



NET+OS Programmer's Guide

Part number/version: 90000785_G

Release date: April 2010

www.digiembedded.com

©2004-2010 Digi International Inc.

Digi, the Digi logo, the Rabbit logo, the MaxStream logo, the When Reliability Matters logo, Digi Connect, Digi Connect SP, Digi Connectware Manager, ConnectPort, PortServer, Rabbit 2000, XBee, and NET+ are trademarks or registered trademarks of Digi International in the United States and other countries. ARM and NET+ARM are trademarks or registered trademarks of ARM Limited. All other trademarks are the property of their respective owners.

Information in this document is subject to change without notice and does not represent a commitment 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.

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

Support

- United States telephone: 1 877 912-3444
- International telephone: 1 952 912-3444
- Fax: 952 912 4952
- Web site: <http://www.digiembedded.com>

Contents

Chapter 1: NET+OS Introduction	1
System components	2
ThreadX RTOS kernel	3
Advanced Web Server (AWS)	3
Internet Address Manager (IAM)	3
System requirements	4
Working with NET+OS/Digi ESP	4
NET+OS tree structure	4
bin	5
debugger_files	6
Documentation	6
gnusrc	7
h	7
lib	7
mibcomp	7
src	7
utilities	8
Part 1: Customization	9
Chapter 2: BSP Overview	10
Overview	11
What is the board support package (BSP)?	11
BSP tree structure	12

Top-level directory	12
bootloader subdirectory	13
devices directory	14
platforms directory	15
Platforms.....	16
Initialization	17
Initializing hardware	17
Initialization sequence for ARM7 platforms	17
Initialization sequence for ARM9 platforms	18
Chapter 3: Creating a New Platform	20
Overview	21
Creating a new platform	21
Step 1: Create a new platform directory	21
Step 2: Copy a similar platform into the new directory	21
Step 3: Update Makefile.bsp	22
Step 4: Update Makefile.example	22
Step 5: Update Makefile.files	22
Step 6: Update Makefiles in the linkerscripts directory	23
Chapter 4: Configuring a New Platform	24
Overview	25
Customizing the BSP for application hardware	25
Task 1: Set the GPIO configuration	25
Task 2: Modify the BSP to set up the required drivers	26
Task 3: Modify the BSP configuration files	28
Task 4: Modify the format of BSP arguments in NVRAM	29
Task 5: Modify error and exception handlers	31
Task 6: Modify the startup dialog	32
Task 7: Modify the POST	33
Other BSP customizing	33
BSP_NVRAM_DRIVER	33
TCP/IP stack	33

File system.....	35
Chapter 5: Customizing the Bootloader	38
Overview	39
Bootloader application images	40
ROM image	40
RAM image	40
Application image structure	42
Application image header	43
Generating an image	45
Configuration file	46
General bootloader limitations	47
Customizing the bootloader utility	48
Customization hooks	48
Part 2: Hardware	58
Chapter 6: Bringing Up New Hardware	59
Verify the debugger initialization files.....	60
Using the MAJIC/MAJICO probe.....	61
Debug the initialization code	62
Preparing to debug the initialization code	62
Debugging the initialization code on ARM7 platforms	64
Debug the INIT.s file	64
Debug the ncc_init routine.....	65
Debug the NABoardInit routine	66
Debug the Ethernet driver startup.....	66
Debugging the initialization code on ARM9 platforms	67
Debug the init.arm file	67
Debug the ncclnit routine	67
Debug the NABoardInit routine	68
Debug the Ethernet driver startup	68
Chapter 7: Memory Map	69

Memory aliasing (ARM7)	70
Memory map (ARM9)	72
Chapter 8: Adding Flash	74
Overview	75
Supported flash memory parts.....	75
Flash table data structure.....	76
Supporting larger flash.....	80
Chapter 9: Hardware Dependencies for ARM7-based Modules	
81	
Overview	82
DMA channels.....	82
Serial ports	83
Software watchdog.....	83
Endianness	83
System timers	83
Interrupts	84
RS-232-style communications.....	85
Chapter 10: Hardware Dependencies for ARM9-based	
Modules	86
Overview	87
DMA channels on the NS9750 and NS9360 Processors	87
DMA Channels on the NS9210 and NS9215 Processors.....	87
Endianness	88
General purpose timers.....	88
System timers	88
All other general purpose timers.....	88
Interrupts	89
System clock.....	89
Part 3: Makefiles	90

Chapter 11: NET+OS Makefile System	91
Overview	92
Makefile hierarchy	93
Building all libraries	94
Building individual libraries	94
Library directory structure	95
Library Makefile variables	95
Adding new libraries to the system	96
Cleaning libraries	96
Bootloader Makefile	96
Example: using the Makefile	98
 Chapter 12: Application Makefile	 99
Building applications	100
Application Makefiles	100
Definitions of the Makefile	101
Makefile hierarchy	103
Makefile targets	104
Building an application	104
Creating .gdbinit files for your debugger	104
Cleaning an application	105
Porting an application to a new platform	105
 Part 4: Building Web Pages	 106
 Chapter 13: Using the Advanced Web Server Utility	 107
Overview	108
The PBuilder utility	108
Comment tags	108
About the Advanced Web Server Toolkit documentation	109
Running the PBuilder utility	109
Linking the application with the PBuilder output files	112
security.c file	112

cgi.c and file.c files	112
Creating Web pages	113
AWS custom variables	113
Data types	114
Displaying variables	115
Changing variables	115
Security	118
Exceptional cases	118
Controlling the MAW module	119
Setting the semaphore timeout	119
Array subscripts	120
Error handling	121
Phrase dictionaries and compression	121
Maintaining and modifying Web content	122
Sample applications	123

Part 5: Miscellaneous124

Chapter 14: Porting NET+OS v6.x Applications to NET+OS

v7.x	125
Changes to the flash driver	126
IAM and ACE	126
Changes to the sockets API	126
Changes to SNMP	128
netos/src/bsp/customize directory	128
Changes to Makefile variables and defines	128
Automatic RAM sizing	129
Porting pre-NETOS 7.x PPP applications	130
Adding a route	130
Deleting a route	130
Adding PAP user/password or adding CHAP ID and secret key pair	130
Checking link status	131
Creating the interface	131
Getting the peer assigned local address	132

Closing the interface	133
Setting authentication and compression	133
Initializing serial port configuration	134
Setting the ring count	134
Dial string settings	135
Chapter 15: Processor Modes and Exceptions	136
Overview	137
Vector table	137
IRQ handler	139
Servicing AHB interrupts in ARM9 based NET+ARM processor	140
Servicing Bbus interrupts in ARM9 based NET+ARM processor	141
Changing interrupt priority	141
Interrupt service routines	150
Installing an ISR	150
Disabling and removing an ISR	150
ARM9 FIQ handlers	150
ARM7 FIQ handlers	151
Chapter 16: Device Drivers	152
Overview	153
Adding devices	153
deviceInfo structure	153
Device driver functions	154
Return values	165
Modifications to Cygwin's standard C library and startup file	166
Modifying the libc.a library and crt0.o startup file	167
NET+OS device drivers	167
Device driver interface	169
Part 6: Troubleshooting	170
Chapter 17: Troubleshooting	171
Diagnosing errors	172

Diagnosing a fatal error	172
Diagnosing an unexpected exception	172
Reserializing a module	173
Observing the LEDs	173
Assigning a MAC address to the module	174
Restoring the contents of flash memory	176
Step 1: Configure the module and the debugger	177
Step 2: Building the bootloader	177
Step 3: Building the application image and starting naftpapp	177
Step 4: Sending rom.bin to the module	178
Step 5: Verifying the boot ROM image on the module	179
Step 6: Verify the contents of flash	179

Using This Guide

Review this section for basic information about this guide, as well as for general support contact information.

About this guide

This guide describes NET+OS operating system, operating environment, or development environment and how to use it as part of your development cycle. The NET+OS operating system, operating environment, or development environment is a network software suite optimized for the NET+ARM family of chips, processor, or modules.

The chapters in this guide are functionally grouped into six parts:

- Part 1: Tools
- Part 2: Customization
- Part 3: Hardware
- Part 4: Makefiles
- Part 5: Miscellaneous
- Part 6: Troubleshooting

Installation directory

The instructions in this document show the installation directory as `C:\netos`. If you install your software in the default directory, be aware that you will see `netos` followed by its version numbers; for example: `C:\netosxx`

Conventions used in this guide

This table describes the typographic conventions used in this guide:

This convention	Is used for
<i>italic type</i>	Emphasis, new terms, variables, and document titles.
bold, sans serif type	Menu commands, dialog box components, and other items that appear on-screen.
Select menu name → menu selection name	Menu commands. The first word is the menu name; the words that follow are menu selections.
monospaced type	File names, pathnames, and code examples.

Related documentation

For additional documentation, see the Documentation folder in the NET+OS Start menu.

Documentation updates

Digi occasionally provides documentation updates through the package manager. Be aware that if you see differences between the documentation you received in your NET+OS package and the documentation on the Web site, the Web site content is the latest version.



NET+OS Introduction



C H A P T E R 1

This chapter introduces the NET+OS development environment and its components.

Overview

The NET+OS products offer an embedded solution for hardware and networking software that are being implemented into product designs.

The NET+OS package includes:

- Either a Digi Connect module, a ConnectCore module, or a development board
- A board support package
- A JTAG debugger
- Networking firmware
- Object code with application program interfaces (APIs)
- Development tools
- Sample code
- Documentation

For information about the Digi Connect or ConnectCore module or the development board, see your hardware reference.

System components

This section describes the components that make up the NET+OS software.

NET+OS runtime software

NET+OS software provides the building blocks to help you create your custom applications. You create your application with calls to APIs for:

- The board support package (BSP)
- ThreadX RTOS kernel
- Basic Internet protocols
- Higher-level protocols and services

Board support package

The NET+OS BSP is a collection of ARM object code, C source-code drivers, and the bootloader. The BSP initializes hardware and software and provides power-on self tests (POST).

The BSP includes a set of APIs that you use to incorporate device peripheral functionality into your application. In addition, the BSP provides the drivers for your Digi Connect module, including those for Ethernet, serial, SPI, flash, USB host, USB device, LCD, PCI/CardBus, and others.

ThreadX RTOS kernel

The ThreadX[®]RTOS, from Express Logic, is based on a high-speed picokernel architecture. ThreadX helps you manage complex event synchronization and memory using threads, queues, application timers, semaphores, and event flags.

Advanced Web Server (AWS)

Using the AWS, you convert your HTML into C code so you can compile that code with the rest of your application. AWS provides support for HTML, multiple Web object sources, object compression, and advanced security.

Internet Address Manager (IAM)

The NET+OS development environment provides services such as the IAM, which lets you acquire IP parameters at startup from multiple prioritized sources, including DHCP, Auto IP, and others.

System requirements

To run the NET+OS development software, your system must meet these requirements:

- Intel architecture (x86) PC running one of these Microsoft operating systems:
 - Windows Vista
 - Windows XP Professional
 - Windows 2000 Professional

Be aware that Windows 9x is no longer supported because of limited system resources in the operating system.

- CPU: Pentium 4/1.8 GHz; 2.4 GHz or faster recommended
- Minimum system RAM: 512 MB; 1GB recommended
- Free disk space: 1 .1GB

Working with NET+OS/Digi ESP

Digi ESP for NET+OS is an Integrated Development Environment (IDE) you can use to develop embedded applications with NET+OS.

