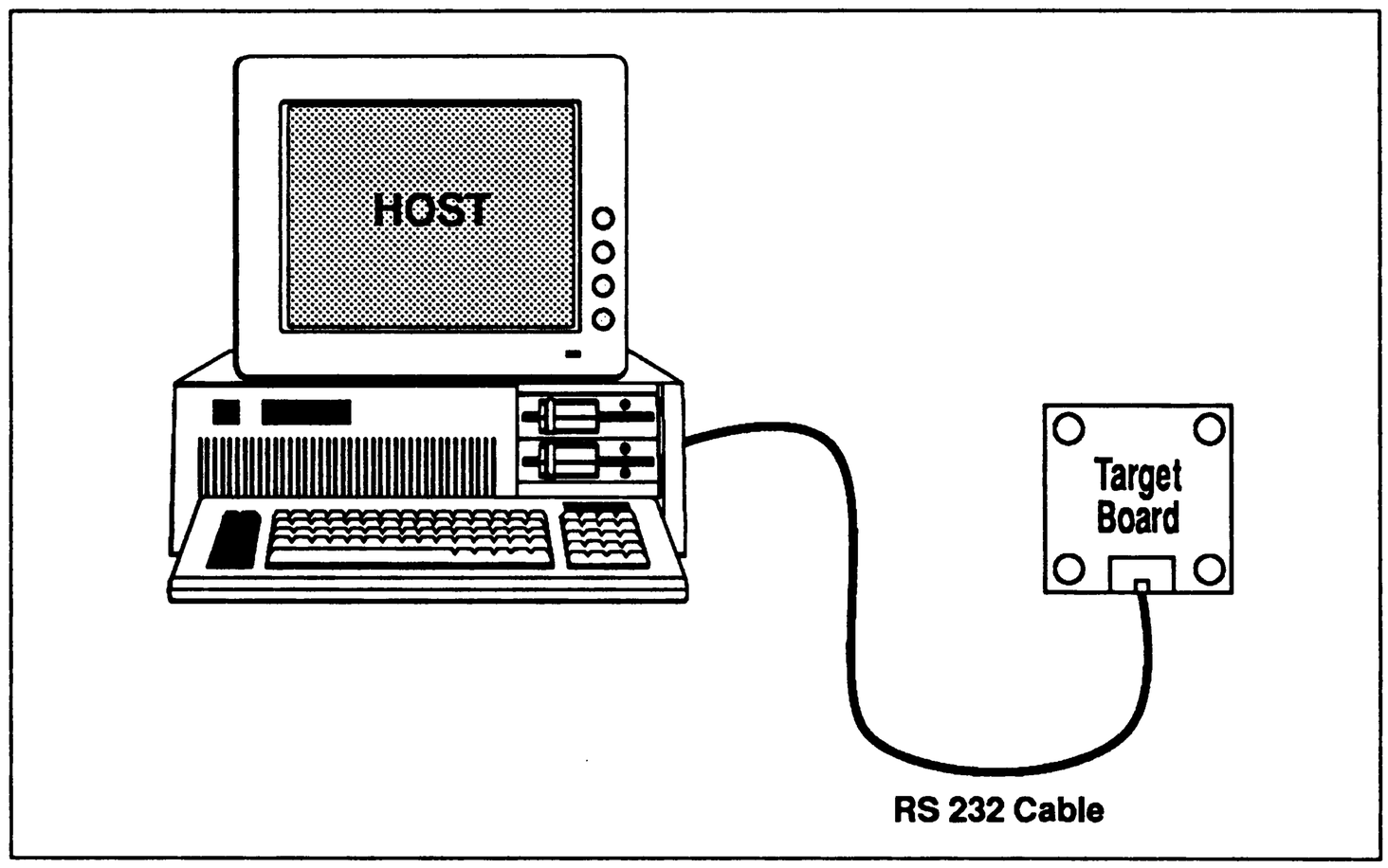


Figure 5.12 — *Host-cable-Target Board*

NINDY was written and tested using Intel's 80960 tools (compiler, assembler, linker, etc.) which produce executable files in Common Object File Format (COFF). The downloader in NINDY assumes the programs to be downloaded are in COFF format. The first packets are assumed to be the header information, and are loaded from the buffer into header structures which are later used to get sizes and addresses of the program sections. NINDY then loads the .text and .data sections into their correct addresses as determined by the header structure for each respective section. Several more packets may be downloaded, containing string information and other information which is not needed by NINDY. These packets are just discarded. The COFF format parser is found in the file "download.c" in the function parse_coff().

