NET+OS with GNU Tools BSP Porting Guide

Making DEVICE NETWORKING easy™

NET+Works with GNU Tools BSP Porting Guide

Operating system/version: NET+OS 6.1
Part number/version: 90000579_B
Release date: March 2006
www.digi.com

©2006 Digi International Inc.

Printed in the United States of America. All rights reserved.

Digi, Digi International, the Digi logo, the Making Device Networking Easy logo, Digi, a Digi International Company, NET+, NET+OS and NET+Works are trademarks or registered trademarks of Digi International, Inc. in the United States and other countries worldwide. All other trademarks are the property of their respective owners.

Information is this document is subject to change without notice and does not represent a committment on the part of Digi International.

Digi provides this document “as is,” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of, fitness or merchantability for a particular purpose. Digi may make improvements and/or changes in this manual or in the product(s) and/or the program(s) described in this manual at any time.

This product could include technical inaccuracies or typographical errors. Changes are made periodically to the information herein; these changes may be incorporated in new editions of the publication.

Contents

Using This Guide

Review this section for basic information about the guide you are using, as well as for general support and contact information.

About this guide

This guide provides general information for porting the NET+OS Board Support Package (BSP) with GNU Tools to a new hardware platform based on the Digi development board.

NET+OS, a network software suite optimized for the NET+ARM devices, is part of the NET+Works integrated product family.

Software release

This guide supports NET+OS 6.1. By default, this software is installed in the `C:/NETOS61_GNU361/`directory.

Who should read this guide

This guide is for engineers who are developing applications with NET+OS.

To complete the tasks described in this guide, you must:

- Be familiar with installing and configuring network software and development board systems
- Have sufficient system or user privileges to do these tasks

What's in this guide

This table shows where you can find specific information in this guide:

To read about	See
The BSP and how it works	Chapter 1, "BSP Overview"
The modes in which NET+OS operates, and how interrupts are handled	Chapter 2, "Processor Modes and Exceptions"
How to modify the BSP to support your application hardware	Chapter 3, "Customizing the BSP for Application Hardware"
Device driver functions and device definitions	Chapter 4, "Device Drivers"
Adding new flash ROM parts to the existing table of supported flash ROM parts	Chapter 5, "Updating Flash Support"
The `bootloader` utility and how it works	Chapter 6, "Bootloader Utility Overview"
How to customize the `bootloader` utility to support your applications	Chapter 7, "Customizing the Bootloader Utility"
The linker files provided for sample projects, including basic GNU Tools and NET+OS sections of the linker files	Chapter 8, "Linker Files"
Hardware dependencies for porting NET+OS to your application hardware	Chapter 9, "Hardware Dependencies"

Conventions used in this guide

This table describes the typographic conventions used in this guide:

This convention	Is used for
italic type	Emphasis, new terms, variables, and document titles.
Select **menu → menu option** or **options.**	Menu selections. The first word is the menu name; the words that follow are menu selections.
monospaced type	Filenames, pathnames, and code examples.

Related documentation

- *NET+Works Quick Install Guide* explains how to install the hardware.
- *NET+Works with GNU Tools Tools Programmer's Guide* describes how to use NET+OS to develop programs for your application and hardware.
- The NET+Works online help provides reference information about the NET+OS application program interfaces (APIs).

For information about third-party products and other components, review the documentation CD-ROM that came with your development kit.

For information about the processor you are using, see the NET+Works hardware documentation.

Documentation updates

Digi occasionally provides documentation updates on the Web site.

Be aware that if you see differences between the documentation you received in your NET+Works package and the documentation on the Web site, the Web site content is the latest version.

Customer support

To get help with a question or technical problem with this product, or to make comments and recommendations about our products or documentation, use this contact information:

- United State telephone: 1 877 912-3444
- International telephone: 1 952 912-3444
- email: digi.info@digi.com
- Web site: http://digi.com

BSP Overview

CHAPTER 1

This chapter provides general information about the NET+OS Board Support Package (BSP) and describes how it operates.

Overview

The BSP consists of the hardware-dependent parts of the real-time operating system (RTOS), which are responsible for:

- Initializing the hardware after a hard reset or a restart
- Handling exceptions
- Device drivers
- Starting the ThreadX kernel
- Starting the Transmission Content Protocol/Internet Protocol (TCP/IP) stack

The NET+OS BSP that is provided with the NET+Works development kit is designed to work on NET+Works development boards. Because most user applications require custom hardware, you will need to modify the NET+OS BSP to work on your application hardware.

The rest of this chapter describes how the BSP initializes the hardware.

Initializing the hardware

At startup, the NET+ARM reads the state of all strapping pins to determine how to configure itself. All NET+ARM strapping pins have input current sources; if left unchanged, the strapping pins are in the high state. (For more information, see the *NS9750 Hardware Reference*.)

Hardware configuration settings are implemented through pullup and pulldown resistors attached to these strapping pins. NET+OS requires the processor to be configured as Big Endian, which is GPIO pin 44 connected to an internal pullup.

Startup

At startup, the NET+ARM performs these steps:

1. Resets all internal devices.
2. Maps ROM static `CS1` to address `0`.
3. After bootup, maps SDRAM dynamic `CS4` to address `0`.
4. Configures `CS1` to map the contents of the ROM to all memory.
5. Disables cache.

Reset

At reset, the ARM processor core performs these steps:

1. Enables ARM mode
2. Sets Supervisor mode
3. Disables `IRQ` and `FIQ` interrupts
4. Sets the program counter (PC) to `0`
5. Starts executing at `0` (4 ms after the reset is deasserted and the PLL has locked)

ROM startup

After a hardware reset, the first code executed is in `C:/NETOS61_GNU/src/bsp/common/reset.s`. This code jumps to the `Reset_Handler_Rom` routine in `C:/NETOS61_GNU/src/bsp/arm9init/init.s`, which performs these steps:

1. Sets the processor mode to `supervisor` and disables all interrupts.
2. Determines whether the hardware is being reset or is waking up from sleep mode by examining the sleep bit in the reset and Sleep Control register. Sleep mode starts when the software sets this bit. The hardware leaves the bit set when it wakes up again.

If the hardware is in sleep mode, forces a hard reset to put everything in the power on reset state by:

- Clearing the sleep bit in the reset register
- Using the watchdog timer to force a reset

3. Sets the supervisor stack pointer to use a piece of SDRAM on `CS4`.
4. Checks whether the application is in software restart mode, which is caused if the program counter (PC) is set back to `0`.
   - If the application is in software restart mode or in the debugger, `Reset_Handler_Rom` ignores the memory initialization code. The debugger initialization script sets bit 1 in the SCM AHB Gen module (`0xA0900000`), which the software then uses to determine whether the application is running in the debugger. The sample debugger initialization scripts that are provided with NET+Works 6.1 set this bit.
   - If the application is in the debugger, `CS4` is initialized in the `ns9750_a.cmd` debugger initialization script.
   - If the application is not in the debugger, `Reset_Handler_Rom` configures SDRAM on `CS4` based on the values in the platform's `setting.s` file. (The `settings.s` file is described in Chapter 2, "Processor Modes and Exceptions.")
5. Performs a memory test on the piece of RAM that will be used as a stack for calling the `nccInit` routine.
6. Calls the `nccInit` routine to initialize the application and calls the customizable routines to configure the board.
7. Sets up stacks for all processor modes.
8. Calls the C library startup routine, which was defined at START.

   This routine does not return.

nccInit routine

To complete the basic hardware initialization of the system, `Reset_Handler_Rom` calls the `nccInit` routine. Note that this routine is called before the C library is initialized.

The `nccInit` routine performs this process:

1. Determines whether the application is in the debugger by looking at the SCM configuration register bit 1, which must be set by the debugger initialization script.
2. Determines whether a software restart condition has occurred by looking at the SCM configuration register bit 2. This bit is set after it is read to determine whether a software restart has occurred.
3. Calls the `SetupSimpleSerial` routine to set up a simple serial driver, which is used to debug board initialization. The simple serial driver can be used in a way similar to `printf`.
4. Calls the `customizeSetupGPIO` routine to set up the general-purpose I/O (GPIO) ports.
5. Calls `customizeSetupCS0` to set up `CS0`.
6. Sets up the other chip selects.

   If a startup reset has occurred, all the chip selects are initialized. Otherwise, the default implementation assumes that the RAM chip selects have already been set up and reinitializes only `CS1`, which normally is used to support non-volatile RAM (NVRAM).
7. Calls `customizeReadPowerOnButtons`, which allows you to store key presses on startup to read and save the state of buttons and jumpers.

   For example, a particular sequence of key presses may force a device into maintenance mode.
8. Verifies that the application can fit in available RAM.
9. Sets flags in memory, which is now set up, to indicate whether:
   - A debugger is present.
   - A software reset has occurred.

C library startup

The C library startup sets up the C runtime environment in two steps:

1. Copies initialized data from ROM to RAM and resets uninitialized data to `0`
2. Calls the `main` routine (described next)

main routine

After the C runtime environment is set up, the C runtime code calls the `main` routine. The `main` routine performs this process:

1. Calls `customizeSetupLedTable` to provide hooks for adjusting the `NAMedTable` define in `customizeled.c`.
2. Executes a power-on self-test (POST) if `APP_POST` is set.
3. Sets up the system vector table.
4. Calls `customizeEnableMmu` to initialize the memory management unit.
5. Calls `NABoardInit` to initialize the low-level flash and NVRAM APIs.
6. Calls `DDIFirstLevelInitilize` to perform preliminary initialization of system device drivers.
7. Calls `NAGetAppCpp`, which initializes the C++ runtime environment if `APP_CPP` is defined.
8. Calls `tx_kernel_enter` to start the ThreadX kernel.

   This function does not return.

NABoardInit routine

The `NABoardInit` routine in the `narmbrd.c` file performs this process:

1. Reads the chip revision and stores it in the `g_NAChipRevision` global variable
2. Initializes the flash and NVRAM read and write APIs

netosStartup routine

The startup sequence is completed in the `netosStartup` routine, which performs these steps:

1. Sets up the semaphores that the flash driver uses in a multithreaded environment.
2. Sets up the sleep function that the `NAWait` function uses.
3. Loads default NVRAM parameters from the `appconf.h` file.
4. Calls `naHdlcload` to load the HDLC driver and `DDISecondLevelInitialize` to load the system device drivers.

   This function calls the `deviceInit` function defined for each device in the device table.
5. Creates the TCP timer thread to trigger calls to `tcp_down_function`, which calls the `TcpDown` application function each second.
6. Calls `customizeDialog` to prompt the user with a configuration dialog box.
7. Calls `netosConfigStdio`.

   If `APP_STDIO_PORT` is defined in `appconf.h` as a valid device name, `netosConfigStdio` redirects `stdin`, `stdout`, and `stderr` to the indicated device.
8. Calls `netosStartTCP` to create and initialize resources (for example, semaphores and memory), and to start up all threads required by the TCP/IP stack.

   The Address Configuration Executive (ACE) is used to set the IP parameters. The parameters can be stacked in NVRAM, or the ACE can set them from the network.

   The application startup is delayed `BSP_STARTUP_DELAY` seconds. `BSP_STARTUP_DELAY`, which is defined in `bsp.h`, defines the number of seconds the system waits before starting the application.
9. Calls `applicationTcpDown` once every tick until the TCP/IP stack is initialized.
10. After the TCP/IP stack is loaded, deletes the timer thread created in step 5.

11 Sets up Point-to-Point Protocol (PPP) -related semaphores to synchronize PPP-related threads.

12 Starts the time APIs and sets the time zone to the value stored in NVRAM, which is `EST` by default.

13 Calls `filesystem_init` if `BSP_INCLUDE_FILESYSTEM_FOR_CLIBRARY` is defined as `1`. The `filesystem_init` call initializes the file system for C library file functions.

14 Calls `applicationStart` after the TCP/IP stack has started.

This table shows you the locations of the routines described in the previous procedure:

Routine name	Location
`netosStartup`	`src/bsp/common`
`DDISecondLevelInitialize`	`src/bsp/common`
`netosConfigStdio`	`src/bsp/common`
`customizeDialog`	`platforms` directory
`netosStartTCP`	`src/bsp/common`
`filesystem_init`	`platforms` directory

Processor Modes and Exceptions

CHAPTER 2

This chapter discusses the modes in which NET+OS operates. This chapter also describes how NET+OS handles interrupts.