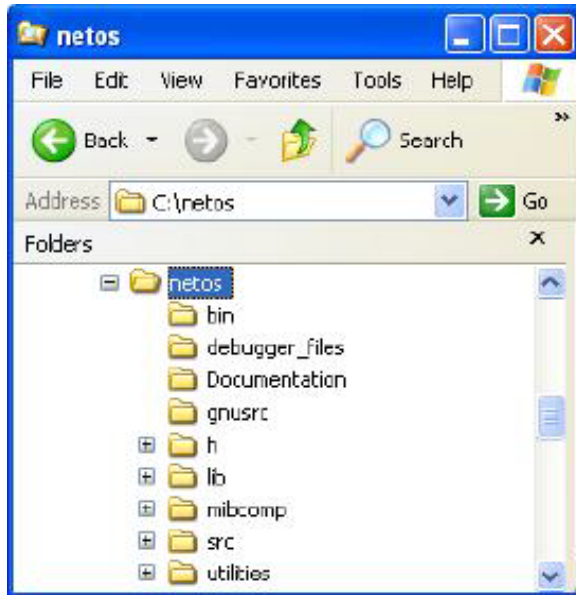
Digi ESP for NET+OS is built on Eclipse and the C/C++ Development Tools (CDT) plugin. Eclipse is an open, extensible IDE, and the CDT is a plugin that provides support for developing applications with C and C++ in the Eclipse platform.

To start Digi ESP, select **Start → NET+OS → Digi ESP**.

The software is located in Program Files\Digi\Digi ESP.

NET+OS tree structure

The NET+OS tree structure is divided into subdirectories, with `netos` as the root directory, as shown next:



The next sections describe the subdirectories under `netos`.

bin

The binary files that are executable on a PC and used by the NET+OS development environment are located in `netos/bin`. This list shows some of the most commonly-used files:

- `spibootldr.exe` — Uses the `netos/src/bsp/platforms/"my_platform"/spibootldr.dat` configuration file for SPI devices.
- `smidump.exe` — MIB compiler for SNMP MIBs written in either the SMI v1 or SMI v2 formats.
- `compress.exe` — Compresses the application image's `.bin` file to save memory in flash.
- `bootldr.exe` — Inserts a header at the beginning of the image based on information read from the `netos/src/bsp/platforms/my_platform/bootldr.dat` configuration file.

`bootldr.exe` calculates a CRC32 checksum for the entire image, including the header, and places it at the end of the updated file.

These are the fields in the `boothdr.dat` file:

Field	Description
WriteToFlash	Used by the bootloader when it downloads a file from a network server to determine whether to write the file to flash. Set to either yes or no.
Compressed	Indicates whether the file should be compressed Set to either yes or no.
ExecuteFromRom	Specifies where the bootloader executes the application: <ul style="list-style-type: none"> ■ To execute directly from flash, set to yes. ■ To decompress the file to RAM, set to no.
flashOffset	Indicates where in flash to write the file to. Set to a hexadecimal value.
ramAddress	Indicates where in RAM to copy the application to decompress it. Set to a hexadecimal value.
MaxFileSize	Indicates the maximum size of the file in bytes. Set to a hexadecimal value.

debugger_ files.

This directory contains sample `gdb` initialization scripts and configuration setting files for the JTAG debugger. The file also contains the `gdbThreadX` script, which sets up macros to view ThreadX structures. This file is located in `netos/debugger_files`.

Documentation

All the NET+OS hardware- and software-related documentation is located in `netos/Documentation`. This directory contains the *NET+OS API Reference* and hardware and software guides

gnusrc

These files allow you to interface the GNU C library I/O functions to the file systems and C library time functions to the real time clock driver. The GNU I/O and time driver interface functions are located in `netos/gnusrc`.

h

Contains the public API header files. When an application calls an API function from a NET+OS library, the respective C file must include the header file for the API routines.

lib

Contains the ARM7 and ARM9 libraries used to build your images.

mibcomp

Contains files used for the SNMP MIB compiler. Store all MIBs referenced in your enterprise MIB in `\netos\mibcomp\mibcomp_win321\smi\mibs\ietf`. This folder is defined as your `SMIPATH` and is used by the MIB compiler.

src

These sections describe some of the subdirectories of `src`.

flash

Contains the files used for the NET+OS NOR flash driver.

sflash

Contains files used to support a serial Flash driver.

fs_intf

Contains the file system interface files used for FTP and email (POP3, SMTP).

posix

Contains sample files used to implement a POSIX-like API.

rphttpd

Contains the Advanced Web Server (AWS) stub files required to use the Advanced Web Server AWS, which include security stubs, User Dictionaries, and CGI stubs.

treck

Contains the NET+OS TCP/IP public header files.

utilities

Contains sample code used for device discovery, which uses the Digi ADDP (Advanced Device Discovery Protocol). Also includes a header (`Include\addp.h`) for the ADDP library interface and an example using the ADDP library for a WIN32 application.

Part 1: Customization



BSP Overview



C H A P T E R 2

This chapter describes the NET+OS board support package (BSP).

Overview

Application development involves writing hardware-independent, high-level software components. Using a NET+ARM module and its associated board support package (BSP,) you can begin software development immediately. The NET+OS development environment is delivered with BSPs to support all NET+ARM processors and all DIGI Connect and ConnectCore modules. Each BSP is tailored to support the module's specific target processor (for example, the NS9360 or NS7520) and the components that surround the processor (memory and PHY).

Some modules can have more than one hardware configuration. For example, it may be possible to configure processor pins to be either a serial port or general purpose I/O pins. You determine how the hardware should be configured to support your application and then configure the BSP to set the correct configuration at powerup and load the proper device drivers to support it.

This chapter describes the BSP and how it supports multiple platforms. It also describes the tree structure of both the BSP and NET+OS.

What is the board support package (BSP)?

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 software restart
- Handling processor exceptions
- Device drivers
- Starting the ThreadX kernel
- Starting the network stack

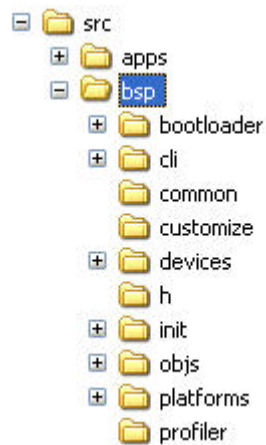
The BSP provides the hardware services in a standardized application programming layer (API) to the application software, allowing the application software to maintain hardware platform independence.

BSP tree structure

These sections describe the BSP tree structure.

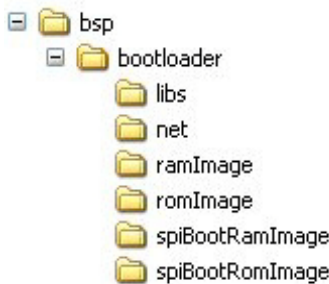
Top-level directory

The NET+OS BSP is located in the `src/bsp` directory. The top level directory contains the `Makefile` for the BSP and the `Makefile` for the bootloader. This figure shows the top level directory:



bootloader subdirectory

The `bootloader` subdirectory, shown next, contains the source code for the SPI and ROM-based bootloaders:



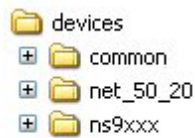
The bootloader has two parts: the ROM image and the RAM image. Because the bootloader size is kept to less than 64K, the `libs` directory contains the libraries that are linked into the bootloader. The bootloader does not link in the standard NET+OS libraries.

This table describes the subdirectories of the `bootloader` directory:

This directory	Contains
libs	Libraries that are specific to the bootloader
net	The network-related code for the BSP
ramImage	The code and Makefile for the portion of the bootloader that runs from RAM
romImage	The Makefile and code for the portion of the bootloader that runs from ROM
spiBootRamImage and spiBootRomImage	The SPI bootloader

devices directory

The devices directory, which contains all the NET+OS device drivers, is shown next:

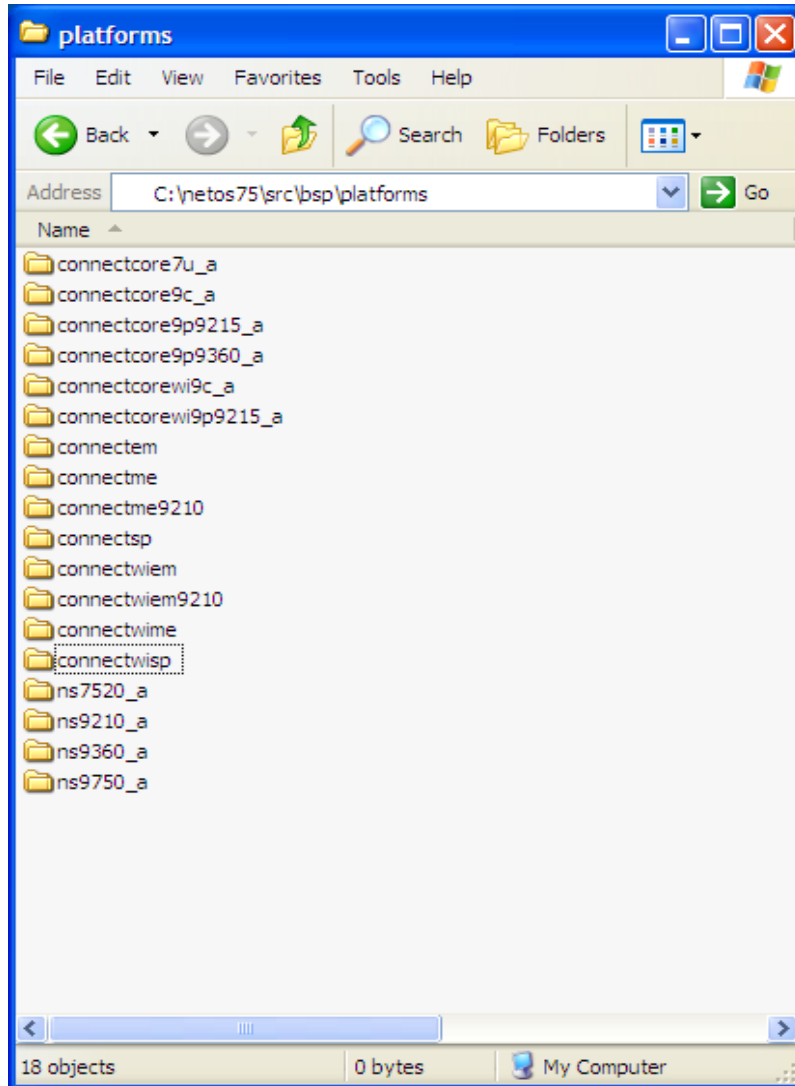


The device drivers are separated into three directories:

This directory	Contains
common	The device drivers that are common to all processors, such as serial and Ethernet
net_50_20	The drivers for the NS7520 and the NET+50
ns9xxx	The drivers for the NS9210, NS9215, NS9360 and NS9750

platforms directory

The `platforms` directory, which contains all the supported platforms, is where you add your platform. This figure shows all of the supported platforms:



When you create a new platform, you copy an existing platform and create a new subdirectory in this tree.