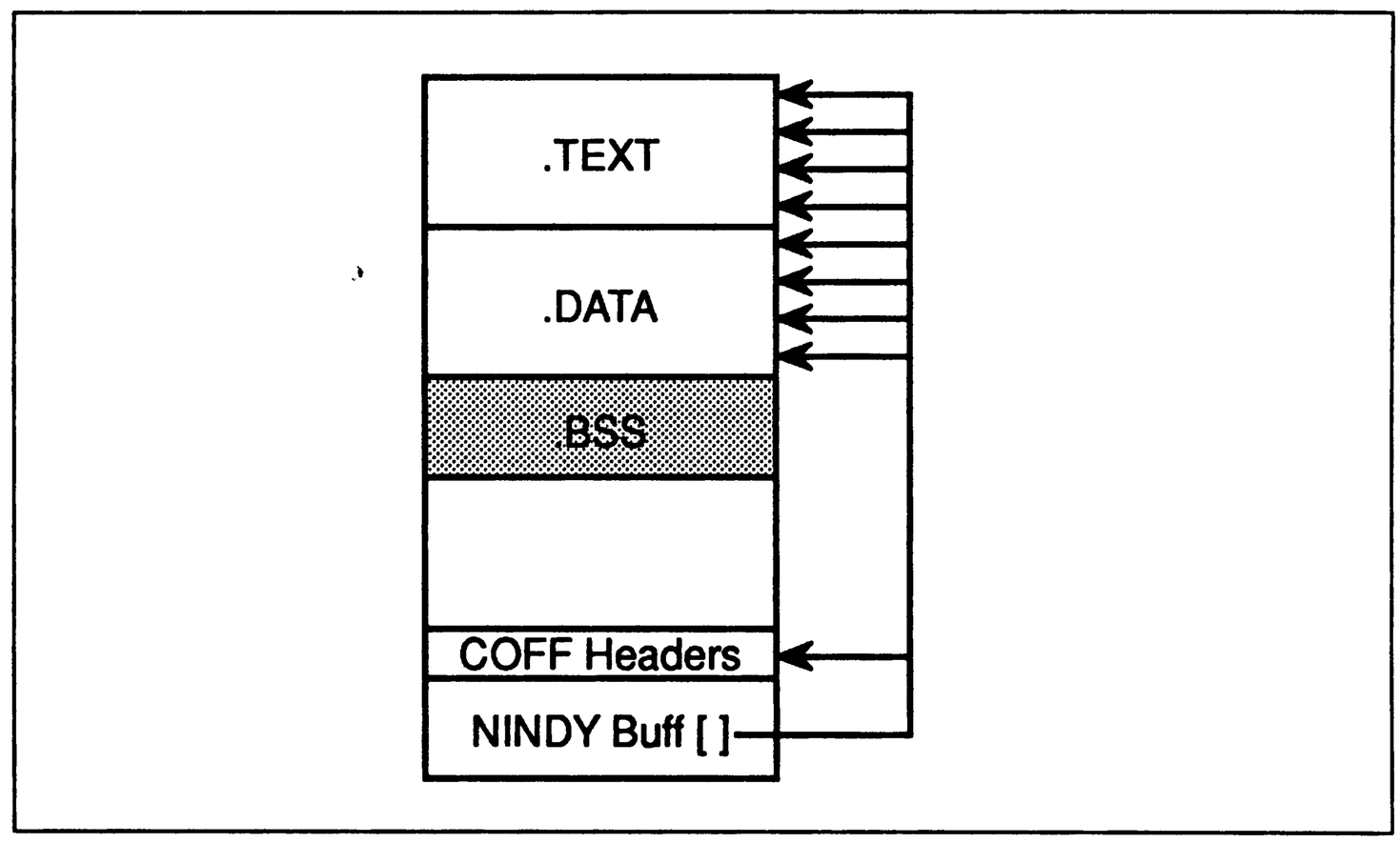


Figure 5.13 *— Downloading to Memory: Xmodem packet fills buff []. NINDY empties buff[] into the correct section in memory using information in the COFF header.*