Overview

The ARM processor supports seven modes. This table lists the modes and describes how they are used:

Mode	Used for
User	Normal user code
SVC (supervisor)	■ Processing software interrupts ■ NET+OS ■ All threads ■ The kernel scheduler
Abort	Processing memory faults
System	Running privileged operating system tasks
Undef (undefined)	Handling undefined instruction traps
IRQ (interrupt)	■ Processing standard interrupts ■ NET+OS
FIQ (fast interrupt)	Processing fast interrupts

Hardware interrupts cause the processor to switch to IRQ mode.

The IRQ handler switches back to SVC mode before it calls the device's service routine, allowing higher priority devices to interrupt the service routine, if necessary.

The rest of this chapter describes how NET+OS handles exceptions.

Vector table

An *exception* occurs when the normal flow of a program halts temporarily; for example, to service an interrupt. Each exception causes the ARM processor to save some state information and then jump to a location in low memory. This location in memory is referred to as the *vector table.*

A vector table is stored from 0x00000000 to 0x0000001f. Each vector consists of a 32-bit word that is a single NET+ARM instruction. The instruction loads the program counter with the contents of a memory location, which implements a 32-bit jump to an interrupt service routine (ISR).

This table shows the vector address for each exception type:

Exception	Vector address
Reset	0x00000000
Undefined instruction	0x00000004
Software interrupt (SWI)	0x00000008 (not used by NET+OS)
Prefetch abort	0x0000000c
Data abort	0x00000010
Interrupt (IRQ)	0x00000018
Fast interrupt (FIQ)	0x0000001c

NET+OS treats these exception types as fatal errors:

- Prefetch aborts
- Data aborts
- Undefined instructions
- Fast interrupts

The handler for these exception types is located in src/bsp/arm9init/init.s.

The default FIQ handler and the exception types in the table call the customizeExceptionHandler routine.

Although NET+ARM does not allow external devices to trigger fast interrupts, application software can program the watchdog timer and the general-purpose timer to trigger a fast interrupt.

The default FIQ handler normally calls customizeExceptionHandler. For more information about FIQs, see "FIQ handler," later in this chapter.

IRQ handler

The BSP provides an IRQ handler. An interrupt request is generated when one or more devices assert their interrupt signal. The IRQ handler reads the Interrupt Service Routine Address register (ISRADDR) and the Active Interrupt Level Status register to determine which devices need to be serviced.

The `IRQ` signal is multiplexed by the interrupt controller built into the NET+ARM to support 32 signals:

- 26 interrupt signals support AHB devices that are internal to the NS9750.
- 1 interrupt signal supports Bbus devices that are internal to the NS9750.
- 4 interrupt signals support external devices.
- 1 interrupt signal is not used and is considered *reserved*.

Application software can selectively enable or disable any of the interrupt signals with `MCEnableIsr` and `MCDisableIsr`.

The IRQ handler for Bbus uses a prioritized interrupt scheme. If more than one device requests service, the handler determines which device has higher priority and services that device first. Interrupts for higher priority devices are enabled before the device's service routine is called, allowing the device's service routine to be interrupted if a higher priority device requests service.

Servicing AHB interrupts

The NET+OS IRQ handler uses this procedure to service an AHB interrupt:

1. A device requests service by asserting its interrupt signal.
2. The NET+ARM latches the request into the ISR Address register (`ISRADDR`).
3. After the signal has been latched, and if the interrupt pin is edge-triggered, the NET+ARM generates the interrupt, even if the device stops asserting its interrupt line.
4. When one of the corresponding interrupts configured in the Interrupt Configuration register is invoked, the NET+ARM asserts the `IRQ` signal to the ARM CPU.

5 If interrupts are enabled when the IRQ signal is asserted, the ARM CPU switches to IRQ mode and jumps to the IRQ handler.

6 The IRQ handler saves the context of the interrupted thread and switches to SVC mode to service the interrupt.

7 The IRQ handler calls MCIrqHandler in the mc_isr.c file, which reads the ISRADDR register to determine which device interrupt to process.

8 The IRQ handler clears the interrupt for the specific device.

When all pending interrupts have been serviced, NET+OS restores the context of the interrupted thread and resumes processing the thread.

Servicing Bbus interrupts

The Bbus IRQ handler uses this procedure to service an interrupt:

1 A Bbus device requests service by asserting its interrupt signal with Bbus Aggregate Interrupt.

2 The MCIrqHandler in mc_isr.c calls BBUS_IrqHandler to service devices on the Bbus.

3 In a loop, Bbus_IrqHandler masks all lower priority interrupts, enables interrupts, and calls the function registered during the MCInstallIsr call.

After the handler completes this procedure, it disables the interrupts that are lower priority than the one currently being processed. The loop repeats until the handler services all interrupt levels. When all pending interrupts have been serviced, control is retuned back to MCIrqHandler.

Changing interrupt priority

You can change the interrupt priority level by changing the order of the MCAhbPriorityTab and MCBbusPriorityTab arrays in the bsp.c file. The tables in the next sections, "AHB interrupts" and "Bbus interrupts," show the contents of the arrays, ordered from lowest to highest priority. You can specify each priority only once.

NET+OS treats incorrect ordering as a fatal error (that is, NET+OS calls customizeErrorHandler).

AHB interrupts

The priority of each interrupt in the AHB Bus is controlled by hardware. The priority is set by the order configured in the Interrupt Configuration register. When an interrupt occurs:

- Its handler is stored in the ISR Address register.
- Its priority level is stored in the Active Interrupt Level Status register.

The driver executes the interrupt handler, with the priority level passed as a parameter. An interrupt with a higher priority can preempt the current interrupts. After the call of the interrupt handler is completed, the interrupt driver automatically clears the interrupt to be reused.

This table lists the supported interrupt sources in the AHB Bus and the associated software directives. The priority for each AHB source interrupt is specified in the `MCAhbPriorityTab` array in the `bsp.c` file.

Interrupt sources with a higher-numbered priority level can interrupt the service routines of devices with lower-numbered priority levels.

AHB Interrupt source	Software directive
External 3	EXTERNAL3_INTERRUPT
External 2	EXTERNAL2_INTERRUPT
External 1	EXTERNAL1_INTERRUPT
External 0	EXTERNAL0_INTERRUPT
Timer 14 and 15	BUS AGGREGATE_INTERRUPT
Timer 12 and 13	TIMER12-13_INTERRUPT
Timer 10 and 11	TIMER10-11_INTERRUPT
Timer 8 and 9	TIMER8-9_INTERRUPT
Timer 7	TIMER7_INTERRUPT
Timer 6	TIMER6_INTERRUPT
Timer 5	TIMER5_INTERRUPT
Timer 4	TIMER4_INTERRUPT
Timer 3	TIMER3_INTERRUPT

AHB Interrupt source	Software directive
Timer 2	TIMER2_INTERRUPT
Timer 1	TIMER1_INTERRUPT
Timer 0	TIMER0_INTERRUPT
Reserved	AHB_PERIPH15_INTERRUPT
I2C	12C_INTERRUPT
PCI External 3	PCI_EXTERNAL3_INTERRUPT
PCI External 2	PCI_EXTERNAL2_INTERRUPT
PCI External 1	PCI_EXTERNAL1_INTERRUPT
PCI External 0	PCI_EXTERNAL9_INTERRUPT
PCI Arbiter	PCI_ARBITER_INTERRUPT
PCI Bridge	PCI_BRIDGE_INTERRUPT
LCD	LCD_INTERRUPT
Ethernet PHY	ETH_PHY_INTERRUPT
Ethernet Transmit	ETH_TRANSMIT_INTERRUPT
Ethernet Receive	ETH_RECEIVE_INTERRUPT
Reserved	N/A
Bbus Aggregate	TIMER14-15_INTERRUPT
AHB Bus Error	AHB_BUS_ERROR_INTERRUPT
Watchdog	WATCHDOG_INTERRUPT

Bbus interrupts

The priority in the Bbus is controlled by the logic in the Bbus interrupt handler. Each device on the Bbus shares the Bbus Aggregate interrupt, a common interrupt on the AHB bus.

When a device signals an interrupt, these steps occur:

1. The hardware sets bits in the Bbus Bridge Interrupt Status register to indicate which device on the Bbus is signaling the event.
2. If the device's interrupt level is not masked off, the hardware generates an IRQ exception, causing the NET+OS interrupt driver to be executed.
3. The Bbus Interrupt Handler determines which device is signaling the interrupt condition and calls the ISR that is registered to it.
4. The ISR processes the interrupt and then returns.
5. At this point, the interrupt driver checks for more pending interrupts. If any interrupts are found, their ISRs are called as well.
6. When all pending interrupts have been processed, the NET+OS interrupt driver returns control to the application.

This table lists the supported interrupt sources in the Bbus and the associated software directives. The priority for each Bbus interrupt source is specified in the `MCBbusPriorityTab` array in the `bsp.c` file. Interrupt sources with a higher-numbered priority level can interrupt the service routines of devices with lower-numbered priority levels.

Bbus interrupt source	Software directive
IEEE 1284	IEEE_1284_INTERRUPT
Bbus DMA 16	BBUS_DMA16_INTERRUPT
Bbus DMA 15	BBUS_DMA15_INTERRUPT
Bbus DMA 14	BBUS_DMA14_INTERRUPT
Bbus DMA 13	BBUS_DMA13_INTERRUPT
Bbus DMA 12	BBUS_DMA12_INTERRUPT
Bbus DMA 11	BBUS_DMA11_INTERRUPT
Bbus DMA 10	BBUS_DMA10_INTERRUPT
Bbus DMA 9	BBUS_DMA09_INTERRUPT
Bbus DMA 8	BBUS_DMA08_INTERRUPT
Bbus DMA 7	BBUS_DMA07_INTERRUPT
Bbus DMA 6	BBUS_DMA06_INTERRUPT

Bbus interrupt source	Software directive
Bbus DMA 5	BBUS_DMA05_INTERRUPT
Bbus DMA 4	BBUS_DMA04_INTERRUPT
Bbus DMA 3	BBUS_DMA03_INTERRUPT
Bbus DMA 2	BBUS_DMA02_INTERRUPT
Bbus DMA 1	BBUS_DMA01_INTERRUPT
AHB DMA 2	AHB_DMA02_INTERRUPT
AHB DMA 1	AHB_DMA01_INTERRUPT
Utility	UTIL_INTERRUPT
Bbus peripheral	BBUS_PERIPH10_INTERRUPT
Serial 1 receive	SER1RX_INTERRUPT
Serial 2 receive	SER2RX_INTERRUPT
Serial 3 receive	SER3RX_INTERRUPT
Serial 4 receive	SER4RX_INTERRUPT
Serial 4 transmit	SER4TX_INTERRUPT
Serial 3 transmit	SER3TX_INTERRUPT
Serial 2 transmit	SER2TX_INTERRUPT
Serial 1 transmit	SER1TX_INTERRUPT
USB	USB_INTERRUPT
Bbus DMA	BBUS_DMA_INTERRUPT

Interrupt service routines

The IRQ handler calls Interrupt Service Routines (ISRs) to service interrupts that external devices generate. You can implement ISRs as standard C functions. The ISRs must clear the interrupt condition – usually by acknowledging it – and service the interrupt. Then the ISRs can return as standard C functions.

Because interrupts are enabled for higher priority interrupt levels when the ISR is called, an ISR with a higher priority can interrupt the processing of one with a lower priority.

Installing an ISR

You install an ISR by calling `MCInstallIsr`. After this routine returns, the ISR is installed, and the interrupt associated with the ISR is enabled.

Disabling and removing an ISR

To disable and remove an ISR, call `MCUninstallIsr`, which disables the interrupt and uninstalls the ISR handler.

FIQ handlers

A fast interrupt (FIQ) is a higher priority interrupt than an IRQ, and therefore, it can interrupt an IRQ at any time. If you install an interrupt to be a FIQ, you must give it the higher priority.

The default handler installed by the BSP treats a FIQ exception as an error (that is, it calls `customizeErrorHandler`).

To program `MCInstallIsr` to generate a FIQ interrupt, specify the `isrFlag` argument with the `MC_FLAG_FAST_IRQ` option flag. You can configure only the interrupts listed in the AHB Interrupt table. Unlike an IRQ, only one interrupt can be configured for an FIQ.

The FIQ handler must clear the ISRADDR register after the handler completes its task.

The watchdog timer and the two general-purpose timers generate a FIQ interrupt. To enable these interrupts, set the corresponding bits in the Interrupt Enable register. (For descriptions of the System Control register, Timer 1 and Timer 2 Control registers, and the Interrupt Enable register, see the *NS9750 Hardware Reference*.)