Platforms

If you are adding a new platform to your NET+OS development environment, start with a platform that is similar to yours. This table shows the list of supported platforms provided with the NET+OS development environment:

Platform name	CPU
ns7520_a	NS7520
connectme	NS7520
connectem	NS7520
connectwime	NS7520
connectwiem	NS7520
connectwiem9210	NS9210
connectme9210	NS9210
connectsp	NS7520
connectwisp	NS7520
ns9750_a	NS9750
ns9360_a	NS9360
ns9210_a	NS9210
connectcore9c_a	NS9360
connectcorewi9c_a	NS9360
connectcore9p9215_a	NS9215
connectcorewi9p9215_a	NS9210
connectcore9p9360_a	NS9360
connectcore7u_a	NS7250

For a description of your platform, see the hardware reference for your module or processor.

Initialization

This section describes the power-up and initialization of the NET+OS development environment for NOR-Flash based platforms. In general, you do not need to modify the initialization code.

For NAND-Flash based platforms, see the *Digi NET+OS U-Boot reference Manual*.

Initializing hardware

These are the locations of the hardware initialization code:

- **ARM7-based platforms.** `src/bsp/init/arm7`
- **ARM9-based platforms.** `src/bsp/init/arm9`

The `main()` routine is located in `src/bsp/common/main.c`.

Initialization sequence for ARM7 platforms

`Reset_Handler`, located in the `INIT.s` file, is the first routine that is executed when the processor is powered on. `Reset_Handler` must perform these steps:

- 1 Initialize supervisor mode and disable interrupts.
- 2 (*NET+50 only*) Initialize the PLL.
- 3 Execute a software reset to get the hardware into a known state.
- 4 Put the DMA controller into test mode so the DMA context RAM can be used as a temporary stack.
- 5 Jump to the `ncc_init` routine (located in `NCC_INIT.c`).
- 6 Set up the system control register.
- 7 Initialize the GPIO pins.
- 8 Set up the chip selects.
- 9 Run the memory test.
- 10 Verify that the application will fit into RAM and return.
- 11 Set up the stacks for the different processor modes.

- 12 Jump to the C library startup routine, which sets up the C runtime environment.
The C Library startup routine calls `main()` when it completes.
- 13 Initialize the C++ runtime environment.
- 14 Execute the Power On Self Test (POST) if the POST is enabled.
- 15 Initialize the processor vector table.
- 16 Execute `NABoardInit` to initialize the flash and NRAM drivers.
- 17 Call the low level device driver initialization routines, which perform any hardware specific set up required before the operating system starts.
- 18 Start the ThreadX operating system.

Initialization sequence for ARM9 platforms

`Reset_Handler`, located in the `init.arm` file, is the first routine that is executed when the processor is powered on. `Reset_Handler` must perform these steps:

- 1 Determine whether or not the application is booting from SPI:
 - **Booting from SPI.** The initialization code sets a flag that is read later. The code that initializes the memory controller is skipped because this is already done during the SPI boot.
 - **Not booting from SPI.** The initialization code initializes the memory controller so the application can run from SDRAM.
- 2 On the NS9360 and NS9750 platforms, take the BBUS out of reset.
- 3 Use the Ethernet receive FIFO as a temporary stack on the NS9210 and NS9215 platforms for the call to `ncclInit`. On the NS9360 and NS9750 platforms, use a section of RAM as a temporary stack.
- 4 Jump to the `ncclInit` routine in the `NCC_INIT.c` file, which contains the rest of the hardware initialization routines in the `NCC_INIT` routine.
- 5 Read and save registers that tell whether the application is in the debugger or this is a software restart.
If either of these is true, the application can skip over some sections of the hardware initialization.
- 6 Set up the `SimpleSerialDriver`.
This step allows you to use the `mprintf` routine, which you can use to print debug information during bootup.

- 7 Set up the GPIO pins.
- 8 Enable the instruction cache.
- 9 Set up the chip selects.
- 10 Run the memory test.
- 11 Verify that the application will fit into RAM and return.
- 12 Set up the stacks for the different processor modes.
- 13 Jump to the C library startup routine, which initializes the C runtime environment.
The C Library startup code calls `main`, in `src/bsp/common/main.c`, when it completes.
- 14 Initialize the C++ runtime environment.
- 15 If the Power On Self Test (POST) is enabled, execute it.
- 16 Set up the vector table.
- 17 Enable the Memory Management Unit (MMU).
- 18 Call `NABoardInit`, which initializes the flash and NVRAM drivers.
- 19 Perform the first level device driver initialization.
- 20 Start ThreadX.

Creating a New Platform

C H A P T E R 3

This chapter describes how to add support for a new platform for your application.

Overview

If you use the same module in two or more products, you may need to configure it differently in each product. Most of the configuration information for the module and BSP is stored in a set of files in a platform directory. The NET+OS development environment ships with template platforms for each of the modules. This chapter describes how to create copies of the template platforms, which you will modify to configure the BSP to your application's requirements.

Creating a new platform

Each subdirectory in the `src/bsp/platforms` directory contains the files that are specific to a specific module. You need to create your own platform subdirectory so you can modify the configuration files in it for your application's requirements. In addition, you need to update several NET+OS `Makefiles` to support the new platform.

Step 1: Create a new platform directory

Create a new directory in `src/bsp/platforms` for your new platform. The `Makefiles` use the directory name as the platform name, so use a name that is easy to type on the command line.

Step 2: Copy a similar platform into the new directory

Determine which NET+OS platform template supports your modules. The name of the platform directory is based on the module name. (For example, for the `connectcore9c`, the template directory is `src/bsp/platforms/connectcore9c_a.`) Then copy the files from that directory into the platform subdirectory you just created.

Step 3: Update Makefile.bsp

The `Makefile.bsp` file in the `platform` directory sets up variables that are used by other Makefiles. Edit this file and update the variables as needed by your platform:

- Set `PROCESSOR` to the name of the processor your platform uses, such as `NS9215`.
- Set `CPU` to the type of CPU the processor uses, either `arm9` or `arm7`.
- Set `SPI_BOOTLOADER` to `ENABLE` if the platform boots from a SPI part, or to `DISABLE` if it boots from NOR flash.
- Set `BUILD_BOOTLOADER` to `TRUE` to build the bootloader when the BSP library for the platform is built.
- Set `WIRELESS_PLATFORM` to `TRUE` if the platform supports an 802.11 interface.
- Set `GRAPHICS_PLATFORM` to `TRUE` if the platform supports a display device and support for the `NET+OS` graphics and `WxWidgets` libraries should be built into the BSP.

Step 4: Update Makefile.example

The `Makefile.examples` file in the `platform` directory sets up a list of sample applications the platform supports. Update the list as needed for your platform.

Step 5: Update Makefile.files

The `Makefile.files` file is used by other Makefiles to set up source and include paths. Edit this file and update `PLATFORM_INC`, `PLATFORM_SRC` and `PLATFORM_BSP_OBJ` as needed by your platform:

- Set `PLATFORM_INC` to list the include paths as needed by your platform that are not part of the standard BSP include paths.
- Set `PLATFORM_SRC` to list the source paths needed by your platform that are not part of the standard BSP source paths.

Configuring a New Platform

C H A P T E R 4

This chapter describes how to configure a NET+OS BSP platform.

Overview

This chapter describes how to configure a platform to the requirements of your application.

All the files discussed in this chapter are in the `platform` directory you created using instructions from Chapter 5, “Creating a New Platform.”

Customizing the BSP for application hardware

Task 1: Set the GPIO configuration

You can configure many of the processor pins to support one of several functions. For example, GPIO pin 0 on the NS9360 can be configured to be one of these functions:

- The TxData pin for serial port B
- The DONE signal for DMA channel
- An input to timer 1
- A general purpose I/O pin

You must determine how these pins should be configured to support your hardware.

The `gpio.h` file has a set of macro definitions that determine how each pin is configured by the BSP initialization code. (The macro definitions are described in the “BSP/Device Drivers/Signal Multiplexing and GPIO” section of the *API Reference*.) To set the GPIO configuration to support your application hardware, edit the `gpio.h` file in your platform subdirectory.

The BSP code generates compiler errors if you select an invalid GPIO configuration; for example, configuring one pin to perform two functions.

To test your GPIO configuration, execute the BSP Makefile:

1 Open a command shell by clicking the XTools icon.

2 Enter this command:

```
cd src/bsp directory
```

3 Enter this command:

```
"make PLATFORM=myPlatform"
```

where

myPlatform is the name of your platform.

Task 2: Modify the BSP to set up the required drivers

You must configure the `bsp_drivers.h` file (in your `platforms` directory) to enable the drivers you want to run with your application. To enable the drivers, set the values of these macros:

- `BSP_SERIAL_PORT_1`
- `BSP_SERIAL_PORT_2`
- `BSP_SERIAL_PORT_3`
- `BSP_SERIAL_PORT_4`
- `BSP_INCLUDE_PARALLEL_DRIVER`
- `BSP_INCLUDE_I2C_DRIVER`
- `BSP_INCLUDE_LCD_DRIVER, BSP_INCLUDE_USB_DRIVER`
- `BSP_NVRAM_DRIVER`
- `BSP_INCLUDE_RTC_DRIVER`
- `BSP_INCLUDE_LCD_DRIVER`
- `BSP_INCLUDE_USB_DRIVER`
- `BSP_INCLUDE_PCI_DRIVER`
- `BSP_SPI_PORT`
- `BSP_PWM_MAXIMUM_CHANNELS`
- `BSP_QUAD_DECODER_ENABLE`

For more information about these macros, see the BSP configuration section of the *API Reference*.

The default configuration works with a development board. Note that drivers that use the same GPIO pins cannot function properly at the same time. Be sure to review the `bsp_drivers.h` and `bsp_serial.h` files carefully.

Make sure the GPIO pins needed for a device are configured to support it. For example, the GPIO pins used by the serial ports can be configured for other functions on the NS9360 processor. If you want to use the serial ports, make sure the GPIO pins are configured to act as serial ports.

Serial ports

The BSP supports either two serial ports on ARM7-based processors or four serial ports on ARM9 processors.

To set a serial port to a mode other than those already set up by the standard NET+OS release, modify the `gpio.h` file to set the GPIO pins to the appropriate value.

To disable the RS-232 serial peripheral interface controller, set `BSP_SERIAL_PORT_X`, where *x* is the number of the serial port, to `BSP_SERIAL_NO_DRIVER`.

I2C controller (ARM9 processors only)

The BSP is configured by default to enable support of the I2C peripheral device. To disable the I2C controller, set `BSP_INCLUDE_ITC_DRIVER` to `FALSE`.

LCD controller (ARM9 processors only)

The BSP is configured by default to enable support of the I2C peripheral device. To disable the LCD controller, set `BSP_INCLUDE_LCD_DRIVER` to `FALSE`.

PCI driver (NS9750 only)

The BSP is configured by default to enable support of the PCI peripheral device. To disable the PCI device driver, set `BSP_INCLUDE_PCI_DRIVER` to `FALSE`.

RTC (NS9210, NS9215, and NS9360)

The BSP supports a real time clock on NS9210, NS9215 and NS9360-based platforms.

- To enable the RTC device driver, set `BSP_INCLUDE_RTC_DRIVER` to `TRUE`.
- To disable the RTC device driver, set `BSP_INCLUDE_RTC_DRIVER` to `FALSE`.

Task 3: Modify the BSP configuration files

The BSP configuration settings are stored in files in the `platforms` directory. (For information about the content of the configuration files, see the *API Reference* and comments in the files.) You need to modify the configuration settings to support your application hardware.