Download to Flash (a QT960 board feature)

Another download feature is provided, which is a download to Flash EPROM. The Xmodem protocol is the same, but instead of downloading bytes directly to their respective locations, the bytes are burned into the Flash in their respective locations. The .text and .data sections are burned into the Flash EPROM instead of being placed in RAM. When a program is downloaded to Flash, it should be linked with the startup file "crtfnin.o". This startup file copies the .data section over to RAM before running the program.

The Flash will program as quickly as the data will transfer, and has been tested at up to 19,200 baud rate without the transfer having to wait for the Flash to program. Placing a program in Flash allows you to keep a program around after power off as if it were in EPROM, but erase it and reprogram it quickly as if it were in RAM.

The actual Flash burning program is found in "flash.c" in a function called download_flash(). It uses a parallel Flash programming algorithm, programming all four banks of Flash simultaneously (the first such algorithm of its kind, I might add). The Flash space must be blank before downloading (or the download will fail), so there is a function called erase_flash() which will erase the Flash space (the NINDY command "ef" calls this function). The erasing of Flash is not an automatic event when downloading using the "df" command. This enables several programs to be downloaded to Flash at different locations. Flash is automatically erased when the "do" command is used however. For further information on Flash EPROM programming see AP-316, *Using the 28F256 Flash Memory for In-System Reprogrammable Nonvolatile Storage.*

Memory Display

NINDY displays memory in several forms using several features built into the 80960 for display of these forms. The display quad words ("dq"), display triple words ("dt"), display double words ("dd"), display word ("di"), display short ("ds"), and display byte ("db") all use different 80960 load instructions to obtain the desired memory form. The ldq, ldt, ldl, ld, ldos, and ldob instructions are used respectively. To display a memory location as a floating point number (KB versions only), the "number" is first sent through an 80960 instruction, classr or classrl depending on the size of the floating point number desired, which classifies the "number" as to its type (denormalized, normal finite, infinity, zero, or not a number).

NINDY uses its own print function print() for strings, a function called out_hex() to print hexidecimal numbers, and a function called out_fp() which prints out floating point numbers to the terminal (KB only). These functions are found in "io.c". The alternative was for us to use printf() (the standard print routine for C), however printf() calls many other functions to do its job causing a very large set of otherwise unneeded library functions to be linked in. Since NINDY is a monitor program, we felt it should leave as much room as possible for an application to run. We decided we could get by without printf() inside of NINDY. Your application program running on top of NINDY cannot access NINDY's print routines, so must use printf() or other standard C routines for output.

Appendix A
NINDY File Descriptions

Common includes between versions

include\block_io.h — structures and defines for block I/O in NINDY
include\bus.h — defines for the Cx bus configuration
include\defines.h — #defines used throughout NINDY
include\globals.h — global variables used throughout NINDY
include\iac.h — description of the IAC structure
include\qtcommon.h — shared information between NINDY and libqt.a
include\regs.h — #defines for the application registers

Files common to all versions

convert.c — data manipulation routines (eg. ASCII to hex conversions)
dis.c — code for the disassembler
disp_mod.c — memory manipulation routines
download.c — COFF parsing and Xmodem protocol routines
eat_time.s — wait loop used to slow down access to peripherals
errno.c — added only for linking against ic960 V2.0 libraries
fault.c — fault handler routines
float.s — floating point assembly routines
fp.c — floating point dependent functions
fp_flt.s — fault routines which include floating point registers
gdb.c — support for GDB mode commands
i_handle.s — default interrupt handler routines
i_table.s — initial memory image interrupt table
int_rout.c — C routine called from the interrupt handler
io.c — Input/Output routines for NINDY
ldstore.s — assembly routines providing multiple word load/store
lookup.h — lookup table for disassembly routines
main.c — calls the initialization routines and starts the monitor
monitor.c — monitor command routines

no_fp.c	— stubs for floating point dependent functions
nofp_flt.s	— fault routines which exclude floating point registers
parse.c	— command line parser for monitor
trace.c	— trace fault handling routines
version.h	— version file for the NINDY monitor