The Interrupt Enable register and the Interrupt Status register function identically for both IRQ and FIQ levels. Because NET+OS normally does not use FIQs, the IRQ handler does not contain special handling for FIQs. Instead, the IRQ handler dispatches FIQs according to their interrupt source default priorities.

Installing a FIQ handler

▶ **To install a FIQ handler:**

1. Write the address of the application FIQ handler to memory location `0x0000003C`.
2. Enable the FIQs bit in the Interrupt Configuration register for the specific source interrupt.
3. Modify the IRQ handler routine to exclude the FIQs from being dispatched with the IRQs.

The IRQ handler code is in these files:

- `na_isr.c`
- `reset.s`
- `init.s`.

Be aware that NET+OS normally does not use FIQs. The statistical profiler utility, however, which helps you identify system bottlenecks so you can improve system performance, does use FIQs. For an example of how you can install and use FIQs, see `bsp/profiler/profilerAPI.c`.

Disabling and removing a FIQ

To disable and remove a FIQ, call `MCUninstallIsr`. No other interrupt can be configured for a FIQ unless the previous FIQ is removed.

Customizing the BSP for Application Hardware

CHAPTER 3

This chapter describes how to customize the NET+OS Board Support Package (BSP) for your application hardware. This chapter also provides general information about the BSP and presents the basic tasks for porting the BSP to a new hardware platform.

Overview

This table lists and briefly describes the basic tasks for porting the BSP to your application hardware. You may find it helpful to print this table and use it as a checklist as you port the BSP.

Task	Action
1	Purchase Ethernet media access controller (MAC) addresses from the IEEE.
2	Create a new `platform` directory.
3	Modify the BSP `makefile`.
4	Modify the linker scripts.
5	Modify the BSP configuration files to support your application hardware.
6	Modify the BSP to start up the required drivers.
7	Modify the format of BSP arguments in non-volatile random access memory (NVRAM).
8	Modify the error and exception handlers.
9	Verify the debugger initialization files.
10	Debug the initialization code.
11	Modify the startup dialog.
12	Modify the power-on self-test (POST) routines.
13	Modify the Address Configuration Executive (ACE), which controls TCP/IP configuration on startup. (For details about ACE, see the online help.)

The rest of the chapter discusses each task in detail.

Follow the reference design

When you design your application hardware, follow the NET+Works reference design as closely as possible. This practice allows you to reduce the amount of modification to the BSP and reduces your risk during board bring-up (debug).

In addition, use the same parts as used on the NET+Works development board, especially memory peripherals and Ethernet PHY devices.

Verify the features your hardware supports

Make sure your hardware supports these features:

- **Flash at CS1.** The NS9750 boots from this flash on powerup.
- **RAM (32-bit wide) at CS4**. If you are using multiple chip selects for SDRAM, you must put the largest SDRAM on `CS4`. `CS4` is mapped to address `0` after the Memory Controller is enabled.

  The BSP autodetects and configures additional SDRAM memory on the other chip selects.
- **JTAG port.** This port, which allows you to debug the hardware and software, is essential for bringing up a new board.
- **Extra serial port.** This port is used to display standard out messages for debugging. Diagnostic information easily can be communicated to the debugging engineer by way of standard I/O `printf`.
- **Enough RAM to run your entire application, even if your product runs from ROM**. Being able to run an application from RAM greatly simplifies debugging.
- **A way to disable flash ROM**. This feature is necessary because flash can be overwritten accidentally. In such a situation, the NET+ARM CPU executes garbage instructions when you start it up.

Task 1: Purchase and assign Ethernet MAC addresses

Each device on a network needs a unique Ethernet MAC address. Your company must purchase its own block of addresses from the IEEE. After you purchase a block of addresses, you must assign an address to each board.

The addresses are stored in either NVRAM or flash ROM. Digi provides an Ethernet MAC address with each development board, but you need a unique address for your own boards.

Task 2: Create a new platform subdirectory

To support your application hardware, you need to modify code in the BSP.

The `src/bsp/platforms` directory contains a set of subdirectories for each supported platform. Each subdirectory contains all the code specific to a particular development board. For example, the subdirectory `ns9750_a` contains the code needed to support the NS9750 development board.

You need to create a new subdirectory to hold the platform-specific code for your application hardware. In this document, the new subdirectory is referred to as the *platform directory*.

▶ **To create the new platform subdirectory:**

1. Determine which development board platform is closest to your application hardware.
2. Copy the platform's subdirectory, subtree, and all its contents to `src/bsp/platforms/`*`your platform`*.

   You modify the directory subtree to interoperate with your hardware.

Be aware that it is unusual to modify any code outside the platform tree unless you are adding your own drivers or modifying existing devices. *If you believe you must modify files outside the platform tree, contact Digi technical support for confirmation.*

Task 3: Building and modifying the BSP makefile

The next step is to build the BSP `makefile` to support your platform. You can modify the `makefile`, but in most cases, no changes are necessary.

Building the BSP

▶ **To build the makefile:**

1. From Windows, open an X-Tools window using either of these steps:
   - Select **Start → Programs → C:\NET + OS61_GNU→ X-Tools v3.0 Shell.**
   - Double-click the X-Tools icon on your desktop.
2. At the `bash ($)` prompt, enter:

   ```
   xtools arm-elf
   ```
3. Change to the `src/bsp` directory:

   ```
   cd src/bsp
   ```
4. Enter this command:

   ```
   make PLATFORM=platform
   ```

   where you replace *`platform`* with the name of your platform's subdirectory.

Modifying the BSP

You may want to modify the `makefile`. For example, you could change the `makefile` to build your platform by default. This change lets you build the BSP just by entering `make`.

▶ **To modify the `makefile` to build your platform by default:**

1. At the `bash ($)` prompt, locate the line at the beginning of the `makefile` that sets the default value of the make variable *`PLATFORM`*.
2. Change this line to make your platform the default.

The targets `clean` and `all` are supported in this `makefile`, so to rebuild the entire BSP, enter `make clean all` in the `src/bsp` directory.

Task 4: Modify the linker scripts

The `customize.ldr` file declares a set of constants used to generate the linker scripts. These constants control the size and location of the program sections.

Constants you might need to change

This table lists the constants you might need to change for most applications:

Constant	Description
`RAM_SIZE`	The size of the RAM part on the board. The linker generates an error if the application is too large to fit in RAM.
`FLASH_SIZE`	The size of the flash part on the board. The linker generates an error if a ROM-based application is too large to fit in ROM.
`FLASH_START`	The starting address of flash. For the NS9750 processor, this address is typically `0x50000000`.
`NON_CACHE_MALLOC_SIZE`	Defines the size of the non-cacheable region of memory that is available through the `nonCacheMalloc` and `nonCacheFree` routines. The size must be a multiple of 1 MB. The memory returned from `malloc` is in a cacheable region.
`RAM_START`	The starting address of RAM.
`FILE_SYSTEM_SIZE`	The number of bytes to be allocated for the file system in flash.
`BOOTLOADER_SIZE_IN_FLASH`	The amount of flash ROM to be reserved for the `bootloader`, which is the code that executes and decompresses an application to run from RAM. This is used only if you want to run your image from RAM. You also can use this constant to calculate where the application image starts in flash. The `bootloader` currently fits into one sector of flash that is typically 64 K. It is important that this is large enough to fit the `bootloader` ROM image, which is in the `src/bsp/bootloader/ROMimage/rom.bin` directory.

Constant	Description
MAX_CODE_SIZE	The largest possible size of the uncompressed application image. Use this constant to reserve enough RAM to hold the application image. When you create the bootloader, use this constant to reserve a section of memory to hold the application. When you create your application, use MAX_CODE_SIZE to reserve memory in uncached memory so that the alias of the application does not collide with RAM used for data storage. The compression algorithm used by the bootloader generally achieves 2:1 compression. A good rule of thumb is to set this constant to twice the amount of flash available to hold the compressed application image. The linker generates an error if an application image is larger than this value.
INIT_DATA_START	Determines where the init data is stored in RAM. The init data section stores information read by the initialization code that needs to be accessed later. Enough space must be left from the start of RAM to hold the vector table, and possibly a FIQ routine, if you decide to write one.
INIT_DATA_SIZE	The size of the memory area used to hold the start of switches and buttons read at powerup.
CODE_START	The start of ROM code.
NVRAM_FLASH_SIZE	Determines how much flash ROM is reserved for NVRAM storage. Set this constant to 0 if flash is not used for NVRAM.

Bootloader considerations

The `bootloader` utility, which is executed on startup, decompresses the application image in flash to RAM and executes it. The `bootloader` must:

- **Know where in RAM to decompress the application image to.** The `bootloader` creates the application header from information in the `bootldr.dat` file. Included in `bootldr.dat` is a field, `ramAddress`, whose value determines the load address of the application in memory. When the header is generated, it is "tacked onto" the beginning of the application image. To determine where to decompress the application to, the `bootloader` reads the `ramAddress` field in the application's header.
- **Be positioned in RAM where it will not overwrite itself when it decompresses the application.** The `ramAddress` field *must* be set to the value of `BOOTLOADER_CODE_START` in the `customize.ldr` file.

Task 5: Modify BSP configuration files

You need to configure the BSP for your platform. The BSP configuration settings are stored in files in the `platforms` directory. The online help and comments within the files describe the content of the configuration files. Modify the configuration settings to support your application hardware.

The next sections describe the files you must modify to support your hardware. You may find it helpful to review the Memory Controller information in the *NS9750 Hardware Reference*.

sysclock.h file

The value for the external oscillator or crystal that supplies the input frequency to the NS9750 is defined in the `sysclock.h` `platforms` file. This line defines the input frequency:

```
#define NA_ARM9_INPUT_FREQUENCY 398131200
```

The value `398131200` is the input frequency to the NS9750 development boards. If your input frequency is different, you must modify this value.

bsp.c file

Static memory table

The `MCStaticMemoryTable` array in the `bsp.c` file in the `platforms` directory holds the timing settings for the SRAM (flash) memory parts. The values in the table correspond to the SRAM register settings for the NS9750 memory controller. (For more information, see the *NS9750 Hardware Reference*.)

The data structure that corresponds to this table is defined in the `bsp.h` header file and described in the online help. The values in the table correspond to the SRAM part supplied on the NS9750 development board. The online help also has a description of this table. If you are using a flash part that is different from what's on the NS9750 development board, you need to modify this table.

Interrupt tables

When you change the system interrupt priority, you must update these tables:

- **MCBbusPriorityTab.** This array in the `bsp.c` file in the `platforms` directory contains the priority of each interrupt in the Bbus. The `MCBbusPriorityTab` allows flexible prioritization for all BBUS interrupts in the NET+ARM that drive the `BBUS_AGGREGATE_INTERRUPT` in the `MCAhbPriorityTab` table. The `MCBbusPriorityTab` table is configured with interrupts of lower priority at the beginning and interrupts of higher priority toward the end of the array.
- **MCAhbPriorityTab.** This array in the `bsp.c` file in the `platforms` directory contains the priority of each interrupt in the AHB Bus. The `MCAhbPriorityTab` allows flexible prioritization for all the AHB interrupts in the NET+ARM that drive the ARM processor IRQ. The table is configured with interrupts of lower priority at the beginning and interrupts of higher priority toward the end of the table.

For more information about interrupts, see the sections "AHB interrupts" and "Bbus interrupts" in Chapter 2, "Processor Modes and Exceptions."

settings.s file

The settings.s file contains the SDRAM settings used to program CS4 (the RAM at address 0). You need to configure these settings before accessing SDRAM, which is done in the Reset_Handler routine. You also need to verify that the memory settings in this file are correct for your SDRAM.

The register settings supplied in the NS9750 platform are for the PC133 parts supplied on the development board. The register settings in this file are described in detail the *NS9750 Hardware Reference*.

cs.c file

The BSP_MPMC_REFRESH_RATE define contains the value for the SDRAM refresh rate. This define is used to calculate the value for the Dynamic Memory Refresh Timing register in the memory controller. You must modify this define to match the refresh rate for the memory parts you are using.

The next table lists the customization hooks in the cs.c file, which contains the routines that configure the NET+ARM chip selects to support memory parts. You need to modify the code in these routines to support your application hardware. Note that CS4 is already programmed in the initialization code with the parameters in settings.s. CS1 connected to flash.

Customization hook	Hardware feature/default values set
customizeGetRamSize	Returns the total amount of RAM on the system and calls the customizeable customizeGetCSSize routine.
customizeGetCSSize	Returns the total number of bytes of memory the chip select is configured to support by examining the address mask in the CS mask register.
customizeSetupCS0	CS0 has its powerup value when this function is called. The table in the bsp.c file is used to set the registers in the Memory Controller. CS0 has an optional SRAM device connected to it.