Interrupt tables (ARM9 processors)

When you change the system interrupt priority, you must update these interrupt tables in the `bsp.c` file in the `platforms` directory:

- `NABbusPriorityTab` — This array contains the priority of each interrupt in the Bbus. The `NABbusPriorityTab` allows flexible prioritization for all Bbus interrupts in the NET+ARM platform that drive the `BBUS_AGGREGATE_INTERRUPT` in the `NAAhbPriorityTab` table.

The `NABbusPriorityTab` table is configured with interrupts of higher priority at the beginning and interrupts of lower priority at the end of the array.

- `NAAhbPriorityTab` — This array contains the priority of each interrupt in the AHB Bus. The `NAAhbPriorityTab` allows flexible prioritization for all the AHB interrupts in the NET+ARM platform that drive the ARM processor IRQ.

The table is configured with interrupts of higher priority at the beginning and interrupts of lower priority toward the end of the table. For more information about interrupts, see the “AHB interrupts” and “Bbus interrupts” sections in the hardware reference.

pci.c file (NS9750 modules only)

The `pci.c` file contains `customizePCISetup`, 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. 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.

customizeLed.c file

The `customizeLed.c` file contains the `NALedTable` structure global data 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 information about programming GPIO inputs in the hardware reference for the processor you are using.

customizeReset.c file

This file contains the `customizeRestart` and `customizeReset` functions, which determine what the system should do in case of a reset or restart request. This is where you place application-specific code that will be executed just before resetting the device.

Simple serial driver

A simple serial driver is provided for debugging the BSP before the main serial driver is loaded. The driver assumes that serial port 1 will be used at 9600 baud. To use a different port or baud rate, modify this driver.

The driver for the NS7520 and the NET+50 is located in the `simpleSerial.c` file in the `bsp/devices/net_50_20/serial` directory.

The driver for the NS9750/NS9360 is in the `bsp/devices/ns9xxx/common/serial` directory. The driver for the NS9210/NS9215 is in the `bsp/devices/ns9xxx/ns9215/serial` directory.

Task 4: Modify the format of BSP arguments in NVRAM

The BSP stores some configuration arguments in NVRAM. Customization hooks in `boardParams.c` read and write the configuration values.

NET+OS provides a simple NVRAM driver that can support several types of storage devices. Edit `bsp_drivers.h` and set `BSP_NVRAM_DRIVER` to indicate which storage device should be supported. For more information about `BSP_NVRAM_DRIVER`, see the online help.

The format of data in NVRAM is determined by the `devBoardParamsType` in `boardParams.h` structure. This structure, and the APIs that read and write it, support development boards; they do not support application hardware. Modify or rewrite this structure and its supporting APIs as needed by your application.

The NVRAM API has these functions:

Customization hook	Hardware feature/default values set
<code>customizeGetMACAddress</code>	<p>Determines the Ethernet MAC address used to communicate on the network.</p> <p>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.</p>
<code>customizeGetSerialNumber</code>	<p>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.</p>
<code>customizeSaveSerialNumber</code>	<p>Sets the serial number for the unit.</p> <p>The serial number is used only in some sample applications and in the startup dialog. Neither the API libraries nor any other part of the BSP uses the serial number. 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 stores a 9-character serial number in NVRAM.</p>
<code>customizeSetMACAddress</code>	<p>Sets the Ethernet MAC address for the unit.</p> <p>The default implementation stores the MAC address as a 6-byte array in NVRAM.</p>

Customization hook	Hardware feature/default values set
<code>customizeUseDefaultParameters</code>	Determines default configuration values and returns them in a buffer. The default implementation determines the default values through constants set in <code>appconf.h</code> . Modify this routine to support your application.
<code>CustomizeReadDevBoardParams</code>	Reads the configuration from NVRAM into a buffer. Modify this routine to support your application.
<code>customizeWriteDevBoardParams</code>	Writes the configuration to NVRAM. The default implementation accepts the current configuration as a buffer and writes the buffer into NVRAM.
<code>customizeGetIPParams</code>	Reads the IP-related configuration values from NVRAM.
<code>customizeSaveIPParams</code>	Writes the IP-related configuration values to NVRAM.

Task 5: Modify error and exception handlers

The `errhdlr.c` file in the `platforms` 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_sys.h`, you can configure the default error handler to do one of these:

- 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_sys.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 continue execution. You may need to modify the exception handler to better support your application.

Task 6: 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.

To turn off the dialog, update the file `bsp_sys.h` by defining the manifest constant `BSP_ENABLE_DIALOG` to `FALSE`.

If you do decide to prompt end users with a dialog, you usually have to rewrite the code in `dialog.c` to properly support your application. The `BSP_DIALOG_PORT` constant in your platform's `bsp_sys.h` file sets the I/O port for the dialog.

Task 7: 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 supplied with the NET+OS development environment tests the processor. You may want to create other POST routines that test additional hardware on your board.

Other BSP customizing

This section describes additional customizing you may want to do.

BSP_NVRAM_DRIVER

The `BSP_NVRAM_DRIVER` constant in `bsp_sys.h` defines the non-volatile memory type used to store the configuration information. This list describes the settings:

- `BSP_NVRAM_NONE` — No NVRAM driver is to be built
- `BSP_NVRAM_LAST_FLASH_SECTOR` — The last sector of flash memory to be used for NVRAM
- `BSP_NVRAM_SEEPROM` — The serial EEPROM driver is to be built
- `BSP_NVRAM_SEEPROM_WITH_SEMAPHORES` — The serial EEPROM driver with semaphore protection is built
- `BSP_NVRAM_LAST_SFLASH_SECTOR` — The last sector of serial flash is to be used for NVRAM

TCP/IP stack

The TCP/IP stack is the software module that handles networking functionality and is started as part of the BSP initialization process. You configure the TCP/IP stack using these functions and constants:

Function or constant	Description
BSP_WAIT_FOR_IP_CONFIG	<p>This constant in <code>bsp_net.h</code> determines whether the BSP waits for the stack to be configured before starting the application by calling <code>applicationStart()</code>. Previous versions of NET+OS waited for the stack to be configured.</p> <p>Your application should not use any network resources until the stack configures itself by setting an IP address on at least one interface. To determine whether an IP address has been assigned to an interface, use <code>customizeIamGetIfAddrInfo()</code>.</p> <ul style="list-style-type: none">■ To cause the BSP to wait for an IP address to be configured on at least one interface before calling <code>applicationStart</code>, set <code>BSP_WAIT_FOR_IP_CONFIG</code> to <code>TRUE</code>.■ To call <code>applicationStart</code> without waiting for an IP address to be assigned, set <code>BSP_WAIT_FOR_IP_CONFIG</code> to <code>FALSE</code>.
BSP_ENABLE_ADDR_CONFLICT_DETECTION	<p>This constant in <code>bsp_net.h</code> enables IP address conflict detection during initial IP address configuration. If you defined <code>BSP_ENABLE_ADDR_CONFLICT_DETECTION</code> to <code>TRUE</code>, the IAM subsystem sends ARP probes to detect IP address conflict for static IP address protocols.</p> <p>See the Network Interface Configuration in the NET+OS API Reference for more information on Address Conflict Detection.</p>
NAIpSetKaInterval	<p>This function in <code>naip_global.c</code> overrides the default value for the TCP keepalive interval, which is 2 hours (7200 seconds). If <code>ka_interval == 0</code>, keepalive is turned off.</p>
NAIpSetDefaultIpTtl	<p>This function in <code>naip_global.c</code> sets the default value for the time-to-live field of outgoing packets. This value is used unless it is overridden on a specific socket by the <code>IP_TTL</code> socket option.</p>

Function or constant	Description
NAIpSetTcpMsl	This function in <code>naip_global.c</code> overrides the default value for the TCP MSL and TCP <code>TIME_WAIT</code> interval. The default value of TCP MSL is 120 seconds. The <code>TIME_WAIT</code> interval is set to <code>(tcp_msl * 2)</code> .
APP_NET_HEAP_SIZE	This constant in <code>appconf.h</code> sets the TCP/IP stack heap size for dynamic allocations. The TCP/IP stack allocates all packet buffers from this piece of memory.

Internet Address Manager

The Internet Address Manager (IAM) module determines the IP address and other network settings during initialization. IAM can use either static values stored in NVRAM or protocols such as DHCP to query the network for a configuration.

You edit the `iamParams.c` and `iamCallbacks.c` files to control how IAM acquires network configuration parameters. The `iamParams.c` file is stored in the `platform` directory. The `iamCallbacks` file, stored in `src/bsp/customize`, normally does not need to be changed. If, however, you do need to change it, copy it in your platform's directory and edit the copy. The BSP source paths are set up so the copy in the `platform` directory is used instead of the copy in `src/bsp/customize`. For more information about IAM, see the online help.

For information about configuring TCP/IP memory usage, see the *NET+OS Programmers Guide*.

File system

You can configure the BSP to interface the C library file I/O functions to the file systems. The NET+OS development environment currently supports two file systems:

- **Native file system.** Used to create RAM volumes on RAM memory and flash volumes on non-removable flash memory.
- **FAT file system.** Used to create FAT volumes on removable media such as USB flash memory sticks.

Use these constants to configure the file systems:

Constant	Description
BSP_INCLUDE_FILESYSTEM_FOR_CLIBRARY	To include the native file system in the C library and create a RAM and flash volume as part of the BSP initialization process, set this constant in <code>bsp_fs.h</code> to TRUE.
BSP_NATIVE_FS_MAX_INODE_BLOCK_LIMIT	<p>When the BSP creates a native file system volume, this constant in <code>bsp_fs.h</code> specifies the percentage of the maximum number of inode blocks you can allocate to store inodes for a volume. Using this constant, you specify the upper limit of the number of blocks reserved to store inodes. Valid values are from 1 to 100.</p> <p>For more information, see the <code>NAFSinit_volume_cb</code> file system API function in the online help.</p>
BSP_NATIVE_FS_MAX_OPEN_DIRS	<p>When the BSP creates a native file system volume, this constant in <code>bsp_fs.h</code> specifies the maximum number of open directories the file system will track. A directory is considered open if any of its files are open. Valid values are from 1 - 64. For more information, see the native <code>NAFSinit_volume_cb</code> file system function in the online help.</p>
BSP_NATIVE_FS_MAX_OPEN_FILES_PER_DIR	<p>When the BSP creates a native file system volume, this constant in <code>bsp_fs.h</code> specifies the maximum number of open files per directory that the file system will track. Valid values are from 1 to 64.</p> <p>For more information, see the native <code>NAFSinit_volume_cb</code> file system function in the online help.</p>
BSP_NATIVE_FS_BLOCK_SIZE	<p>When the BSP creates a native file system volume, this constant in <code>bsp_fs.h</code> specifies the block size used for the volume. Valid values are:</p> <ul style="list-style-type: none"> ■ <code>NAFS_BLOCK_SIZE_512</code> ■ <code>NAFS_BLOCK_SIZE_1K</code> ■ <code>NAFS_BLOCK_SIZE_2K</code> ■ <code>NAFS_BLOCK_SIZE_4K</code>
BSP_NATIVE_FS_RAM0_VOLUME_SIZE	When the BSP creates the native file system RAM volume, this constant specifies the size of the RAM volume in bytes.