Files common to any Kx version

kx\kx_break.c	— breakpoint specific functions for the Kx
kx\kx.s	— Kx dependent assembly code
kx\kx_ftbl.s	— fault table for the Kx
kx\kx_init.s	— system initialization file for the Kx

Files common to any Cx version

cx\cx_break.c	— breakpoint functions for the Cx
cx\cx.s	— assembly routines specific to the Cx
cx\cx_ftbl.s	— fault table for the Cx
cx\cx_init.s	— initialization code for the Cx

Files specific to the QT960 version

qt960\copymem.s	— copies variables placed in ROM to RAM
qt960\flash.c	— Flash EPROM read/write routines
qt960\qt_hw.c	— hardware dependent routines for the QT960 board
qt960\qt_data.c	— board specific data for the QT960 board
qt960\qt_dl.c	— download file for the QT960 board
qt960\qt_io.c	— low level I/O routines (board specific)
qt960\qt510.h	— initialization values for the 82510 serial chip

Files specific to the QT960 boot tests

qt960\test_cal.c	— tests reset flags to see if boot tests should be run
qt960\test_int.c	— interrupt tests for the QT960 boot tests
qt960\test_asm.s	— assembly functions needed for the boot tests
qt960\test_def.h	— #defines for the QT960 boot tests
qt960\tests.c	— suite of QT960 boot tests
qt960\test_tim.c	— verifies timers for the QT960 boot tests

Files specific to the EV80960CA version

evca\copymem.s — copies variables placed in ROM to RAM
evca\ev_boot.s — boot record for the EVCA board
evca\evca_dat.c — board specific data for the EVCA board
evca\evca_dl.c — download file for the EVCA board
evca\evca_io.c — low level I/O routines (board specific)
evca\evca510.h — initialization values for the 82510 serial chip
evca\evca_hw.c — hardware dependent routines for the EVCA board

Files common to all NINDY library versions

libnin\copyrght.c — copyright notice for inclusion with libnin.a
libnin\dto_stub.c — stubs for library calls for the new Intel libraries
libnin\errno.c — errno declared for the old Intel libraries
libnin\gcc_stub.c — stubs for library calls for the GNU libraries
libnin\heapsize.s — default size of heap
libnin\lib.c — system library routines
libnin\libasm.s — assembly routines needed by libnin.a
libnin\newlib.c — stubs to make the old Intel libraries compatible
libnin\ver960.c — library version number

Files specific to the QT960 library version

libqt\copyrght.c — copyright notice for inclusion with libqt.a
libqt\dma.c — DMA routines for the 82380
libqt\eat_time.s — wait loop used when talking to peripherals
libqt\flashlib.c — user callable Flash programming routines
libqt\qtasms.s — assembly routines needed for libqt
libqt\time_qt.c — benchmark timer routines
libqt\ver960.c — library version number

Files specific to the EV80960CA library version

libevca\copyrght.c — copyright notice for inclusion with libevca.a
libevca\eat_time.s — wait loop used when talking to peripherals
libevca\evcaasm.s — assembly routines used by libevca.a
libevca\noint_tm.c — non-interrupting version of benchmark timer
libevca\t82c54.h — header for non-interrupting version of benchmark timer
libevca\time_ev.c — benchmark timer routines
libevca\ver960.c — library version number

Startup and load files for all NINDY library versions

crt\crtfnin.s — C Run Time startup file for files running out of Flash

crt\crtnin.s — C Run Time startup file for a program running on NINDY

crt\evnindy.ld — linker/loader file for program running on NINDY on EV80960CA

crt\evrom.ld — linker/loader file for NINDY running on EV80960CA

crt\flrom.ld — linker/loader file for NINDY running in Flash on QT960

crt\flash.ld — linker/loader file for program running in Flash on QT960

crt\qtbig.ld — linker/loader file for program running on NINDY on QT960 (1 Meg SRAM)

crt\qtnindy.ld — linker/loader file for program running on NINDY on QT960

crt\qtrom.ld — linker/loader file for NINDY running in ROM on QT960

Working Library files

lib\crtfnin.o — to be linked with Flash application code running on the QT960

lib\crtnin.o — to be linked with all other code running on NINDY

lib\libevca.a — benchmark timers and EV80960CA board specific libraries

lib\libnin.a — basic I/O (read/write) libraries (use with all NINDY I/O)