Customization hook	Hardware feature/default values set
customizeSetupCS1	Sets up CS1 (flash ROM). CS1 contains the flash code, which the processor initially starts executing on bootstrap. CS1 is preconfigured through the use of strapping pins and must always be connected to flash. The size and starting address of flash come from the linker directive file that is created from customize.ldr.
customizeSetupCS2	Sets up CS2 (Static Memory). The table in the bsp.c file is used to set the registers in the Memory Controller.
customizeSetupCS3	Must set up CS3 (Static Memory). CS3 has its powerup value when this function is called. The table in bsp.c is used to set the registers in the Memory Controller for this chip select.
customizeSetupCS4	Called to customize CS4 (SDRAM). CS4 is initially set up in init.s with the parameters from settings.s. The size of the RAM on CS4 must be specified in the customize.ldr file. CS4 gets mapped to address zero after the memory controller is enabled.
customizeSetupCS5	Must set up CS5 (optional SDRAM) and fill in the size of the amount of RAM detected on this chip select. This is then used to create the memory map.
customizeSetupCS6	Must set up CS6 (optional SDRAM) and fill in the size of the amount of RAM detected on this chip select. This is then used to create the memory map.
customizeSetupCS7	Must set up CS7 (RAM) and fill in the size of the amount of RAM detected on this chip select. This is then used to create the memory.
customizeSetupMMCR	Sets up the memory management control register (MMCR), which controls the SDRAM refresh timing.This routine sets up the MMCR for the NET+OS development board. The refresh rate is calculated from the MPMC_REFRESH_RATE define in bsp.h. You need to adjust this value to equal the refresh rate of the SDRAM part you are using. A default refresh rate already has been set up, but you may want to optimize this value.

Because the routines in cs.c execute before RAM is set up and before the C library is initialized, the routines cannot use:

- Global variables
- Static variables
- Constants created with the C const keyword

A small amount (512 bytes) of SDRAM is used to support a stack. The routines can create local variables on this stack if the variables are small enough to fit.

gpio.h file

The NS9750 has 50 pins that are multiplexed with functions that include GPIO functionality. You can quickly configure these pins using the definitions in the gpio.h file. The multiplexed functions include: serial, LCD, Timers, DMA, 1284, USB, Ethernet, external IRQs, and GPIO.

By selecting options other than BSP_GPIO_MUX_INTERNAL_USE_ONLY, you can define, set up, and program groups of pins at system startup to functions other than GPIO.

For information about how pins are multiplexed, see the gpio.h file and the *NS9750 Hardware Reference*. The gpiomux_def.h public header contains definitions used by the gpio.h file.

This table lists the customization hooks in gpio.h:

Customization hook	Hardware features/default values set
BSP_GPIO_MUX_ETHERNET_CAM	Controls how the Ethernet Content Address Memory (CAM) signals reject and request are multiplexed. Default is set to internal.
BSP_GPIO_MUX_USB_POWERRELAY	Controls how the USB Power Relay signal is multiplexed. Default is set to internal.
BSP_GPIO_MUX_1284	Controls how the IEEE-1284 parallel interface is multiplexed. Default is set to internal.
BSP_GPIO_MUX_IRQ_0-3	Controls how the IRQ0-IRQ3 signals are multiplexed. Defaults are set to internal.

Customization hook	Hardware features/default values set
BSP_GPIO_MUX_DMA_0-1	Controls how the external DMA0 and DMA1 signals are multiplexed. Defaults are set to internal.
BSP_GPIO_MUX_TIMER_0-15	Controls how the Timer 0 though Timer 15 signals are multiplexed. Defaults are set to internal.
BSP_GPIO_MUX_LCD	Controls how the LCD interface, not including the LCD Line End signal, is multiplexed. Default is set to multiplex to an 8-bit LCD application. For details about 8-bit LCD applications, see gpiomux_def.h.
BSP_GPIO_MUX_LCD_LINE_END	Controls how the LCD Line End signal is multiplexed. Default is set to multiplex the primary path. For details about the LCD Line End Primary Path, see gpiomux_def.h and the *NS9750 Hardware Reference*.
BSP_GPIO_MUX_SERIAL_A	Controls how the Serial Port A interface is multiplexed. Default is set to multiplex to an 8-wire serial application. For details about 8-wire serial applications, see gpiomux_def.h.
BSP_GPIO_MUX_SERIAL_B	Controls how the Serial Port B interface is multiplexed. Default is set to multiplex to a SPI serial application. For details about SPI serial applications, see gpiomux_def.h.
BSP_GPIO_MUX_SERIAL_C	Controls how the Serial Port C interface is multiplexed. Default is set to multiplex to a 4-wire serial application. For details about 4-wire serial applications, see gpiomux_def.h.
BSP_GPIO_MUX_SERIAL_D	Controls how the Serial Port D interface is multiplexed. Default is set to internal.
BSP_GPIO_INITIAL_STATE_PIN0-49	Controls the GPIO setup state when the multiplexed pin is set up as a GPIO. For information about setting up initial states of GPIO pins, see gpiomux_def.h.

mii.c file

The Ethernet PHY driver is located in the `mii.c` file in the `platforms` directory. If your hardware does not use a supported PHY, you must modify the driver to support it. Chapter 9, "Hardware Dependencies," describes the supported Ethernet PHYs.

For information about the routines in the `mii.c` file, see the online help.

customizeCache.c file

This file contains the `mmu` table, which determines the cache setup for each section of the processor's address map and the access level (read-only, read-write, or no-access) for each region.

You must update this table if:

- Your application uses a different amount of RAM or flash.
- Your application uses memory mapped devices.
- You want to change the cache mode or access level for a region.

For details about how to update `mmuTable`, see the online help.

pci.c file

This file contains `customizePCIStartup`, which is called by `pciVeryEarlyInitialization` and expects a return pointer to a `pci_init_t` structure that contains user-specific data needed for PCI configuration space.

You must customize the values in the returned `pci_init_t` structure to suit your application. For more information about the `pci_init_t` structure, see the `pci.h` public header file.

customizeButtons.c file

This file contains the `customizeReadPowerOnButtons` call, which can be used to sense external inputs at powerup. The initialization code can use this information to run special memory tests or system diagnostics.

customizeLed.c file

The `customizeLed.c` file contains the global data structure `NALedTable` table, which the NET+OS LED driver uses to determine how to turn LEDs on and off. The LEDs are connected to GPIO pins. For more information, see the section "gpio.h file" and the *NS9750 Hardware Reference* section about programming GPIO outputs.

customizeReset.c file

This file contains the `customizeRestart` and `customizeReset` functions.

These functions determine what the system should do in case of a reset or restart request. This is where application-specific code should be placed just before resetting the device.

Task 6: Modify the new BSP to start up the required drivers

You must configure the `bsp.h` file to enable the drivers that you want to run with your application. The default configuration works with a development board. Note that drivers that use the same GPIO pins cannot properly function at the same time.

Be sure to review the `bsp.h` file carefully.

USB device controller

The BSP is configured by default to support the USB device. You also must modify the development board to support this interface. For information about modifying the board, see the *NE9750 Hardware Reference*.

To disable the USB device, use either of these methods:

- **Recommended method**. Undefine `BSP_INCLUDE_USB_DRIVER` in the `bsp.h` file.
- **Alternate method**. Remove all USB driver entries from the device driver table in the `devices.c` file.

To test the USB device functionality, use the instructions in the *NS9750 Jumpers and Components Guide*. To modify the development board, see the `ReadMe` file in the `nausbdevapp` example.

The USB device example uses GPIO pin `17` to set up `pnp` functionality; your system uses this pin to detect whether the device is active and ready to receive commands.

1284 controller

The BSP is configured by default to disable support of the 1284 peripheral device. To enable the 1284 controller, use either of these methods:

- **Recommended method**. Define `BSP_INCLUDE_PARALLEL_DRIVER` in the `bsp.h` file.
- **Alternate method**. Add the 1284 driver entries from the device driver table in the `devices.c` file.

In addition, you must modify the development board to support the 1284 controller. For information about modifying the board to support this interface, see the *NE9750 Hardware Reference*.

See the instructions in the `ReadMe` file of the `naparaclient` example to set up all the necessary GPIO settings.

I2C controller

The BSP is configured by default to enable support of the I2C peripheral device. To disable the I2C controller, use either of these methods:

- **Recommended method**. Undefine `BSP_INCLUDE_ITC_DRIVER` in the `bsp.h` file.
- **Alternate method**. Remove the I2C driver entries from the device driver table in the `devices.c` file.

You do not need to modify any specific setting for the I2C device because the device has its own I/O lines.

LCD controller

The BSP is configured by default to enable support of the LCD peripheral devices. To disable the LCD controller, use either of these methods:

- **Recommended method**. Undefine `BSP_INCLUDE_LCD_DRIVER` in the `bsp.h` file.
- **Alternate method**. Remove the LCD driver entries from the device driver table in the `devices.c` file.

The LCD, timer, serial port C, serial port D, and 1284 share some of the GPIO pins; if you modify the LCD GPIO configuration, you must verify each GPIO pin setting.

PCI driver

The BSP is configured by default to enable support of the PCI peripheral device. To disable the PCI device driver, use either of these methods:

- **Recommended method**. Undefine `BSP_INCLUDE_PCI_DRIVER` in the `bsp.h` file.
- **Alternate method**. Remove the PCI driver entries from the device driver table in the `devices.c` file and enable code in the `BSP_INCLUDE_PCI_DRIVER` definition in the `ncc_init.c` file that disables the PCI module.

Serial ports

The BSP is designed to support four serial ports. In the standard NET+OS release, however, the BSP sets up one serial port to support asynchronous RS-232 style communications and one SPI interface.

To set a serial port to a mode other than those already set up by the standard NET+OS release (such as SPI or HDLC), modify the `gpio.h` file to ensure that correct GPIO pins are set to the correct value.

To disable the RS-232 serial peripheral interface controller, use either of these methods:

- **Recommended method**. Undefine `BSP_SERIAL_PORT_X` where *x* is `1`, `2`, `3`, or `4` in the `bsp.h` file.
- **Alternate method**. Remove the serial driver entries from the device driver table in the `devices.c` file.

You do not need to disable the serial driver to use the HDLC driver; however, in the `appconf.h` file for each example, you must set up the correct serial port number for each function.

Task 7: Modify the format of BSP arguments in NVRAM

The BSP stores some configuration arguments in NVRAM. The configuration values are read and written by way of customization hooks in `boardParams.c`.

You must modify these customization hooks to support your application:

Customization hook	Description
`customizeGetMACAddress`	Determines the Ethernet MAC address used to communicate on the network. Each device on the network needs a unique Ethernet MAC address. You must purchase a block of Ethernet MAC addresses from the IEEE and modify this routine to return an address from this block. The default implementation returns a value that was stored in NVRAM.
`customizeGetSerialNumber`	Returns the serial number for the unit. The serial number is used only in some sample applications and in the startup dialog. It is not used by the API libraries or in any part of the BSP except the dialog. If you rewrite the dialog, you can omit this routine. The default implementation returns a 9-character serial number read from NVRAM. Many developers use the Ethernet MAC address as the unit's serial number.

Customization hook	Description
customizeSaveSerialNumber	Sets the serial number for the unit. The serial number is used only in some sample applications and in the startup dialog. It is not used by the API libraries or in any part of the BSP except the dialog. If you rewrite the dialog, you can drop this routine. The default implementation stores a 9-character serial number in NVRAM.
customizeSetMACAddress	Sets the Ethernet MAC address for the unit. The default implementation stores the MAC address as a 6-byte array in NVRAM.
customizeUseDefaultParameters	Determines default configuration values and returns them in a buffer. The default implementation determines the default values through constants set in appconf.h. You must modify this routine to support your application.
customizeReadDevBoardParams	Reads the configuration from NVRAM into a buffer. You must modify this routine to support your application.
customizeWriteDevBoardParams	Writes the configuration to NVRAM. The default implementation accepts the current configuration as a buffer and writes the buffer into NVRAM.
customizeGetIPParameters	Reads the IP-related configuration values from NVRAM.
customizeSaveIPParameters	Writes the IP-related configuration values to NVRAM.

Task 8: Modify error and exception handlers

The errhndlr.c file in the platform directory contains customization hooks for an error handler and an exception handler.

Error handler

Code in the BSP calls the error handler, `customizeErrorHandler`, when fatal errors occur. Using constants in `bsp.h`, you can configure the default error handler to either:

- Report the error by blinking LEDs in a pattern.
- Reset the unit when a fatal error occurs.