Constant	Description
BSP_NATIVE_FS_FLASH0_OPTIONS	<p>When the BSP creates the native file system flash volume, this constant specifies the advanced options to use. Valid values are the bitwise ORing of these options:</p> <ul style="list-style-type: none"> ■ NAFS_MOST_DIRTY_SECTOR — Uses the default sector transfer algorithm that selects the sector with the most dirty blocks. If you do not specify a sector transfer algorithm, or if you specify multiple sector transfer algorithms, the default algorithm is used. ■ NAFS_RANDOM_DIRTY_SECTOR — Uses the alternative sector transfer algorithm that randomly selects a sector with dirty blocks. ■ NAFS_TRACK_SECTOR_ERASES — Enables tracking the number of sector erases for each sector of a flash volume. ■ NAFS_BACKGROUND_COMPACTING — Enables the background sector compacting thread. This feature automatically reclaims the dirty blocks in the flash volumes and converts them to erased blocks. <p>For more information, see the <code>NAFSinit_volume_cb</code> native file system function in the online help.</p>
BSP_NATIVE_FS_FLASH0_COMPACTING_THRESHOLD	<p>If the <code>BSP_NATIVE_FS_FLASH0_OPTIONS</code> constant includes <code>NAFS_BACKGROUND_COMPACTING</code>, this constant specifies the percentage of erased blocks in a flash sector to gain to trigger the sector compacting process. Valid values are from 1 to 100.</p> <p>For more information, see the <code>NAFSinit_volume_cb</code> native file system function in the online help.</p>

Customizing the Bootloader

C H A P T E R 5

This chapter describes the NOR-Flash based bootloader utility and the ways in which you can customize it.

See the *Digi NET+OS U-Boot Reference Manual* for information on customizing U-Boot, used on NAND-Flash based products.

Overview

The NET+OS bootloader is executed immediately after the hardware is power cycled. The bootloader determines whether a valid application is stored in flash, as shown here:

If	The bootloader
A valid application is stored in flash	Copies (or decompresses) the application from flash to RAM and tries to execute
A valid application is <i>not</i> stored in flash	Tries to download a new image over the network or serial port.
A valid backup recovery application is stored in flash	Copies (or decompresses) the backup recovery application from flash to RAM and tries to execute.
A valid backup recovery application is <i>not</i> stored in flash	Tries to download a new image over the network.
The network download of the new image was <i>not</i> successful	Tries to download a new image over the serial port

If you are using the software debugger, the `bootloader` also is used to download images for debug over the network. When you use flash, the `bootloader` is stored in the first few sectors of flash and is loaded automatically by the processor when it powers up.

The NS9360, NS9750 and NS9215 chips can boot from SPI-EEPROM SPI flash devices. You enable SPI-EEPROM boot logic through bootstrap resistors. For details, see the hardware reference for your processor.

When boot logic is enabled, it copies the contents of SPI serial flash (or SPI-EEPROM) to system memory, allowing you to boot from low-cost serial memory. The CPU is held in reset while the data is copied. The boot logic interfaces to serial port B using the BBus to perform the transactions that are required to copy the boot code from SPI serial flash (or SPI-EEPROM) to external memory. For details about SDRAM settings, see “SPI Bootloader Overview” in the online help.

In either case, the `bootloader` is automatically copied from ROM to RAM by the hardware after a hard reset. The application image can be compressed to save space in serial flash.

In normal operation, the `bootloader` 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 is copied from flash to RAM by hardware and executed in RAM
- **RAM image.** The main body of the `bootloader`, 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 (or SPI EEPROM). The processor automatically copies the ROM image to RAM after a reset and immediately starts to execute the `bootloader` ROM image. The `bootloader` uses the BSP initialization code to configure the hardware. After the hardware is initialized, the ROM image decompresses the RAM image section of the `bootloader` to a different location in RAM and executes it.

RAM image

The RAM image is stored as an application image in flash (or SPI EEPROM). 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, `customize.ldr`.
- The application image must be built with the `boothdr` utility, which is located in `/bin`.

The RAM image of the `bootloader` determines whether the application image is valid by performing a checksum test on it. If the application image fails the checksum test, the RAM image attempts to recover by:

- Executing the backup recover application stored in flash

The RAM image uses these steps to perform the recovery:

- 1 Checks for valid header in the application image.
- 2 Gets the address of the backup recovery image from the application image header.
- 3 Checks for a valid backup recovery image by performing a checksum test.
- 4 Executes the backup recovery image if the image is valid.

The RAM image of the `bootloader` determines whether the backup recovery application image is valid by performing a checksum test on it. If the backup recovery application image fails the checksum test, the RAM image attempts to recover by:

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

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

If the RAM image of the bootloader is unable to download the application image through the TFTP recovery method, it attempts to recover by:

- Using the serial recovery method to download the application image

The RAM image uses these steps to perform the recovery:

- 1 Starts the serial recovery process to download the application from another device through the serial port.
- 2 Validates the downloaded application image by performing a CRC32 checksum
- 3 Stores the image into flash
- 4 Resets the unit, which restarts the process

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;
    WORD32 backupAddress;
    WORD32 reserved1;
    WORD32 reserved2;
    WORD32 reserved3;
    WORD32 reserved4;

} bImageHeaderType;
```

This table describes how the fields are used:

Field	Description
<i>headerSize</i>	Set to indicate the size of the complete header, including the application-specific section. The application starts immediately after the end of the header.
<i>naHeaderSize</i>	Set to indicate the size of the NET+OS portion of the image header in bytes, including this field.
<i>signature</i>	Set to the ASCII string bootHdr to identify this header as a valid image header.
<i>version</i>	Set to 1 for the NET+OS 7.4 and later versions of the image header.
<i>flags</i>	A bit field of flags. For details about bit values, see the next table.
<i>flashAddress</i>	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.
<i>ramAddress</i>	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.
<i>size</i>	Holds the size of the image (not including the header) in bytes.
<i>backupAddress</i>	Holds the address of the backup recovery image in Flash memory.
<i>reserved1</i>	Reserved for future use.
<i>reserved2</i>	Reserved for future use.
<i>reserved3</i>	Reserved for future use.
<i>reserved4</i>	Reserved for future use.

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

Bit value	Description
BL_WRITE_TO_FLASH	If you set this bit, the image is written to the address in flash specified in the <i>flashAddress</i> field. If you clear this bit, the image is run immediately without writing it to flash. The image is moved or decompressed to the address in the <i>ramAddress</i> field before it is executed.
BL_LZSS_COMPRESSED	If you set this bit, the application portion of the image is compressed. It is decompressed to the address in the <i>ramAddress</i> field before it is executed.
BL_LZSS2_COMPRESSED	If you set this bit, the application portion of the image is compressed using the LZSS2 compression algorithm. It is decompressed to the address in the <i>ramAddress</i> field before it is executed.
BL_EXECUTE_FROM_ROM	If you set this bit, the application is executed from ROM. The application must not be compressed. If you do not set this bit, the application is decompressed or moved to the address in the <i>ramAddress</i> field before it is executed.
BL_BYPASS_CRC_CHECK	If you set this bit, the application image is executed without first performing the CRC32 checksum test to determine if the image is valid. This option allows for faster application image boot times. However, if the application image is corrupted, none of the recovery methods will be executed.

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 application 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 the NET+OS development environment 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 `bsp/platforms/my_platform` directory.

This table describes the keyword/value pairs:

Keyword	Value description
WriteToFlash	Set to one of these options: <ul style="list-style-type: none"> ■ Yes. Sets the <code>BL_WRITE_TO_FLASH</code> bit in the <i>flags</i> field of the header. ■ No. The bit is left clear.
Compressed	Set to one of these options: <ul style="list-style-type: none"> ■ Yes. Sets the <code>BL_LZSS2_COMPRESSED</code> bit in the <i>flags</i> field of the header. ■ No. The bit is left clear.
ExecutedFromRom	Set to one of these options: <ul style="list-style-type: none"> ■ Yes. Sets the <code>BL_EXECUTE_FROM_ROM</code> bit in the <i>flags</i> 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 <code>0x</code> .
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 <code>0x</code> .
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 <code>0x</code> .

Keyword	Value description
BypassCrcCheck	Set to one of these options: <ul style="list-style-type: none"> ■ Yes. Sets the BL_Bypass_CRC_CHECK bit in the <i>flags</i> field of the header. ■ No. The bit is left clear.
BackupAddress	Specifies the address of the backup recovery application image stored in Flash memory. 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	0x4000
MaxFileSize	0xD0000
BypassCrcCheck	No
BackupAddress	0x2130000

General bootloader limitations

Be aware of these general limitations of 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 file name, 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, they must have the same IP address.
- IPv4 only.

Customizing the bootloader utility

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

The code in `blmain.c` (or `spi_blmain.c` for the SPI bootloader) 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
<code>NABReportError</code>	Called whenever an error occurs
<code>getMacAddress</code>	Gets the Ethernet MAC address the <code>bootloader</code> should use
<code>isImageValid</code>	Determines whether an image is valid
<code>shouldDownloadImage</code>	Determines whether the <code>bootloader</code> should download a new image
<code>getDefaultFilename</code>	Determines the name of the file to download
<code>downloadImage</code>	Downloads a new application image
<code>customizedApplicationImageHandler</code>	User implemented application image processing algorithm

NABIReportError

Called when an error is detected.

The error is reported to the user.

Format

```
void NABIReportError (errorCode);
```

Arguments

Argument	Description
<i>errorCode</i>	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.

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 (void);
```

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, int imageIsInRAM)
```

Arguments

Value	Description
<i>imageInfo</i>	Pointer to the image header
<i>imageIsInRam</i>	Either of these: <ul style="list-style-type: none"> ■ Non-zero. The image is currently in RAM. ■ Zero. The image is currently in serial flash.

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. If the image is not in RAM, `isImageValid` first reads the image in serial flash into RAM.

You can extend the default implementation to determine whether the application can and should run on the hardware; for example, by doing one, some, or all these steps:

- 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.