lib\libqt.a — benchmark timer, Flash programming, QT960 board specific libraries

lib\libqtka.a — libqt.a compiled for the KA

lib\evnindy.ld — linker/loader file for program running on NINDY on EV80960CA

lib\evrom.ld — linker/loader file for NINDY running on EV80960CA

lib\flrom.ld — linker/loader file for NINDY running in Flash on QT960

lib\flash.ld — linker/loader file for program running in Flash on QT960

lib\qtbig.ld — linker/loader file for program running on NINDY on QT960 (1 Meg SRAM)

lib\qtnindy.ld — linker/loader file for program running on NINDY on QT960

lib\qtrom.ld — linker/loader file for NINDY running in ROM on QT960

Working ROM files

rom\evca.hex — file ready for burning into 27512 EPROM for the EV80960CA

rom\qt0.hex — file to burned into 27c202 (U33) on the QT960

rom\qt1.hex — file to burned into 27c202 (U34) on the QT960

rom\qt2.hex — file to burned into 27c202 (U43) on the QT960

rom\qt3.hex — file to burned into 27c202 (U44) on the QT960

rom\flrom — file ready for downloading to and running NINDY out of Flash on the QT960

Appendix B
C-Library Calls

QT960 Library Calls — (found in LIBQT.A)

DMA

init_dma()
This function must be called before the first call to "dma()". It sets up the 82380 dma controller with channel 0 as the highest priority. The data transfer is two cycles, and uses 32 bit source and destination busses. Channel 4 is not used.

dma (target_address, requester_address, bytes, channel)

```
unsigned int target_address;
unsigned int requester_address;
unsigned int bytes;
unsigned int channel;
```

This function starts up a dma transfer of "bytes" + 1 (zero bytes requested actually transfers one byte). "Target_address" is the address the data is going to, and "requester_address" is the address the data is coming from.

Benchmark Timer

long init_bentime(Hz)

```
int Hz;     /* not really used, but kept for compatibility */
            /* with the QT960 benchmark timer. */
```

This function must be called before the first call to "bentime()". It starts the timer running, and returns a long integer value which is the overhead for the call to "bentime()".

long bentime()
This function returns the current timer value in microseconds. The returned value is a long integer. Note: the timer is free running so a call to "bentime()" returns only the current value of the timer.

term_bentime()
This function terminates the free running timer. "Init_bentime()" must be called to restart the timer once "term_bentime()" has been called.

Waitstate Change

wait_states (first, burst, write_stretch)

```
int first;            /* value must be between 0 and 3 */
int burst;            /* value must be between 0 and 3 */
int write_stretch;    /* used if first or burst = 0 */
```

This function is specific to the QT960 board. It changes the waitstates for SRAM. "First" is the number of waitstates on the first access to SRAM and "burst" is the number of waitstates for the subsequent accesses. For example if "first" = 1, and "burst" = 0, a quad word access to SRAM would be (1-0-0-0).

Flash

init_flashO

This function initializes the Flash timer. It must be called before any of the other Flash routines are called.

erase_flashO

This function erases the Flash space. It is a whole chip erase.

program_flash(flash_start, data_start, bytes)

```
unsigned int flash_start;
unsigned int *data_start;
int bytes;
```

This function programs the flash at the starting location, "flash_start", for the given number, "bytes". "data_start" is a pointer to the data to be copied.

EV80960CA Library Calls — (found in LIBEVCA.A)

Interrupting Benchmark Timer

These functions use timer 2 of the 82C54 for a 16 bit interrupting timer. The 16 bit timer interrupts every 5 milliseconds to update a counter in memory, so use of this timer is somewhat intrusive for programs longer than 5 milliseconds, however it allows you to time until the counter in memory rolls over which is 200+ days later.

unsigned int init_bentime(Mhz)

```
int Mhz;     /* this value is not really used, */
/* but kept for compatibility */
```

This function must be called before the first call to "bentime()". It starts the timer running, and returns a long integer value which is the overhead for the call to "bentime()".

unsigned int bentime(void)

This function returns the current timer value in microseconds. The returned value is a long integer. Note: the timer is free running so a call to "bentime()" returns only the current value of the timer.

term_bentime(void)

This function is provided for compatibility.