You may need to modify the error handler if you want to report the error in some other way or take some other action.

Exception handler

The unexpected exception handler, `customizeExceptionHandler`, is called when these exceptions occur:

- Undefined instruction
- Software interrupt
- Prefetch abort
- Data abort
- Fast interrupt

Using constants in `bsp.h`, you can configure the exception handler to:

- Handle these exceptions by resetting the unit.
- Blink an error code on LEDs.
- Continue execution at the point at which the exception returned.

Digi does not recommend that you try to continue execution. You may need to modify the exception handler to better support your application.

Task 9: Verify the debugger initialization files

When you use the EPI MAJIC debugger, you must initialize hardware registers on the board that the BSP ROM startup code would normally set up. Debugger initialization scripts are set up as part of the installation procedure for NET+OS 6.1. The scripts contain commands that the debugger executes before the application is downloaded and executed.

The MDI server is the interface between the gdb debugger and the MAJIC. The debugger initialization script, `ns9750_a.cmd`, is located in the directory from which the MDI server is executed. During the installation procedure, you are prompted for the name of this directory. The MDI server reads this script when you start to download code to the board using `gdb`.

The next table shows the debugger initialization files that the MDI server uses:

File name	Contents
`startice.cmd`	The JTAG settings and reads in the `ns9750_amd` file
`ns9750_a.cmd`	The sequence of commands to initialize SDRAM
`epimdi.cfg`	MAJIC settings, including the network parameters

The debugger script initializes SDRAM and sets a bit in a register to indicate that the application is executing in the debugger.

If you are using a different type of SDRAM, you must modify the settings in the `ns9750_a.cmd` file. This file programs the registers in the memory controller. For a detailed description of these registers, see the *NS9750 Hardware Reference*.

Task 10: Debug the initialization code

After you complete the modifications and create the debugger initialization scripts for your application hardware, you need to debug the code.

To debug code from RAM, you use the EPI MAJIC and download the code through the gdb debugger into the RAM on your board. The next sections describe this procedure.

Preparing to debug the initialization code

Before you start debugging the initialization code, complete these tasks:

1. If you are using the MAJIC for the first time, verify its connection to the Ethernet:
   a. From either the bash shell or a DOS window, ping the IP address of the MAJIC:

      ```
      ping IP_ADDR
      ```

      where *IP_ADDR* is the IP address of the MAJIC.

   b. If you don't get a response, verify that the Ethernet cable is connected to the MAJIC and that the status light on the MAJIC is green.
2. Rebuild the BSP with your changes:
   a. Change to the BSP directory:

      ```
      cd src/bsp
      ```

   b. Enter this command:

      ```
      make PLATFORM=my_platform
      ```

      where *my_platform* is the name of your platform.

      The default platform is the `ns9750_a`.
3. Disable the POST by setting the `APP_POST` constant in the `root.c` file to `0`.
4. Carefully review all the settings in the `appconf.h` file. Make sure that `stdio` is directed to the correct serial port. The default is `com/0`.
5. Build the application:
   a. Copy the template application, which is located in `src/apps/template`.
   b. In the `src/apps/template/32b` directory, enter:

      ```
      make clean
      make all
      ```
6. Load the application with `gdb` or `gdbtk`.

   Digi recommends that you use the sample `gdbinit` file, which is provided in the `ns9750_a platform` directory. Copy the file into the directory that contains your `image.elf` (`src/apps/template/32b`).

7 Set up the debugger to view assembler instructions, and then step one instruction. This leaves the program counter (PC) at the beginning of the startup code.

8 Verify that the debugger initialization file has configured the application board such that:
   - The Chip Select registers for ROM and RAM are set up to support the parts and memory map.
   - You can read and write RAM on your application board.

9 Debug the initialization code by stepping through it, as described in the next section.

Debugging the initialization code

Debug the initialization code in stages, using the same order of the steps presented in this section:

1 `init.s` file

2 `nccInit` routine

3 `NABoardInit` routine

4 Ethernet driver startup

Note: This section describes debugging from RAM. You also need to step through the `init.s` code when it runs from ROM.

Debug the init.s file

The `init.s` file, located in `src/bsp/arm9init`, performs initialization functions. Step through the code in `init.s`, and verify that it works correctly. You usually do not need to change the code to support custom hardware boards.

The first function executed in NET+OS is the `Reset_Handler` routine in the `init.s` file. If your board is not working, you should set a breakpoint on the `Reset_Handler` routine and step through it.

For a description of `init.s`, see the section "ROM startup" in Chapter 1, "BSP Overview."

Debug the nccInit routine

The `nccInit` routine, located in `bsp/arm9init/ncc_init.c`, performs most of the board-specific hardware setup by calling a set of functions that you customize to support your board. After you customize these routines (described in Task 5), you need to check `nccInit` and your customized routines to verify that they are working correctly.

For a description of `nccInit`, see the section "nccInit routine" in Chapter 1, "BSP Overview."

If you have a problem starting the development board, you can use these diagnostic tools:

- A simple serial driver that is loaded in `nccInit`.
- A special `prinft` routine, `mprintf`. A protype of this routine is located in `h/ncc_init.h`. You can use `mprintf` to display diagnostic information before the serial driver is loaded in `netosStartup`.
- A `NETOS_DEBUG` flag, in `nccInit`. This flag can provide useful information.

Debug the NABoardInit routine

The `NABoardInit` routine, which is located in `src/bsp/arm9init`, provides some low level initialization routines for flash and NVRAM. Step through the initialization code in the `narmbrd.c` file to verify that the NVRAM APIs are initialized to support the NVRAM on your application hardware. You can configure the board to use a flash sector as NVRAM.

Debug the Ethernet driver startup

▶ **To debug the Ethernet driver startup:**

1. Put a breakpoint on the `eth_reset` routine (in `eth_reset.c`) and let the program run until you reach the breakpoint.
2. Step into the `customizeMiiReset` routine (in the `mii.c` file) and then into `customizeMiiIdentifyPhy`.
3. Verify that `customizeMiiIdentifyPhy` returns a value not equal to `0xffff`. Verify that `mii_reset` returns `0` and that `customizeMiiIdentifyPhy` identifies the PHY on your application hardware.

4. Step into `customizeMiiNegotiate` and verify that `customizeMiiCheckSpeed` determines whether you are connected to a 100 Base-T network.
5. Step into `customizeMiiCheckDuplex` to determine whether you have a full- or half-duplex link.

Task 11: Modify the startup dialog

The BSP prompts you to change configuration settings after a reset. The dialog implemented for the development boards prompts you to set the board's serial number, Ethernet MAC address, and IP networking parameters. The dialog code is in the `dialog.c` file in the `platforms` directory.

If you plan to use the dialog in your product, change it to support your application. The `customizeDialog` function calls the `NAGetAppDialogPort`, `NAOpenDialog`, and `NACloseDialog` functions to determine which port to use for the dialog and to open and close it.

If you do not want a dialog, replace the code in `dialog.c` with an empty version of `customizeDialog` that just returns.

Generally, you do not need to customize these functions. To support your application, however, you usually need to completely rewrite the other functions called by `customizeDialog` to display the current configuration settings and prompt. The I/O port for the dialog is set by the `APP_DIALOG_PORT` constant in your application's `appconf.h` file.

Task 12: Modify the POST

If the `APP_POST` constant is set, the BSP automatically runs the POST from the `main.c`, which is located in `src/bsp/common`.

The POST routines that ship with NET+OS test the NS9750 processor. You may want to create other POST routines that test additional hardware on your board.

Task 13: Modify the ACE

The Address Configuration Executive (ACE) is an API that runs at startup to acquire an IP address.

You need to customize the contents of two files in the `platform` directory – `aceCallbacks.c` and `aceParams.c` – that contain information the ACE uses.

aceCallbacks.c

The `aceCallbacks.c` file contains a set of callback functions that the ACE invokes at different points in the startup process. You need to customize these callbacks for your application.

For example, the `customizeAceLostAddress` routine is called when the lease for an IP address has expired. The default implementation resets the unit. You could customize `customizeAceLostAddress` to notify your application of the problem so that your application can try to recover by closing and restarting network connections. (For details about these functions, see the online help.)

aceParams.c

The `aceParams.c` file contains the code that reads and writes ACE configuration information in NVRAM. Generally, the only parts of the `aceParams.c` file you need to customize are these definitions:

- **The dhcp_desired_params array**. Contains a list of the Dynamic Host Configuration Protocol (DHCP) options that you want the client to request from the server.

  Add any other DHCP options you want the client to request from the server.

- **NADefaultEthInterfaceConfig**. Contains the configuration that ACE uses if none is stored in NVRAM. This configuration controls which protocols are used to get an IP address and the options used with them. The default configuration uses all protocols to get an IP address. Customize this configuration as needed.

Device Drivers

CHAPTER 4

This chapter provides information about device drivers and device definition.

Overview

NET+OS integrates device drivers with the low-level I/O functions provided in the Cygwin standard C library. Each entry in the `deviceTable` array of the `devices.c` file defines a device that the system supports.

The rest of this chapter describes the `deviceTable` array and the device driver functions.

Adding devices

To add a device, you add an entry to the `deviceTable` array. Application software can then access the device through the standard C programming language I/O routines — `open`, `read`, `write`, `ioctl`, and `close`.

deviceInfo structure

The entries in `deviceTable` are `deviceInfo` structures. The `ddi.h` file defines the `deviceInfo` structure. The fields in this structure define the device driver's interface to NET+OS.

The `deviceInfo` structure is defined as shown here:

```
typedef struct
{
          char *name;
          int channel;
          devEnterFnType *deviceEnter;
          devInitFnType *deviceInit;
          devOpenFnType *deviceOpen;
          devCloseFnType *deviceClose;
          devReadFnType *deviceRead;
          devWriteFnType; *deviceWrite;
          devIoctlFnType *deviceIoctl;
          unsigned flags;
} deviceInfo;
```

This table defines the fields in the `deviceInfo` structure:

Field	Description
name	Pointer to a null-terminated string that is the device channel's name. The name must be unique for each device.
channel	Channel number for the device name. This number is passed to the device driver for all I/O requests.
deviceEnter	Pointer to the driver's first-level initialization routine for the channel. `DDIFirstLevelInitialization` calls this routine once, during initialization, when the C library initializes its I/O library. Kernel services are not available at this point.
deviceInit	Pointer to the driver's second-level initialization routine for the channel. `DDISecondLevelInitialization` calls this routine once, at startup, after the kernel has been loaded.
deviceOpen	Pointer to the device's `open` routine for the channel. This routine is called whenever an application opens the channel to indicate that a new session is starting. The *flags* field indicates whether the channel: ■ Was opened for read, write, or read/write mode ■ Operates in blocking or non-blocking mode
deviceClose	Pointer to the driver's `close` routine for the channel. This routine is called at the end of every session.
deviceRead	Pointer to the driver's `read` routine for the channel.
deviceWrite	Pointer to the driver's `write` routine for the channel.
deviceIoctl	Pointer to the driver's I/O control routine for the channel.
flags	Bit field that indicates which bits are valid in the *flags* field of an open call to the device. A bit set in this field indicates that the bit also can be set in the driver's `open` routine.

Device driver functions

This table provides a summary of the device driver functions in the `deviceInfo` structure. The next sections describe each function. For details, see the online help.

Function	Description
`deviceEnter`	First-level initialization function for a device table
`deviceInit`	Second initialization function for the device channel
`deviceOpen`	Informs the device driver that a new session is starting on the channel and which I/O mode will be used during the session
`deviceClose`	Informs the device driver that the application is closing its session
`deviceRead`	Reads data from the device to the caller's buffer
`deviceWrite`	Writes a buffer of data to a device
`deviceIoctl`	Sends commands to the device

The return values for the functions are in a table in the section "Return values," later in this chapter.

deviceEnter

First-level initialization function for a device table.

When the C library initializes its I/O functions, `deviceEnter` is called for each entry in the device table. This routine is called only once for each channel and performs the basic initialization that the device driver needs.

Because this routine is called before the kernel has started, kernel services are not available at this time. C library functions, however, are available.

Format

```
int deviceEnter (int channel);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry

For this routine's return values, see the table in the section "Return values."

deviceInit

Second initialization routine for the device channel.

After the kernel has loaded, the device driver table is scanned, and the `deviceInit` routines for each channel are called. The `deviceInit` routine is called once for each channel and completes any additional initialization needs for the device driver. Kernel services are available, and interrupts are enabled.

Format

```
int deviceInit (int channel);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry

For this routine's return values, see the table in the section "Return values."

deviceOpen

Notifies the device driver that a new session is starting on the channel and tells the driver which I/O mode will be used during the session. This routine is called when the application calls the `open` system call.