```
static BOOLEAN shouldDownloadImage(void)
{
    #if (BSP_BOOTLOADER_BOOT_FROM_NETWORK_ONLY == TRUE)
        return TRUE;
    #else
        int result = TRUE;
        blImageHeaderType imageInfo;
        memset(&imageInfo, 0, sizeof(blImageHeaderType));
        if (blReadFromSFlash(NAAppOffsetInSFlash, (char *)&dlBuffer[0], sizeof (blImageHeaderType), 0)
            != BL_SUCCESS)
            NABReportError(SIMPLE_SPI_EEPROM_READ_FAIL);
        fmemcpy(&imageInfo, &dlBuffer[0], sizeof (blImageHeaderType));
        result = (isImageValid(&imageInfo, 0/*image is in EEPROM*/) == FALSE);
        return result;
    #endif
}
```


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.

To boot from the network only, set `BSP_BOOTLOADER_BOOT_FROM_NETWORK_ONLY` to `TRUE`. The function always returns `TRUE` without checking whether the image in flash is valid.

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 file name 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 file name to the DHCP/BOOTP server, using one of these approaches:

- Hard-coding a file name that identifies the product
- Determining the features supported by the hardware and generating a file name that has this information encoded in it
- Generating a file name 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
<i>destination</i>	Pointer to the memory buffer that will hold the image
<i>maxLength</i>	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

customizedApplicationImageHandler

User implemented bootloader Flash application image processing routine

Format