Non-Interrupting Benchmark Timer

These functions use counters 0 and 1 of the 82C54 to form a 32 bit timer. The 32 bit timer rolls over every 7 minutes, so using this timer for benchmarks which are over 7 minutes long will result in inaccurate results.

long init_bentime_noint(Mhz)

```
int Mhz;     /* this value is the value of the TIMER */
/* oscillator (not the 960 clock frequency) */
```

This function must be called before the first call to "bentime_noint()". It starts the timer running, and returns a long integer value which is the overhead for the call to "bentime_noint()".

long bentime_noint(void)

This function returns the current timer value in microseconds. The returned value is a long integer. Note: the timer is free running so a call to "bentime_noint()" returns only the current value of the timer.

term_bentime_noint(void)

This function is provided for compatibility.

System Calls — (found in LIBNIN.A)

Values for the flags and errno messages may be found at the end of this section.

access(pathname, mode)

```
char *pathname;
short mode;
```

This function determines the accessibility of a file. 0 is returned if the file accessibility matches the "mode". –1 is returned if accessbility does not match.

mode:

00 checks for existence
02 checks for write permission
04 checks for read permission
06 checks for read and write permission

DOS handles modes 00 and 04 identically and modes 02 and 06 identically.

Upon an error condition on the host side, errno will be set to ENOENT if the path or file is not found, or EACCES if permission is denied on the file.

close(file_descriptor)

```
short file_descriptor;
```

The function close() closes the file associated with "file_descriptor" (obtained from an open() or creat() call), clears any internal data buffers and updates the directory associated with the file.

close() returns 0 if successful and -1 otherwise. errno is set to EBADF (bad file number) if the file descriptor is not of a valid, open file.

creat(pathname, mode)

```
char *pathname;
short mode;
```

This function creates a new file or rewrites an existing one. "mode" only applies to new files. If the file exists with write permission, the file is truncated to length 0. If the file exists and is read only, creat() fails.

mode:

S_IWRITE creates file with write permission only
S_IREAD creates file with read permission only
S_IWRITE I S_IREAD creates file with read and write permission

If creat() is successful it returns a file descriptor, otherwise it returns -1. errno is set to ENOENT if the file or path is not found, EMFILE if there are too many open files, or EACCES if permission is denied on the file.

lseek(file_descriptor, offset, where)

```
short file_descriptor;
int offset;
short where;
```

lseek() moves the file pointer associated with the "file_descriptor" to the location specified by the combination of "offset" and "where". "offset" is in number of bytes.

where:

0 the pointer is set to the beginning + offset
1 the pointer is set to the current pointer + offset
2 the pointer is set to the end + offset

lseek() returns the new file pointer position (in bytes) if successful, or -1 otherwise. errno is set to EBADF for a bad file_descriptor or EINVAL for an invalid argument.

open(pathname, flags, [mode])

```
char *pathname;
short flags;
unsigned short mode;
```

This function opens the file specified by "pathname", and sets the file pointer to the beginning. "mode" sets the access permission for a file being created by open(). The O_CREAT flag is used to create a file and must be accompanied by the "mode" argument.

flags:

can be only one of the following:

O_RDONLY	open for read only
O_WRONLY	open for write only
O_RDWR	open for read/write

and/or may be OR'd with any of the following:

O_APPEND	file pointer is set to EOF before writing
O_CREAT	file is created if it does not exist "mode" must be used with this argument
O_EXCL	used with O_CREAT to cause open to return error if the file already exists
O_TRUNC	if file exists, length is truncated to 0. File attributes remain the same (same owner and permissions)
O_BINARY	opens the file in binary mode
O_TEXT	opens the file in text mode

Caution *— \n may be translated to \r\n causing byte counts to be unpredictable for functions such as lseek(), write() and read() when file is opened in text mode.*

A file descriptor is returned when open() is successful and the file pointer is set to the beginning of the file. open() returns -1 if an error opening the file occurs. errno is then set to one of the following:

ENOENT - no such file or directory
EMFILE - too many open files
EACCES - permission denied
EINVACC - invalid access code

read(file_descriptor, buffer, length)

```
short file_descriptor;
char *buffer;
unsigned short length;
```

The function read() attempts to read "length" bytes from the file associated with file_descriptor, into the "buffer". read() begins reading at the current file pointer and increments the file pointer by the number of bytes read. read() may also read from the STDIN input stream. The maximum number of bytes which may be read is 65534.

read() returns the number of bytes read, or -1 if an error occurred. read() returns 0 on EOF. errno may be set to EACCES if read permission is denied for "file_descriptor", or EBADF if the "file_descriptor" is invalid.

stat(pathname, buffer)

```
char *pathname;
struct stat buffer;
```

stat() gets information about an open file. The information is returned in the stat structure pointed to by "buffer".

The stat structure contains the following fields:

st_dev - drive number of disk containing the file
st_ino - not used for DOS
st_mode - bit mask for file mode information
st_nlink - set to the constant 1
st_uid - not used for DOS
st_gid - not used for DOS
st_rdev - drive number of disk containing the file
st_size - file size in bytes
st_atime - time of last file modification
st_mtime - time of last file modification
st_ctime - time of last file modification

The st_mode bit mask has the following bits:
One of the following will be set:

S_IFREG set if the pathname is an ordinary file
S_IFDIR set if the pathname is a directory

One or both of the following will be set:

S_IREAD set if user has read permission on the file
S_IWRITE set if user has write permission on the file

Read/write bits are also included, set according to the file's permission mode. User-execute bits, also included, are set according to the open file's extension.

stat() returns 0 if the status information is successfully obtained. -1 is returned otherwise, and errno is set to ENOENT (file or path not found).

systemd(string)

```
char *string;
```

systemd() invokes a system command "string". If "string" is null, systemd() will invoke a shell on the host. The COMSPEC (DOS) and PATH environment variables are checked to locate the command to execute.

0 is returned if the command was successfully executed, -1 is returned otherwise.

This command is usually named system(), however it is named systemd() to be compatible with the EVA960 SDM.

long time(t)

```
long *t;
```

This function returns the time in seconds elapsed since 00:00:00 GMT, January 1, 1970. If "t" is not a null pointer, time is also returned in "t".

unlink(pathname)

```
char *pathname;
```

This function deletes the file specified by "pathname". Wildcards are not allowed in "pathname". Read only files cannot be deleted by this call. Using unlink() on an open file will cause unpredictable results.

unlink() returns 0 on successful completion. -1 is returned on an error and errno is set to ENOENT if the path or file is not found or EACCES if permission is denied on the file.

write(file_descriptor, buffer, length)

```
short file_descriptor;
char *buffer;
unsigned short length;
```

The function write() attempts to write "length" bytes from the "buffer" to the file associated with "file_descriptor". write() begins writing at the current file pointer and increments the file pointer by the number of bytes written. write() may also write to the STDOUT output stream. The maximum number of bytes which may be written is 65534.

write() returns the number of bytes written, or -1 if an error occurred. write() returns 0 on EOF. errno may be set to EACCES if write permission is denied for "file_descriptor", or EBADF if the "file_descriptor" is invalid.

errno messages:

```
#define EACCES      5
#define EBADF       6
#define EINVACC     12
#define EINVAL      19
#define EMFILE      4
#define ENOENT      2
```

permission flags:

```
#define S_IREAD     0x400
#define S_IWRITE    0x200
```

open flags:

```
#define O_APPEND    0x800
#define O_BINARY    0x8000
#define O_CREAT     0x100
#define O_EXCL      0x400
#define O_RDONLY    0x1
#define O_RDWR      0x4
#define O_TRUNC     0x200
#define O_TEXT      0x4000
#define O_WRONLY    0x2
```

status flags:

```
#define S_IFDIR     0x4000
#define S_IFREG     0x10000
```

```
struct stat {
                    short   st_dev;
           unsigned short   st_ino;
           unsigned short   st_mode;
                    short   st_nlink;
           unsigned short   st_uid;
           unsigned short   st_gid;
                    short   st_rdev;
                      int   st_size;
                      int   st_atime;
                      int   st_mtime;
                      int   st_ctime;
}
```