When `deviceOpen` is called, the driver performs these steps:

1. Checks that the channel number is valid, the channel is open, and the flags are appropriate.

   If an error condition is detected, the driver returns an error without sending any information.
2. Sets an internal flag to indicate that a session is in progress on the channel.
3. Performs any other initialization tasks required by the device.
4. Returns `a` value.

Format

```
int deviceOpen (int channel, unsigned flags);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry
flags	Bit field formed by ORing together one or more of these values: ■ O_RDONLY ■ O_WRONLY ■ O_RDWR ■ O_NONBLOCK

For this routine's return values, see the table in the section "Return values."

deviceClose

Informs the device driver that the application is closing its session. This routine is called when the application calls the `close` system call.

When `deviceClose` is called, the driver performs these steps:

1. Checks that the channel is open and the configuration is valid for the device.

   If an error condition is detected, the driver returns an error without sending any information.
2. Either sets the channel semaphore or returns `EBUSY` if the semaphore is already set.
3. Updates internal flags to indicate that the session has been closed.
4. Performs any other processing tasks as necessary.
5. Clears the channel semaphore.
6. Returns `EXIT_SUCCESS`.

Format

```
int deviceClose (int channel);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry

For this routine's return values, see the table in the section "Return values."

deviceRead

Reads data from the device to the caller's buffer. This routine is called when the application calls the `read` system call.

When `deviceRead` is called, the driver performs these steps:

1. Sets *`bytesRead`* to `0`.
2. Checks that the arguments are correct and the channel is open.
3. Checks for a pending error on the device.

   If an error condition is detected, the driver returns an error without transferring any data.
4. Either sets the channel semaphore or returns `EBUSY` if the semaphore already is set.
5. If no data is available, performs one of these steps:
   - **Blocking mode.** Waits until some data is received.

     If an error condition is detected, the driver aborts the transmission and returns an appropriate completion code.
   - **Non-blocking mode.** Releases the semaphore and returns `EAGAIN`.
6. Copies the data from the driver buffers until either all the data has been copied or the caller's buffer has been filled.
7. Updates *`bytesRead`*.
8. Releases the channel semaphore.
9. Returns a completion code.

Format

```
int deviceRead (int channel, void *buffer, int length,
         int *bytesRead);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry
buffer	Pointer to caller's receive buffer
length	Length of caller's receive buffer (number of bytes)
bytesRead	Pointer to the number of bytes actually read

For this routine's return values, see the table in the section "Return values."

deviceWrite

Writes a buffer of data to a device. This routine is called when the application calls the `write` system call.

When `deviceWrite` is called, the driver performs these steps:

1. Sets *`bytesWritten`* to `0`.
2. Checks that the arguments are correct and the channel is open.
3. Checks for a pending error on the device.

   If an error condition is detected, the driver returns an error without transferring any data.
4. Either sets the channel semaphore or returns `EBUSY` if the semaphore already is set.
5. Opens a transmit buffer and fills it with data from the caller's buffer.
6. Starts the transmit operation for the transmit buffer.
7. *This step applies to blocking mode only.* If an error condition is detected, aborts the transmission and returns an appropriate completion code.
8. If there is more data in the caller's buffer, repeats steps 5 through 7 until there is no more data.
9. Updates *`bytesWritten`* to indicate the number of bytes transmitted.
10. Releases the channel semaphore.
11. Returns a completion code.

Format

```
int deviceWrite (int channel, void *buffer, int length,
         int *bytesWritten);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry
buffer	Pointer to caller's buffer; not necessarily aligned
length	Length of caller's receive buffer (number of bytes)
bytesWritten	Pointer to `int` to load with number of bytes actually written

For this routine's return values, see the table in the section "Return values."

deviceIoctl

Sends commands to the device. This routine is called when the application calls the `ioctl` system call.

When `deviceIoctl` is called, the driver performs these steps:

1. Checks that the arguments are correct and that the channel is open.

   If an error condition is detected, the driver returns an error without sending any commands.
2. Either sets the channel semaphore or returns `EBUSY` if the semaphore is already set.
3. Executes the command.
4. Releases the channel semaphore.
5. Returns `EXIT_SUCCESS`.

Format

```
int deviceIoctl (int channel, int request, char *arg);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry
request	Commands encoded as integers
arg	Pointer to any extra information needed or to a buffer to return information

You can define your own return values.

For this routine's return values, see the table in the next section "Return values."

Return values

The NET+OS low level device driver interface (DDI) routines map to the DDI application layer calls as shown in this table:

DDI routine	DDI application layer call
deviceOpen	open
deviceClose	close
deviceIoctl	ioctl
deviceRead	read
deviceWrite	write

All the DDI functions return 0 on success and an error number value otherwise. The C library interprets this value and passes it up to the application that is calling the functions.

The application return values fall into one of two categories:

- **Data passing functions.** The read and write function calls.
- **Setup functions.** The open, close, and ioctl function calls.

The deviceRead and deviceWrite data passing functions use the arguments **bytesRead* and **bytesWritten*, respectively, to pass the data size information back to the application read and write function calls. The application call returns the data size if the low level function succeeds.

For example, if deviceRead returns 0, and the **bytesRead* argument is set to 100, the read function returns 100. Alternatively, when deviceRead returns a non-zero, the read function returns -1 regardless of what's loaded into the **bytesRead* argument.

The setup functions are similar, but they do not communicate any data size up. When a DDI function succeeds (for example, deviceIoctl returns 0), the application function also returns 0 (in this case ioctl returns 0). Alternatively, when deviceIoctl returns a non-zero, the ioctl function returns -1.

When any low level DDI function returns a non-zero value, the value is loaded into the system error numbers and causes the application layer call to return -1. System error numbers can be checked by a call to getErrno.

Values and definitions for error numbers are in the `errno.h` system error header file. The system error header file is in the `/cygwin/user/arm-elf/include/sys` folder.

The next table includes common error number return values with a typical description. In general, the values that are returned are specific to the driver that is being accessed. For more information, see the online help for the driver.

Value	Description
EBUSY	Device is busy.
EINVAL	Invalid argument.
ENOENT	No such file or directory.
EAGAIN	Unable to complete operation now; try again later.
EBADF	Bad file number.
EIO	I/O error.
ENOMEM	Out of memory.
EROFS	Read-only file system.
ENXIO	Invalid device.
ETIMEDOUT	Operation timed out.
ERANGE	An argument has an invalid range.
EACCESS	Permission denied.
EFAULT	Bad address.
ENOSPC	No space available on device.
ENODEV	No such device.
ENOMEM	Memory allocation failure.
EXIT_SUCCESS	Call completed successfully.

Modifications to Cygwin's standard C library and startup file

The standard C library has been rebuilt to support the NET+OS DDI. A customized version of the startup files and C libraries is in the `C:/NETOS61_GNU/lib/32b/gnu` directory. All the sample applications link to these files instead of to the standard GNU versions.

To use the NET+OS device drivers and the ThreadX kernel, you must make your applications link to these files. For an example, see either of the `makefile`s supplied in the sample applications or the GNU Tools linker documentation.

You can find all the necessary changes to the C library's source code and the `crt0.S` startup file in the `C:/NETOS61_GNU/gnusrc` directory.

Note: The C library that is shipped with NET+OS is not re-entrant. For more information, see your GNU Tools documentation.

Modifying the libc.a library and crt0.o startup file

The NET+OS version of the source file is in the `gnusrc` directory.

▶ **To modify the libc.a and crt0.o files:**

1. Copy `cygwin/usr/arm-elf/lib/be/libc.a` to the `C:/NETOS61_GNU/gnusrc` directory.
2. To open a GNU X-Tools shell, enter this command:

   ```
   xtools arm-elf
   ```
3. To produce the new `libc.a` and a new `crt0.o` file to support NET+OS I/O devices, change to the `C:/NETOS61_GNU/gnusrc` directory and enter:

   ```
   make all
   ```
4. Copy `gnusrc/libc.a` and a new `crt0.o` to the `C:/NETOS61_GNU/lib/32b/gnu` directory.

   Note that the `crtbegin.o`, `crtend.o`, `crti.o`, and `crtn.o` files in the `C:/NETOS61_GNU/lib/32b/gnu` directory are copied from `/cygwin/use/lib/gcc-lib/arm-elf/3.2/be`.

   Because these startup files are for C++ applications, you do not need to modify them.

Updating Flash Support

CHAPTER 5

This chapter describes how to update flash memory.

Overview

NET+OS includes application program interface (API) functions for reading, writing, and erasing flash memory. The internals of the flash memory API rely on `flash_id_table` in the `naflash.c` file, (located in `C:/NETOS61_GNU/src/flash`) to define the known flash parts. The flash API is guaranteed to function only with parts that are defined in the `flash_id_table`. If the part is not recognized, you need to update the `flash_id_table`.

The rest of this chapter describes the `flash_id_table` and the procedures for updating flash. For details about the flash API functions, see the online help.

NET+OS 6.1 supports these flash ROM parts:

Manufacturer	Part numbers
AMD	AM29LU16
ST Micro	M29W320DB

Flash table data structure

The `flash_id_table_t` data structure, defined in the `flash.h` file, is shown here. The tables that follow the code list the structure's data types and fields.

```
typedef struct
{
                              WORD8   ccode;
                              WORD32  ccode_addr;
} flash_cmd_t;

typedef struct
{
                              WORD16 mcode;
                              WORD16 mcode_addr;
                              WORD16 dcode;
                              WORD16 dcode_addr;
                              WORD16 total_sector_number;
                              WORD32 sector_size;
```

```
                                WORD16 prog_size;
                                WORD16 access_time;
                                flash_cmd_t *id_enter_cmd;
                                WORD16 id_enter_len;
                                flash_cmd_t *id_exit_cmd;
                                WORD16 id_exit_len;
                                flash_cmd_t *erase_cmd;
                                WORD16 erase_len;
                                flash_cmd_t *write_cmd;
                                WORD16 write_len;
                                flash_cmd_t *sector_erase_cmd;
                                WORD 32 *sector_size_array;
} flash_id_table_t;
```

This table lists the data types used in the `flash_id_table_t` structure:

Data type	Description
`WORD8`	Unsigned byte
`WORD16`	Unsigned short
`WORD32`	Unsigned long

This table summarizes the fields in the `flash_id_table_t` data structure:

Field	Description
mcode	Manufacturer's code
mcode_addr	Address of manufacturer's code
dcode	Device code
dcode_addr	Address of device code
total_sector_number	Total number of sectors
sector_size	Size of sector (in bytes)
prog_size	Program load size (in bytes)
access_time	Access time (in nanoseconds)
id_enter_cmd	Pointer to the enter identify flash command
id_enter_len	Number of cycles for the enter identify flash command

Field	Description
id_exit_cmd	Pointer to the exit identify flash command
id_exit_len	Number of cycles for the exit identify flash command
erase_cmd	Pointer to the erase flash command
erase_len	Number of cycles for the erase flash command
write_cmd	Pointer to the write flash command
write_len	Number of cycles for the write flash command
sector_erase_cmd	For AMD only
sector_size_array	For non-uniform sector sizes

Adding new flash

When you add support for new flash ROM, you need to provide definitions for the new flash device, such as the number of flash sectors, the flash sector size, and the program load size. You also need to modify the ROM type value in the `flash_id_table` definition.

For example, to add support for ST Micro M29W800AB flash ROM, you would edit the `flash.h` file as shown here:

```
/* ST Micro M29W800AB*/
#define STM_M29W800AB_FLASH_SECTORS 0x013U
/* We are using block instead of sector */
#define STM_M29W800AB_FLASH_SECTOR_SIZE VARIABLE_SECTOR_SIZE
#define STM_M29W800AB_PROG_SECTOR_SIZE 0x0002U
```

▶ **To add support for new flash ROM:**

1. In the `flash.h` file, add the definitions for the new flash device.
2. (*Optional step for keeping track of devices supported.*) In `flash.h`, modify the ROM type value; for example:

   ```
   #define STM_29W800AB 0x0D
   ```

3 Edit the `naflash.c` file and modify the `flash_id_table` definition. Add the new flash part entries to the start of the table to allow faster software identification of the flash part.

4 Modify other command sequences such as `id_enter_cmd`, `id_exit_cmd`, and so on.

See the documentation supplied by the manufacturer of the flash device you are using.

To rebuild the driver, use `flash.mak` in the `C:/NETOS61_GNU/` directory to rebuild the flash library in the top-level directory. Rebuilding this library rebuilds the flash driver.

For example, for the STM_29W800AB, add this entry to the end of the `flash_id_table`, based on the ROM type value defined in step 2:

```
{
0x20, 0x00, 0x005B, 0x01, STM_M29W800AB_FLASH_SECTORS,
VARIABLE_SECTOR_SIZE, STM_M29W800AB_PROG_SECTOR_SIZE, 120,
(flash_cmd_t *)STM_M29W800AB_flash_id_enter_cmd,
sizeof(STM_M29W800AB_flash_id_enter_cmd) / sizeof(flash_cmd_t), (flash_cmd_t *),
STM_M29W800AB_flash_id_exit_cmd,
sizeof(STM_M29W800AB_flash_id_exit_cmd) / sizeof(flash_cmd_t),(flash_cmd_t *),
STM_M29W800AB_flash_erase_cmd,
sizeof(STM_M29W800AB_flash_erase_cmd) / sizeof(flash_cmd_t),
(flash_cmd_t *)STM_M29W800AB_flash_write_cmd,
sizeof(STM_M29W800AB_flash_write_cmd) / sizeof(flash_cmd_t),
(flash_cmd_t *)STM_M29W800AB_flash_block_erase_cmd,
(WORD32*) STM_M29W800AB_flash_block_size_array
}
```

This table shows the definitions for the values in the example:

Value	Definition
0x20	Manufacturer's code
0x00	Address of manufacturer's code
0x005B	Device code
0x01	Address of device code

Supporting larger flash

If you are adding larger flash, you need to perform additional steps, described next.

▶ **To support larger flash configurations:**

1. Increase these three constants in `flash.h`:
   - `MAX_SECTORS`—The maximum number of flash sectors supported
   - `MAX_SECTOR_SIZE`—The maximum sector size supported
   - `MAX_FLASH_BANKS`—The maximum number of flash banks supported
2. Rebuild the flash library in the top-level directory, using `flash.mak`.

Bootloader Utility Overview

CHAPTER 6

This chapter describes the `bootloader`, a utility you use to recover from a failed download of new firmware and to decompress an application in ROM to run from RAM.

Overview

To recover after a flash download of new firmware fails, you use the `bootloader`. When the download fails, the `bootloader` automatically downloads a new image from a network server.

The `bootloader` runs from ROM and links in an image that is copied to RAM and executed. The image may be compressed to save ROM space. In normal operation, the RAM image verifies that the application image stored in flash is correct, decompresses it to RAM, and executes it. The application image also has a boot image header, which determines where, in RAM, to decompress it.

Digi recommends that you use the `bootloader` to run your application.

The `bootloader` utility consists of two application images:

- **ROM image**. A small application that runs from ROM
- **RAM image**. Your large application, which runs from RAM.

  The RAM image verifies that the application image stored in flash is correct, decompresses it to RAM, and executes it.

The rest of this chapter describes these images and provides details about how the `bootloader` utility functions.

Bootloader application images

This section provides a description of the ROM and RAM application images that the `bootloader` utility uses.

ROM image

The ROM image is located in the first (and possibly the second) sector of flash. The processor automatically starts to execute code from the beginning of flash after a reset, and so immediately starts to execute the bootloader ROM image. The `bootloader` uses the BSP initialization code to configure the hardware.

The ROM image initializes the hardware. After the hardware is initialized, the ROM image decompresses the RAM image section of the `bootloader` to RAM and executes it.

RAM image

The RAM image is stored as an application image in flash. Like other applications, the RAM image has a boot image header. Information in the header determines where, in RAM, to decompress the image. The RAM image runs after it is decompressed to RAM.

The RAM image has these requirements:

- Sufficient RAM must be available to hold the RAM image portion of the `bootloader` (about 128 KB), the compressed application image downloaded from the network, and the decompressed version of the application image.

  The maximum sizes of both the compressed and decompressed versions of the application image are set in the linker script customization file.
- The application image must be built with the `boothdr` utility, which is located in `/bin`.

If the application image fails the checksum test, the RAM image attempts to recover by:

- Downloading a replacement for it using `TFTP`
- Using the DHCP/BPPTP server to get the network/file name to download information

The RAM image uses these steps to perform the recovery:

1. Initializes the Ethernet driver.
2. Initializes the UDP stack.
3. Downloads the application image from a network server to RAM.
4. Validates the downloaded application image by performing a `CRC32` checksum.

5 Stores the image into flash.

6 Resets the unit, which restarts the process.

The application image, which this procedure replaces, passes the checksum test and is executed.

Application image structure

An application image consists of:

- An application image header, which has two parts:
  - A NET+OS header
  - An optional custom header
- The application itself
- A checksum, which is computed over the entire image, including the headers

The next section describes each component of the application image header.

Application image header

The application image header has two sections of variable length. The first part contains data that the `bootloader` uses, and the second part contains application-specific data that you define. Fields at the start of a section determine the size of the two sections.

This data structure defines the application image header:

```
typedef struct
{
    WORD32 headerSize;
    WORD32 naHeaderSize;
    char signature[8];
    WORD32 version;
    WORD32 flags;
    WORD32 flashAddress;
    WORD32 ramAddress;
    WORD32 size;
    } blImageHeaderType;
```

This table describes how the fields are used:

Field	Description
headerSize	Set to indicate the size of the complete header, including the application-specific section. The application starts immediately after the end of the header.
naHeaderSize	Set to indicate the size of the NET+OS portion of the image header in bytes, including this field.
signature	Set to the ASCII string `bootHdr` to identify this header as a valid image header.
version	Set to `0` for this version of the image header.
flags	A bit field of flags. See the next table for details about bit values.
flashAddress	If the image is to be written to flash, set this field to the address to which the image will be written. The entire image, including the header, is written to flash.
ramAddress	Holds the image's destination address in RAM. When an image is written to RAM to be executed, only the application part of the image, without the header, is written.
size	Holds the size of the image (not including the header) in bytes.

These bit values are defined for the *flags* field:

Bit value	Description
BL_WRITE_TO_FLASH	If this bit is set, the image is written to the address in flash specified in the *flashAddress* field. If this bit is clear, the image is run immediately without writing it to flash. The image is moved or decompressed to the address in the *ramAddress* field before it is executed.
BL_LZSS_COMPRESSED	If this bit is set, the application portion of the image is compressed. It is decompressed to the address in the *ramAddress* field before it is executed.
BL_EXECUTE_FROM_ROM	If this bit is set, the application is executed from ROM. The application must not be compressed. If this bit is not set, the application is decompressed or moved to the address in the *ramAddress* field before it is executed.

boothdr utility

The `boothdr` utility converts a binary image into an application image by:

1. Inserting a header at the beginning of the image.

   The data to place inside the header is read from a configuration file.

2. Inserting a customer header.

   You specify this action at the command line by providing the name of a file that contains the custom header.

3. Calculating a CRC32 checksum for the entire image, including the header, and placing it at the end of the file.

The `boothdr` utility takes this command line:

Format

`boothdr` *`config-file input-file output-file`* [*`custom-header-file`*]

Arguments

Argument	Description
`config-file`	The name of the configuration file
`input-file`	The name of the `bin` file to convert
`output-file`	The name of the file to create
`custom-header-file`	The name of a file that contains your custom header as binary data

Generating an image

The template and sample makefiles in the apps and examples directories use these steps to create application images when you build an application:

1 The makefile is compiled and linked.

The application is linked for its execution address in RAM (image.bin) or ROM (rom.bin), but is linked as a ROM application. Normally, this image is set up for debugging.

2 The compression program that ships with NET+OS compresses the image.

3 The bootldr creates an application image that the bootloader supports.

Configuration file

The configuration file contains configuration information in the form of several keyword/value pairs. The default configuration file, bootldr.dat, is stored in the src/linkerScripts directory.

This table describes the keyword/value pairs:

Keyword	Value description
WriteToFlash	Set to one of these options: ■ Yes. Sets the BL_WRITE_TO_FLASH bit in the *flags* field of the header. ■ No. The bit is left clear.
Compressed	Set to one of these options: ■ Yes. Sets the BL_LZSS_COMPRESSED bit in the *flags* field of the header. ■ No. The bit is left clear.
ExecutedFromRom	Set to one of these options: ■ Yes. Sets the BL_EXECUTE_FROM_ROM bit in the *flags* field of the header. ■ No. The bit is left clear.
FlashOffset	Specifies the offset from the beginning of flash where the image is to be written. Set to a hexadecimal value preceded by 0x.

Keyword	Value description
RamAddress	Specifies the absolute address in RAM at which to execute the application. The application is copied or decompressed to this location. Set to a hexadecimal value preceded by 0x.
MaxFileSize	Specifies the maximum size of the image in bytes. The application terminates in error if the combination of the image, header, and checksum is larger than this value. Set to a hexadecimal value preceded by 0x.

Here is an example of a configuration file that uses keyword/value pairs:

```
WriteToFlash        Yes
Compressed          Yes
ExecuteFromRom      No
FlashOffset         0x10000
RamAddress          0x8004000
MaxFileSize         0xD0000
```

General bootloader limitations

Keep in mind these general limitations about the `bootloader`:

- The `bootloader's` DHCP/BOOTP client is limited. The client supports options for getting the IP address, subnet mask, gateway address, boot image filename, and boot image size only. You cannot use the client to get other options.
- The `bootloader's` User Datagram Protocol (UDP) stack supports a limited implementation of UDP and IP that supports only those features needed to support DHCP/BOOTP and Trivial FTP (TFTP).
- The TFTP client supports only file downloads.

The TFTP server and the DHCP/BOOTP server must be located on the same machine (that is, must have the same IP address).

Customizing the Bootloader Utility

CHAPTER 7

This chapter describes how to customize the `bootloader` utility for your applications. You use this utility to recover from a failed download of new firmware.

Overview

You can modify a set of functions in the default `bootloader` to support your specific applications and environments. These functions, referred to as *customization hooks*, are in the `blmain.c` and `blerror.c` files in the `platforms` directory.

The code in `blmain.c` is like a template `bootloader`. If the current application image is corrupt, the code uses the `bootloader` application program interface (API) to download a new application image. To add new functionality to the `bootloader`, you modify the template.

The rest of the chapter describes the functions in the `blmain.c` file. For details about each function, see the online help.

Customization hooks

This table provides a summary of the functions in the `blmain.c` file, which is in the `platforms` directory:

Function	Description
`NABlReportError`	Called whenever an error occurs
`getMacAddress`	Gets the Ethernet MAC address that the `bootloader` should use
`isImageValid`	Determines whether an image is valid
`shouldDownloadImage`	Determines whether the `bootloader` should download a new image
`getDefaultFilename`	Determines the name of the file to download
`downloadImage`	Downloads a new application image

NABlReportError

Called when an error is detected.

The error is reported to the user.

Format

```
void NABlReportError (errorCode);
```

Arguments

Argument	Description
errorCode	Identifies the error type

Return values

None

Implementation

The default implementation reports an error by blinking the LEDs on the development board in a pattern and then returns. The `errorCode` value determines the pattern.

Because this implementation relies on hardware (LEDs) that may not be present on customer boards, it is valid for only the NET+ARM development board.

You can customize the function in a number of ways, depending on the features in the target hardware; for example, by:

- Writing an error message out the serial port
- Blinking the LEDs in a loop, which effectively forces users to reset the device manually after correcting the problem

getMacAddress

Returns a pointer to the Ethernet MAC address that the `bootloader` uses.

Format

```
char *getMacAddress;
```

Arguments

None

Return values

Returns the Ethernet MAC address as an array of characters

Implementation

The default implementation uses the `customizeGetMACAddress` function to read the Ethernet MAC address from NVRAM. You can use the default implementation if the `customizeGetMACAddress` function has been ported to the application hardware.

You may need to modify the default implementation if you want to get the MAC address in a different way. Do not hard-code the MAC address; doing so prevents more than one unit from operating on the network.

isImageValid

Determines whether a downloaded image is valid.

Format

```
int isImageValid (blImageInfoType *imageInfo)
```

Arguments

Value	Description
imageInfo	Pointer to the image header

Return values

Value	Description
TRUE	Image is valid.
FALSE	Image is not valid.

Implementation

The default implementation validates the image by checking the signature in the header and performing a cyclic redundancy check (CRC) on the image.

You should extend the default implementation to determine whether the application can and should be run on the hardware; for example, by:

- Encoding information in the custom section of the image header that identifies the application's hardware requirements and features.
- Encoding the hardware capabilities into the GEN_ID and GPIO bits.
- Verifying that the hardware has the features needed to run the application.
- Verifying that the end user is allowed to run the application on this unit; in other words, making sure the user is not trying to upgrade a low-end unit with the firmware for a high-end unit.
- If the application is to be written into flash, verifying that it fits.
- Verifying that the destination address specified in the image header is valid.

shouldDownloadImage

Determines whether to download an application image from the network.

Format

```
int shouldDownloadImage(void);
```

Arguments

None

Return values

Value	Description
TRUE	Downloads the image from the network.
FALSE	Executes the image in flash.

Implementation

To help debug the bootloader, the default implementation returns TRUE if the image is invalid.

```
BOOLEAN shouldDownloadImage(void)
{
           int result = TRUE;
           blImageHeaderType *imageInfo = (blImageHeaderType *)
           BSP_APPLICATION_ADDRESS;
           result = (isImageValid(imageInfo) == FALSE);
           return result;
}
```

You may want the bootloader to download a new image even if the current image is valid. For example, you may want to let end users force a download by either pushing a button at powerup or selecting an option from a configuration menu.

getDefaultFilename

The Dynamic Host Configuration Protocol (DHCP) client gets the name of the application image from the DHCP or Bootstrap Protocol (BOOTP) server. The client can pass the server the name of the file when the server requests this information, allowing the server to determine which file is appropriate for the client.

How the server uses the information depends on the implementation. If no filename is specified, the server returns the name of the default image file.

This function sets the name of the file that is passed to the DHCP/BOOTP server. The function returns a zero-length string if it wants the default file.

Format

```
char *getDefaultFilename(void);
```

Arguments

None

Return values

A null-terminated ASCII string that is the name of the file that the DHCP client will request from the DHCP/BOOTP server

Implementation

The default implementation returns a pointer to an empty string, which has the effect of requesting the default boot image on the Trivial File Transfer Protocol (TFTP) server.

You will probably want to modify the default implementation to pass a filename to the DHCP/BOOTP server. Some possibilities are:

- Hard-coding a filename that identifies the product
- Determining the features supported by the hardware and generating a filename that has this information encoded in it
- Generating a filename that identifies the features purchased by the user

downloadImage

Downloads an application image from the network into a memory buffer.

Format

```
int downloadImage (char *destination, int maxLength)
```

Arguments

Argument	Description
destination	Pointer to the memory buffer that will hold the image
maxLength	Size of the memory buffer in bytes

Return values

Return value	Description
BL_SUCCESS	Image successfully downloaded
otherwise	Error code that identifies the failure

Implementation

The default implementation uses DHCP to get an IP address and TFTP to download load the image. After the image is downloaded, it is validated.

You can use the default implementation in many applications. For example, you may want to extend the default implementation by:

- Using information in NVRAM to determine:
  - The unit's IP address
  - The IP address of the TFTP server
  - The name of the application image to download
- Passing a vendor class identifier (option 60) to the DHCP server
- Receiving vendor information (option 43) from the DHCP server
- Downloading the image over a serial or parallel port

Linker Files

CHAPTER 8

This chapter describes the linker files that are provided for sample projects.

Overview

The GNU linker combines one or more object modules into a single executable output module. Executable programs are divided into several sections that contain the code and data parts of the application. Commands in the linker files that are supplied with NET+OS determine where to map the sections of applications in memory.

The linker must position sections in an application where actual ROM and RAM will reside. Therefore, the linker file that is used to create images that execute from flash ROM is different from the one that executes from RAM.

The rest of this chapter describes the linker files for the sample projects.

For more information about the GNU linker, see your GNU Tools documentation.

Linker files provided for sample projects

Linker files are provided in `src/linkerScripts` for each sample project. Most projects use the `image.ldr` and `rom.ldr` linker files to create applications that execute from RAM or ROM respectively. Projects that use the deprecated HTTP server use different linker files that link the application against the older version of the library. These linker scripts are generated when applications are built.

The source files for the linker scripts are stored in the `platforms` directory and in the `C:/NETOS61_GNU/bsp/arm9init` directory.

These linker files are provided:

Linker file	Description
`customize.ldr`	The customization file for linker scripts
`httpImage.ldr`	Creates an executable file for debugging and an `image.bin` file for applications that use the deprecated HTTP server
`httpRom.ldr`	Creates an executable file for the `rom.bin` image for applications that use the deprecated HTTP server

Linker file	Description
image.ldr	Generates an executable file for debugging and for the image.bin image
rom.ldr	Generates an executable file for the rom.bin image

Some sections in applications are defined for all GNU applications, and some are specific to NET+OS.

Basic GNU Tools section of the linker files

This table summarizes the GNU Tools section of the linker files:

Section	Description
init	Vector code section
text	Text section, including code
data	Initialized writeable data section
bss	Zeroed data section
rodata	Read-only data
common	Global variables

NET + OS section of the linker files

This table summarizes the NET+OS section of the linker files:

Section	Description
gnu_initdata	Stores jumper and button settings read at startup.
heap	Heap; grows upward.
gnu_ncc_initdata	Stores NET + OS settings determined at startup.
stack	System stack; grows downward.
netosstack	Stack for each processing mode; grows downward.
free_mem	Used for the kernel to create the timer thread and root thread. *Do not use this section for any other purpose*.

Section	Description
non_cache	Non-cacheable region of memory that is available through the nonCacheMalloc and NonCacheFree routines. non_cache must be a multiple of 1 MB.
ttb	Stores the mmuTTB table at the end of RAM. Initially is set up by the bootloader. Note that if you changettb_size, you must change SECOND_LEVEL_TABLE_SIZE in the mmu utility as well. *Do not overwrite this table.*

The ThreadX library is hard-coded to call `tx_application_define (void *first_unused_memory)` after the kernel has been loaded and just before the kernel scheduler starts.

The `free_mem` function creates the root thread responsible for starting NET+OS and the IP stack. Do not pass any other address to create the root thread. The *`first_unused_memory`* argument points to a global variable that is set up by the kernel.

Address mapping

The linker command files that are generated for each application set up an address map and or cache data. You enable or disable the instruction cache by changing `BSP_AUTOMATICALLY_ENABLE_INSTRUCTION_CACHE` in the `bsp.h` file.

The NS9750_a development board currently has:

- 16 MB SDRAM on `CS4`, mapped at `0X0000000`
- 8 MB flash on `CS1`, mapped at `0X50000000`

Hardware Dependencies

CHAPTER 9

This chapter discusses hardware dependencies that you need to be aware of when you port NET+OS to application hardware.

Overview

To port NET+OS to your application hardware, you need to be aware of specific dependencies in these areas:

- Direct Memory Access (DMA) channels
- Ethernet PHY
- Endianness
- Timers
- Interrupts
- Memory map

The rest of the sections in this chapter describe these hardware dependencies.

DMA channels

The NS9750 uses three DMA controllers. Two of them exist on Bbus, and one exists in the Bbus Bridge module. (For detailed information, see the *NS9750 Hardware Reference*.)

One of the Bbus DMA controllers supports all Bbus peripherals except the USB device, and the other is dedicated to the USB device interface. The AHB DMA has two channels, which are dedicated to internal memory-to-memory transfers and are not used by NET+OS. Your application can use the AHB DMA channels.

Ethernet PHY

NET+OS supports PHYs that use the MII interface. The PHY driver, which is implemented in the `mii.c` file, supports the LXT971A lPHY by Intel.

You modify the `mii.c` file to support additional PHYs.

To enable PHY interrupt to monitor the Ethernet link, set BSP_USE_PHY_INTERRUPT to TRUE in the bsp.h file. If you do not set BSP_USE_PHY_INTERRUPT to TRUE, the ThreadX timer is used to monitor the Ethernet link.

The NS9750 series of NET+ARM processors uses Interrupt ID 6 for the Ethernet PHY interrupt, implemented as a level interrupt. If PHY interrupt is enabled, make sure customizeIsMiiInterruptActiveLow returns the correct value.

Endianness

The BSP supports Big Endian mode only.

General purpose timers

This section describes how the general purpose timers are used.

System timers

NET+OS uses the first four of the 16 general purpose timers.

This table shows how timers 0–3 are used:

Timer	How used by NET+OS
0	As the system heartbeat clock. The kernel uses the system heartbeat clock for timing and pre-emption of tasks. The BSP_TICKS_PER_SECOND constant in the bsp.h file controls the frequency of the system heartbeat clock. This value, which determines the heartbeat rate, should be between 1 and 1000. A value of 100, for example, provides a heartbeat rate of one tick every ten milliseconds.
1	Used by NAuWait and NAWait, which the flash driver uses to: ■ Provide delays needed for programming flash ■ Provide the reads that are needed to verify that a flash was properly programmed.

Timer	How used by NET+OS
2	To support the *statistical profiler* that is included with NET+OS. You use the profiler to understand trends of execution. The profiler records the location of an application using two resources – the FIQ interrupt and Timer 2 – that normally are not used.
3	To support the USB device DMA timeout function. Used by the USB device driver to close out a DMA transfer when the received size matches a multiple of the endpoint packet size. For example, if the packet size is 64, this timer is needed to close out the DMA buffer when the data received is 64,128, or *n*x64.

If you do not plan to use a particular feature, you can shut it off, and use the timer in your application. This applies only to the timers that NET+OS uses.

All other general purpose timers

Any custom application can use the rest of the general purpose timers.

Interrupts

The interrupt priorities are specified in the `bsp.c` file in the `platforms` directory. You can modify the priority of the interrupts by using the `MCahbPriorityTab` table in `bsp.c`.

The Bbus peripherals – all four serial ports, the USB device, and the 1284 – combine all their interrupts into one Bbus Aggregate interrupt. As a result, the Bbus Aggregate interrupt priority is raised to a higher level for better performance.

For a description of interrupts in NET+OS, see Chapter 2, "Processor Modes and Exceptions."

For information about the interrupt controller, see the *NS9750 Hardware Reference*.

Memory map

NET+OS has this memory map on the NS9750 development board:

- RAM on `CS4` is mapped from address `0x0` to `0x00ffffff`, for the 16 MB SDRAM supplied on the NS9750.
- ROM on `CS1` is mapped from address `0x50000000` to `0x507fffff`, for the 8 MB flash ROM.

The BSP assumes that RAM is located at address `0x0`, and it dynamically writes the exception vector table to this location.

The BSP sets up chip selects to correspond to the size of the memory parts on your board. The chip selects can be modified in software, in the `cs.c` platforms file; all other functions are fixed.

This table shows a complete memory map:

Address range	Size	System function
0x0000 0000 - 0x0FFF FFFF	256 MB	System Memory Chip Select 4 dynamic memory (RAM)
0x1000 0000 - 0x1FFF FFFF	256 MB	System Memory Chip Select 5 dynamic memory (RAM)
0x2000 0000 - 0x2FFF FFFF	256 MB	System Memory Chip Select 6 dynamic memory (RAM)
0x3000 0000 - 0x3FFF FFFF	256 MB	System Memory Chip Select 7 dynamic memory (RAM)
0x4000 0000 - 0x4FFF FFFF	256 MB	System Memory Chip Select 0 static memory (ROM)
0x5000 0000 - 0x5FFF FFFF	256 MB	System Memory Chip Select 1 static memory (ROM)
0x6000 0000 - 0x6FFF FFFF	256 MB	System Memory Chip Select 2 static memory (ROM)
0x7000 0000 - 0x7FFF FFFF	256 MB	System Memory Chip Select 3 Static Memory (ROM)
0x8000 0000 - 0x8FFF FFFF	256 MB	PCI memory
0x9000 0000 - 0x9FFF FFFF	256 MB	Bbus memory

Address range	Size	System function
0xA000 0000 - 0xA00F FFFF	1 MB	PCI I/0
0xA010 0000 - 0xA01F FFFF	1 MB	PCI CONFIG_ADDR
0xA020 0000 - 0xA02F FFFF	1 MB	PCI CONFIG_DATA
0xA030 0000 - 0xA03F FFFF	1 MB	PCI arbiter
0xA040 0000 - 0xA04F FFFF	1 MB	Bbus-to-AHB bridge
0xA050 0000 - 0xA05F FFFF	1 MB	Reserved
0xA060 0000 - 0xA06F FFFF	1 MB	Ethernet communication module
0xA070 0000 - 0xA07F FFFF	1 MB	Memory controller
0xA080 0000 - 0xA08F FFFF	1 MB	LCD controller
0xA090 0000 - 0xA09F FFFF	1 MB	System control module
0xA0A0 0000 - 0xFFFF FFFF	1526 MB	Reserved

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W


Digi®