```
void customizedApplicationImageHandler (void);
```

Arguments

None

Return Values

None

Implementation

This function allows the user to implement a customized Flash application image processing algorithm. When `#define BSP_BOOTLOADER_RECOVERY_METHOD` is set to `BSP_BOOTLOADER_CUSTOMIZED_RECOVERY` in the platform's `bsp_bldr.h` file, the bootloader will execute the Flash application image processing algorithm in `customizedApplicationImageHandler()` and bypass the default bootloader Flash application image processing algorithm. This customized algorithm must execute an application image if the image is valid and perform an application image recovery if the image is invalid. The application image recovery process can include any of the default bootloader recovery methods.

Part 2: Hardware

Bringing Up New Hardware

C H A P T E R 6

This chapter describes how to bring up new application hardware.

Verify the debugger initialization files

When you use the debugger, you initialize hardware registers on the board that the BSP ROM startup code would normally set up. You can use debugger initialization scripts for this task. The script contains commands that the debugger executes before the application is downloaded and executed.

The NET+OS development environment ships with debugger scripts, located in `netos/debugger_files`, that initialize the supported Digi Connect products and development boards. You must verify that the NET+OS debugger script still works with your application hardware; if it does not, you must create a new debugger script.

NET+OS supports the DIGI JTAG Link debugger. Additionally, support is available in most platforms for Macraigor Raven and Mentor Graphics MAJIC/MAJICO.

The information in this chapter is based on the assumption that you are using one of these debuggers. The `.gdbinit` files generated by NET+OS application `Makefiles` are based on files in `netos/debugger_files`.

The files listed next contain the commands that set up the memory controller to support the SDRAM on the development boards. You must do this before you can download application code into the board's RAM.

Review the file for the processor you are using, and verify that it sets up the memory controller correctly for the SDRAM that your application hardware uses.

This debugger script	Initializes
<code>.gdbinit.ns7520</code>	NS7520 registers
<code>.gdbinit.ns9360</code>	NS9360 registers
<code>.gdbinit.ns9750</code>	NS9750 registers
<code>.gdbinit.ns9215</code>	NS9210/NS9215 Registers

If the script listed in the table does not set up memory correctly for your board, create a new one with commands to do so. You create a new initialization script by copying the one for your processor and editing it as needed for your application hardware. Note that the memory controller set up

on the NS9210 is the same as the NS9215, so the `.gdbinit.ns9215` scripts serve as an example for both.

- **NET+50.** Configure the PLL to the correct clock speed by setting `PLLCR`.
- **ARM7 processors.** Configure the System Control register to set the correct bus speed and endianness and to disable the watchdog timer.
- **ARM7 processors.** Set the valid bit in the `CS0` chip select to 0. The BSP checks this bit to determine whether a debugger is being used.

This is important because the BSP needs to know whether to configure the RAM chip selects, perform a memory test, and turn on cache.

After you create the initialization script, edit the application `Makefiles` to use your new script whenever you select your platform. This table lists the `Makefiles` in `src/linkerscripts` with the platforms they are used for.

Makefile	Supported processors
<code>Makefile.appbuild.ns9360</code>	NS9360
<code>Makefile.appbuild.ns9750</code>	NS9750
<code>Makefile.appbuild.original</code>	NET+50 and NS7520
<code>Makefile.appbuild.ns9215</code>	NS9215
<code>Makefile.appbuild.ns9210</code>	NS9210

Edit the code in the `Makefile` that handles the `gdbinit` target to use your startup script when your platform is selected. You can do this by using an `if` statement that examines the value of the `$(PLATFORM)` `Makefile` variable.

Using the MAJIC/MAJICO probe

The debugger initialization scripts `ns9xxx.cmd` and `ns9215.cmd` are 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`.

This table shows the debugger initialization files that the MDI server uses:

File name	Contents
startice.cmd	JTAG settings and reads in the ns9xxx.cmd file to initialize the target board
ns9xxx.cmd	This file contains the commands to initialize SDRAM on NS9360 and NS9750 based development boards and modules, which use 32-bit SDRAM.
ns9215.cmd	This file contains the commands to initialize SDRAM on the connectcore9p9215_a module, which uses 16-bit SDRAM. The NS9215 and NS9210 both use the same memory controller, so this file can also be used as an example of a configuration file for a NS9210-based device.
epimdi.cfg	MAJIC settings, including 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 ns9xxx.cmd or ns9215.cmd file. This file programs the registers in the memory controller. For a detailed description of these registers, see the hardware reference for the processor you are using.

Debug the initialization code

After you complete the modifications and create the debugger initialization scripts for your application hardware, you may need to debug the initialization code. To debug code from RAM, you use the debugger 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 From the bsp directory, rebuild the BSP with your changes:

- Change to the BSP directory:
`cd src/bsp`
- Enter this command:
`make PLATFORM=my_platform`
where `my_platform` is the name of your platform.
(Instead of entering `make PLATFORM=my_platform`, you can set the bash shell variable by entering `export PLATFORM=my_platform`; then you can build the BSP by entering just `make`.)
- 2 Disable the POST by setting the `APP_POST` constant in the `root.c` file to 0.
- 3 Carefully review all the settings in the `appconf.h` file.
- 4 Build the application:
 - Copy the template application, which is located in:
`/apps/template`
 - In the `src/apps/template/32b` directory, enter:
`make PLATFORM=my_platform clean`
`make PLATFORM=my_platform all`
- 5 Start up the debugger software; for example:
 - GDB server for the Digi JTAG Link
 - `ocdRemote` for the Macraigor Raven
 - `mdi server` for the Mentor Graphics MAJIC/MAGICO
- 6 From the `/src/apps/template/32b` directory, enter this command:
`gdbtk -se image.elf`
- 7 To load your image, from the gdb console window, enter:
`lo image.elf`
- 8 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.
- 9 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.
 - All interrupts are masked off.
 - On NET+50 platforms, the PLL registers are properly programmed for the crystal on your application hardware.

- You can read and write RAM on your application board.
- 10** Debug the initialization code by stepping through it, as described in the next section.

Debugging the initialization code on ARM7 platforms

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

- 1** INIT.s file
- 2** ncc_init() routine
- 3** NABoardInit routine
- 4** Ethernet driver startup

Be aware that this section describes debugging from RAM. You also may need to step through the INIT.s code when it runs from ROM.

Debug the INIT.s file

The `src/bsp/init/arm7/INIT.s` file 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 code in `INIT.s` must perform this process:

- 1** Set the processor mode and disable all interrupts.
- 2** Initialize the PLL (NET+50 only).
- 3** Set the `BSPEED` field in the System Control register to enable full bus speed.
- 4** Execute a soft reset.
- 5** Place the DMA controller into test mode.

This action causes the on-chip static RAM (normally used to store DMA context information and register values) to become available as RAM.

- 6** Set the SVC stack pointer to point to the DMA RAM.
- 7** Call the `ncc_init` routine to continue the initialization process.
- 8** Set up stacks for all processor modes.
- 9** Release the DMA controller from test modes.

10 Call the C library startup routines.

The routines do not return.

Debug the `ncc_init` routine

The `ncc_init` routine 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 6), you need to check `ncc_init` and your customized routines to verify that they are working correctly. The `ncc_init.c` file is in `bsp/init/arm7`.

The `ncc_init` routine must perform this process:

- 1** Set up the Memory Management Control register by calling:
`customizeSetupMMCR`
- 2** Set up the System Control register by calling:
`customizeGetScr`
- 3** Determine whether a software restart has occurred by examining the contents of UNDEF mode R14.
The `Restart` function sets this register when the system is restarted.
- 4** Determine whether a debugger is attached.
The debugger script files indicate the presence of a debugger by clearing the valid bit for chip select 0 (CS0).

- 5 Set up the GPIO ports by calling the `customizeSetupPortX` routines.
- 6 Set up CS0 by calling `customizeSetupCS0`.
- 7 If a debugger is detected, call `customizeSetupCS3` to set up CS3, and call `customizeGetRamSize` to determine the amount of RAM on the system.
- 8 Call the `customizeReadPowerOnButtons` function to read and save the state of buttons and jumpers.
- 9 Verify that the application can fit in the available RAM.
- 10 Set flags in memory, which is now set up, to indicate whether a debugger is present and whether a software restart has occurred.

Debug the NABoardInit routine

The `NABoardInit` routine, located in `src/bsp/init/arm7/narmbrd.c`, 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 In `eth_reset.c`, put a breakpoint on the `eth_reset` routine, and let the program run until you reach the breakpoint.
- 2 In the `mii.c` file, step into the `customizeMiiReset` routine and then into `customizeMiiIdentifyPhy`.
- 3 Verify that:
 - `customizeMiiIdentifyPhy` returns a value not equal to `0xffff`. `-mii_reset` returns 0.
 - `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.

Debugging the initialization code on ARM9 platforms

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

- 1 `init.arm` file
- 2 `nccInit` routine
- 3 `NABoardInit` routine
- 4 Ethernet driver startup

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

Debug the `init.arm` file

The `init.arm` file, located in `src/bsp/init/arm9`, performs initialization functions. Step through the code in `init.arm`, 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 `Reset_Handler` in the `init.arm` file. If your board is not working, set a breakpoint on the `Reset_Handler` routine and step through it.

Debug the `nccInit` routine

The `nccInit` routine, located in `bsp/init/arm9/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.

If you have difficulty starting the development board, use these diagnostic tools:

- A simple serial driver that is loaded in `nccInit`.

- `mprintf`, a special `printf` routine. A prototype 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, located in `src/bsp/init/arm9`, 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 In `eth_reset.c`, put a breakpoint on the `eth_reset` routine, and let the program run until you reach the breakpoint.
 - 2 In the `mii.c` file, step into the `customizeMiiReset` routine, and then into `customizeMiiIdentifyPhy`.
 - 3 Verify that:
 - `customizeMiiIdentifyPhy` returns a value not equal to `0xffff`.
 - `mii_reset` returns `0`.
 - `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.

Memory Map

C H A P T E R 7

This chapter discusses the memory maps for ARM7- and ARM9-based modules.

Memory aliasing (ARM7)

NS7520 modules have this memory map:

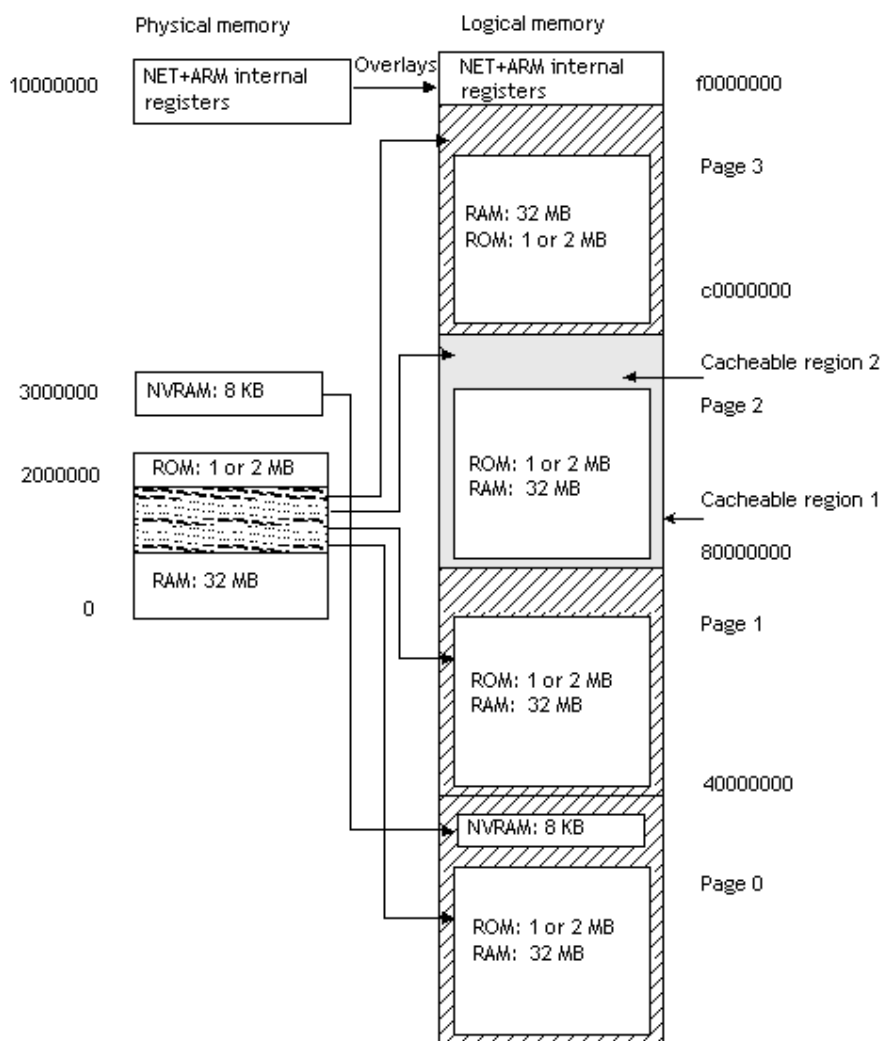
- Addresses from 0xf0000000 to 0xffffffff are reserved for devices internal to the NET+ARM processor.
- RAM on CS1 and CS2 is mapped from address 0x0 to 0x01ffffff.
- ROM on CS0 is mapped from address 0x02000000 to 0x021fffff.
- NVRAM on CS3 is mapped from address 0x03000000 to 0x03001fff.

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

The NET+OS aliases physical memory to four locations in the address map, so each physical word of memory appears at four addresses. The aliasing is done on all platforms. NET+OS configures one aliased copy of memory for instruction cache on platforms that support cache (the NET+50). Code is executed from this area of the address map to improve performance. NET+OS uses uncached areas for general data storage.

In the next figure, which shows the NET+OS memory map with cache enabled:

- Physical memory is mapped four times in logical memory.
- The NET+ARM internal registers appear once.
- Logical page 2 is used for instruction cache.
- All addresses are in hexadecimal notation.



Page 0 contains a slot for up to 32 MB of RAM (using CS1 and C2) at addresses 0x0 through 0x1ffffff.

Either 1 or 2 MB of flash memory on CS0 begins at 0x2000000, and 8 KB of NVRAM starts at 0x3000000.

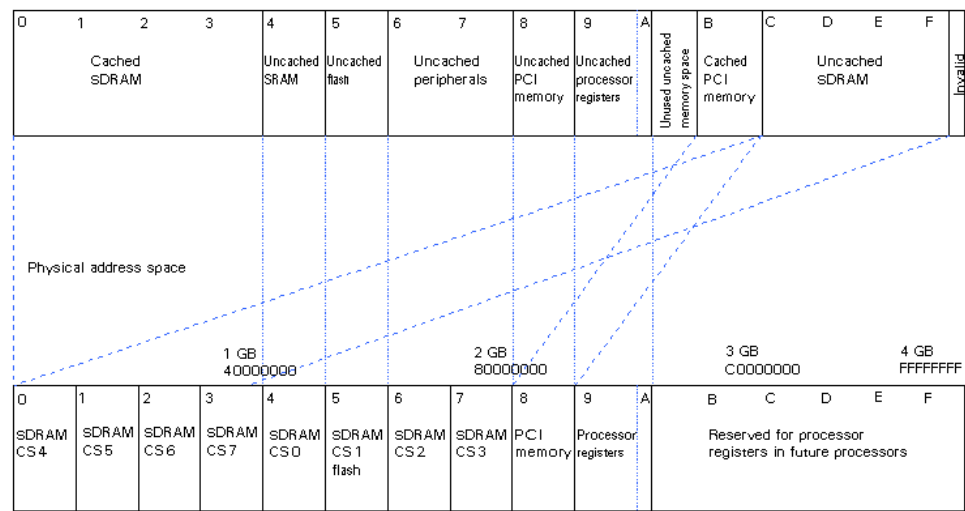
The ROM and RAM spaces are remapped on pages 1, 2, and 3; for example:

Physical address	Which is	Can b accessed at
0x100	RAM	0x4000100, 0x80001000, and 0C000100
0x20000100	Flash ROM	0x60001000, 0xA000100, and 0xE000100
0x30000100	NVRAM	0x3000100 only

Memory map (ARM9)

The ARM9 processors (NS9210, NS9215, NS9360, NS975) have an embedded MMU. The MMU allows you to remap physical addresses to virtual addresses. For simplicity, NET+OS sets most virtual addresses to be the same as their physical addresses. The exception to this is RAM, which is located at physical address 0x0 and is mapped to two different virtual addresses: 0x0 and 0xC000000. Accesses to RAM at virtual address zero use the write-back cache mode. Accesses to RAM at virtual address 0xC0000000 are not cached.

The next illustration shows the NET+OS memory map for the ARM9-based modules. Note that the NS9750 is the only processor that has the PCI address space.



In this diagram:

- The top half shows the virtual address space seen by the CPU and the software.
- The bottom half shows the actual physical address space.
- The first GB of memory is set up as a cached region of memory; this is the address space in which all applications run (Stack, bss data, heap).

The 3-4 GB range is set up for non-cached memory and is mapped to the 0-1 GB of physical memory. The end of the 4 GB range is set up as invalid because these are the addresses of registers in the NET+50 and the NS7520 processors that no longer exist. PCI memory also is mapped to a cached and non-cached region.

All applications use the 0-1 GB range of addresses, which is set up as write-back cache. NET+OS drivers typically use the 3-4 GB to store DMA buffer descriptors that should not be cached.

You usually need to access the uncached region only if you are writing drivers that use DMA; typical applications never need to use this region.

Adding memory or memory mapped devices to an ARM9-based processor requires updates to the MMU Table. See the MMU Driver Device Driver, under the Hardware/Board Support tab in the NET+OS API Reference.

Adding Flash

C H A P T E R 8

This chapter describes how to update the NET+OS flash driver to support additional flash parts.

Overview

The NET+OS development environment 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:/netos/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.

Supported flash memory parts

NET+OS supports these flash memory parts:

Manufacturer	Part number
AMD	AM29F800BB
AMD	AM29F800BT
AMD	AM29DL323DB
AMD	AM29LV160T
AMD	AM20LV160B
AMD	AMD29DL323DT
AMD	AM29DL323DTB
AMD	AM29LV128M
AMD	AM19LV160B
AMD	AM27LV320DB
AMD	AM29LV641MH
AMD	AM29LV641
AMD	AM29LV800BB
AMD	AM29LV800BT
AMD	AM29DW641F
Atmel	AT49BV8011

Manufacturer	Part number
Atmel	AT49BV8011T
Atmel	AT49BV1614A
Fujitsu	MBM29LV800BE
Fujitsu	MBM29LV160B
Fujitsu	MBM29LV160T
Macronix	MX28F4000
Sharp	H28F800SG
SST	28SF040
SST	39VF800
STM	M29W800AB
STM	M29W160DB
STM	M29W320DB
STM	M29DW641F
STM	M29W160B
STM	M29W160T
STM	M29W320EB

If your platform uses one of these flash parts, then you should edit the file `flashparts.h` in your platform directory. This file contains a list of macro definitions, one for each flash part. Set the definitions for the flash parts your platform uses to `TRUE` to enable support for those parts. Set the definitions for flash parts which your platform does not use to `FALSE` to reduce memory usage. For example, if your platform uses the `AM29LV800BB` part, then set `NAFLASH_WANT_TO_SUPPORT_AM20LV800BB` to `TRUE`.

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
<code>mcode</code>	Manufacturer's code
<code>mcode_addr</code>	Address of manufacturer's code
<code>dcode</code>	Device code
<code>dcode_addr</code>	Address of device code
<code>total_sector_number</code>	Total number of sectors
<code>sector_size</code>	Size of sector (in bytes)
<code>prog_size</code>	Program load size (in bytes)
<code>access_time</code>	Access time (in nanoseconds)
<code>id_enter_cmd</code>	Pointer to the enter identify flash command
<code>Id_enter_len</code>	Number of cycles for the enter identify flash command
<code>id_exit_cmd</code>	Pointer to the exit identify flash command
<code>id_exit_len</code>	Number of cycles for the exit identify flash command
<code>erase_cmd</code>	Pointer to the erase flash command
<code>erase_len</code>	Number of cycles for the erase flash command
<code>write_cmd</code>	Pointer to the write flash command
<code>write_len</code>	Number of cycles for the write flash command
<code>sector_erase_cmd</code>	For AMD only
<code>sector_size_array</code>	For non-uniform sector sizes

Adding new flash

When you add support for new flash memory, 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 memory, 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 memory:

- 1 In the `flash.h` file, add the definitions for the new flash device.
- 2 In the `flashparts.h` file in your platform directory, add a macro definition for your flash part which can be used to control whether support for it is built into the library. For example, the macro definition `NAFLASH_WANT_TO_SUPPORT_M29W800AB` is used for the STM M29W800AB.
- 3 In the `netos/src/bsp/common/flashparts.c` file, 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. Surround the table for your part with if statements that use the macro definition you created in step 2. Use the other table definitions as examples.
- 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.)
- 5 Rebuild the BSP by typing “make PLATFORM=your_platform” in the `netos/src/bsp` directory. This will update the copy of the `flash_id_table` in the BSP directory.

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 step.

- 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 driver by typing "make PLATFORM=your_platform" in the `netos/src/flash` directory.

Hardware Dependencies for ARM7-based Modules

C H A P T E R 9

This chapter discusses the NET+OS hardware dependencies for modules that use the NS7520 processor.

Overview

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

- DMA channels
- Serial ports
- Software watchdog
- Endianness
- System clock and timers
- Interrupts

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

DMA channels

This table describes how each of the 13 DMA channels is used in porting NET+OS:

Channel	Used by	What it does
1	Ethernet driver	Moves data from the Ethernet receiver to memory. The Ethernet driver code is in the bsp/devices/ethernet directory.
2	Ethernet driver	Moves data from memory to the Ethernet transmitter.
3 - 6	External peripherals (NS7250)	NS7250 processor only. Only two channels — either 3 and 5 or 4 and 6 — can be configured at one time.
7 and 8	Serial/SPI driver	Receives data
9 and 10	Serial/SPI driver	Transmits data.
11 - 13		Moves data from memory to memory (NS7520 processor only)

Serial ports

The BSP normally sets up both serial ports to support asynchronous RS-232-style communications. This includes:

- `BSP_SERIAL_PORT_x`, defined as `BSP_SERIAL_UART_DRIVERS` in `bsp_serial.h`
- `BSP_GPIO_MUX_SERIAL_y` to a UART selection in `gpio.h`

To use the serial peripheral interface (SPI) controller:

- In `bsp_serial.h`, redefine `BSP_SERIAL_PORT_x` to `BSP_SERIAL_SPI_DRIVER`
- In `gpio.h`, redefine `BSP_GPIO_MUX_SERIAL_y` to `BSP_GPIO_MUX_serial_SPI_MASTER`

Software watchdog

The watchdog device driver uses the internal watchdog if `BSP_WATCHDOG_TYPE` is set to `BSP_WATCHDOG_INTERNAL` in `bsp_sys.h`.

The `NAReset` routine in the `nareset.c` file uses the software watchdog to reset the system. `NAReset` is called by the default implementation of `customizeReset` in `gpio.c`.

Endianness

The BSP supports big endian mode only.

System timers

The code that supports the system timers is in the `bsptimer.c` file. The two timers are described next.

Timer 1

The BSP uses Timer 1 as the system heartbeat clock. The kernel uses the system heartbeat clock for timing and pre-emption of tasks.

The frequency of the system heartbeat clock is controlled by the `BSP_TICKS_PER_SECOND` constant in the `bsp_sys.h` file. The recommended value, is between 1 and 1000. A value of 100, for example, provides a heartbeat rate of one tick every ten milliseconds.

Timer 2

This timer is available for use by the application.

Interrupts

This table describes how interrupt levels are used in the BSP:

Interrupt level	Use
31 (DMA 1)	Ethernet driver receive packet interrupt
30 (DMA 2)	Ethernet driver packet done interrupt
29 (DMA 3)	ENI FIFO receive packet interrupt
28 (DMA 4)	ENI FIFO transmit packet interrupt
27 and 26 (DMA 5 and 6)	Not used
25 (DMA 7)	■ HDLC driver channel 1 receive frame interrupt ■ Serial/SPI 1 DMA mode receive interrupt
24 (DMA 8)	■ HDLC driver channel 1 receive frame interrupt ■ Serial/SPI 1 DMA mode receive interrupt
23 (DMA 9)	■ HDLC driver channel 2 receive frame interrupt ■ Serial/SPI 2 receive interrupt
22 (DMA 10)	■ HDLC driver channel 2 transmit frame interrupt ■ Serial/SPI 2 transmit interrupt
21-17 (ENI ports 1-4 and ENET RX)	Not used
16 (ENET TX)	Ethernet driver transmit interrupt
15 (SER 1 RX)	Serial/SPI driver port 1 receive interrupt

Interrupt level	Use
14 (SER 1 TX)	Serial/SPI driver port 1 transmit interrupt
13 (SER 2 RX)	Serial/SPI driver port 2 receive interrupt
12 (SER 2 TX)	Serial/SPI driver port 2 transmit interrupt
11 through 6	Not used
5 (Timer 1)	System clock tick interrupt
4 (Timer 2)	Not used
3 through 0 (PORTC)	Not used

RS-232-style communications

To use the serial peripheral interface (SPI) controller, disable the serial driver, by undefining `BSP_INCLUDE_SERIAL_DRIVER1` and `BSP_INCLUDE_SERIAL_DRIVER2` in the `bsp_serial.h` file.

Hardware Dependencies for ARM9-based Modules

C H A P T E R 1 0

This chapter discusses NET+OS hardware dependencies for modules that use the NS9360, NS9210, NS9215 and NS9750 processors.

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
- Endianness
- Timers
- Interrupts
- Memory map

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

DMA channels on the NS9750 and NS9360 Processors

The NS9750 and NS9360 processors use three DMA controllers. Two of them exist on the Bbus, and one exists in the Bbus Bridge module. (For detailed information, see the hardware reference for your processor.)

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 DMA channels. These channels can be used for memory-to-memory transfers on both the NS9750 and NS9360 processors and for transfers between memory and an external device on the NS9360 processor. NET+OS does not use these channels. Your application can use the AHB DMA channels.

DMA Channels on the NS9210 and NS9215 Processors

The NS9210 and NS9215 processors use three DMA controllers. One controller supports the Ethernet controller, one controller supports I/O hub devices and one controller supports DMA transfers to external memory. See Hardware Reference manuals for these processors for more information.

Besides support external DMA transfers, external DMA channel 1 is also used by the AES encryption/decryption module. The NET+OS development environment uses AES with SSL and IPSEC. Therefore external DMA channel 1 is not available for use.

Note that the A/D converter and UART D both use the same I/O hub DMA channel. Therefore, only one of these devices can use DMA at a time. NET+OS requires that you disable the driver for UART D if you use the NET+OS A/D driver. The driver will call `__panic` if it determines that the serial driver has been configured to support UART D.

Endianness

The BSP supports big endian mode only.

General purpose timers

This section describes how the general purpose timers are used.

System timers

The statistical profiler uses timer 2 and the FIQ interrupt. These resources are used only if the statistical profiler is enabled by building the BSP with the constant `STATISTICAL_PROFILER` defined. These resources are available for application use if this constant is not defined.

The system clock, `NAuWait` and `NAWait` routines, and the USB device driver each use the first available timer.

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 modify the priority of the interrupts by editing the `NAAhbPriorityTab` and `NABbusPriorityTab` tables in `bsp.c`.

On the NS9360 and NS9750, the Bbus peripherals—all four serial ports, the USB device, and the 1284—combine all their interrupts into one Bbus Aggregate interrupt. These interrupt priorities are set by the table `NABbusPriorityTab` in `bsp.c`. All Bbus interrupts are multiplexed into a single AHB interrupt—the BBus Aggregate Interrupt.

For information about the interrupt controller, see the Hardware Reference Manual for your processors.

System clock

The `NA_ARM9_INPUT_FREQUENCY` constant in `sysClock.h` must be set to the frequency of the signal input to the `X1_SYS_OSC` pin. This is the clock source to the PLL when the PLL is used. If the PLL is bypassed, this signal is divided by 2 to generate the ARM9 CPU clock.

The processor automatically determines the PLL divisor values from hardware bootstrap settings when you use the PLL.

Part 3: Makefiles

NET+OS Makefile System

C H A P T E R 1 1

This chapter describes how to use the NET+OS `Makefiles` to rebuild the NET+OS BSP and libraries.

Overview

You use the `Makefile` system to build the BSP, POSIX, flash libraries, bootloader images, and example applications. This chapter describes the hierarchy of the `Makefile` and how to build, clean, and add libraries. This chapter also describes the bootloader `Makefile` and provides examples of building NET+OS libraries.

You initiate makes in any directory with a `Makefile` by entering this command:

```
make PLATFORM = my_platform
```

where `PLATFORM` is a bash shell variable that you can specify in either the `make` command line or the bash shell.

If you are doing frequent makes, it's convenient to set the shell variable and then enter just `make`. To set the `PLATFORM` shell variable, enter:

```
export PLATFORM=my_platform
```

To view the current value of your `PLATFORM` variable, enter this command:

```
echo $PLATFORM
```

Be aware that you must always specify the platform when you build NET+OS. The `PLATFORM` variable directs `make` to where to get the libraries and which platform directory to build in the BSP.

This list shows the supported platforms:

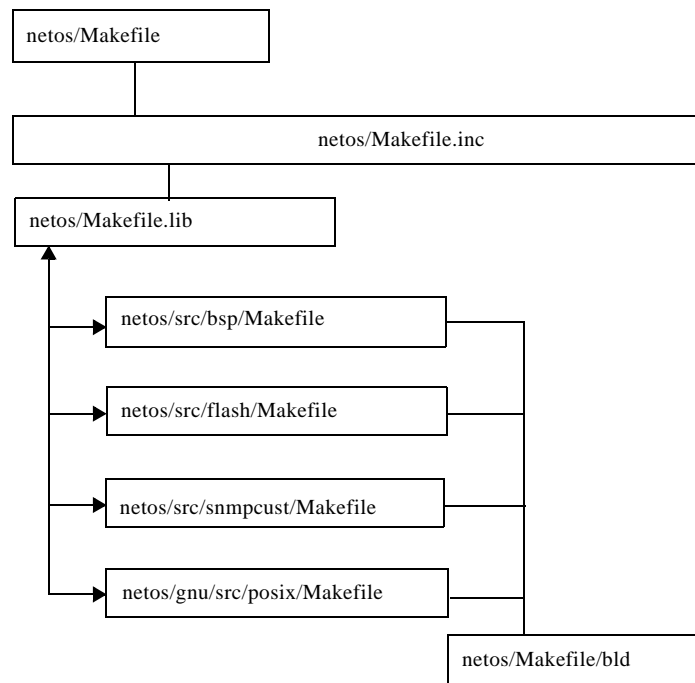
- | | |
|-----------------------|-----------------------|
| ■ connectcore9c_a | ■ connectwime |
| ■ connectcorewi9c_a | ■ net50bga_a |
| ■ connectcore9p9360_a | ■ connectcore9p9215_a |
| ■ connectem | |
| ■ connectme | ■ ns7520_a |
| ■ connectsp | ■ ns9360_a |
| ■ connectwiem | ■ ns9750_a |
| ■ connectme9210 | ■ connect wisp |

Two optional variables are:

- `DEBUG=on` — Turns on the debugging information.
By default, this variable is set to `off`. When this variable is set to `on`, the `NETOS_DEBUG` flag is turned on; `NETOS_DEBUG` turns on the `NA_ASSERT` macro.
- `MODE=verbose` — echoes out each Makefile command.
By default, this variable is set to `silent`. To see the compile line, turn on this variable.
This variable is useful if a problem with search paths occurs or if you want to check which compiler flags are turned on.

Makefile hierarchy

You can execute `make` from any directory that contains a Makefile. The Makefile system is nested and built around many Makefiles. This call graph shows how the system is organized; the list after the call graph describes the directories.



- netos/Makefile.inc, the master Makefile, uses the list of directories in the Makefile.lib file.
- Makefile.inc goes into each directory in Make.lib and executes the make commands.
- netos/Makefile.inc has the top level platform-specific settings, flags for compilation, link options, and processor-specific settings.

Building all libraries

► To build all the libraries, including the BSP:

- 1 Change to the root directory:

```
cd netos
```
- 2 Do one:
 - *Either* enter this command:

```
make PLATFORM=my_platform
```
 - *Or* enter these commands:

```
export PLATFORM=my_platform
make
```

Building individual libraries

To build a single library, you can go into the individual libraries, enter `make`, and specify the platform.

For example, to create `libbsp.a` for the `ns9360_a`:

- 1 Change to this directory:

```
netos/src/bsp
```
- 2 Do one:
 - *Either* enter this command:

```
make PLATFORM=ns9360_a
```
 - *Or* enter these commands:

```
export PLATFORM=ns9360_a
make
```

Library directory structure

The NET+OS library directory structure is keyed off the CPU, ENDIAN, and TOOLSET variables. This is the library directory structure:

```
netos/lib/arm7/32b/gnu/  
netos/lib/arm9/32b/gnu/
```

Except for the BSP library, libbsp.a, all libraries are found in the path shown in the previous paragraph. Because libbsp.a is the only platform-dependent library, the BSP library directory is keyed off the PLATFORM variable; for example:

```
netos/lib/arm7/32b/gnu/bsp/connectme/  
netos/lib/arm9/32b/gnu/bsp/ns9360_a/
```

In addition to libbsp.a, the BSP lib directory contains reset.o and memcpy.o objects. These objects, which are linked in with the application, provide a fast memory copy routine and the vector table.

Library Makefile variables

The child library Makefiles contain the name of the object files, include path, source path, and any other defines you want to pass to the compiler. These Makefiles are used to build the bsp, flash, sflash, and posix libraries.

This table lists the variables you need to define in the child Makefile:

Variable	Description
OBJS	Location of object files.
SRCDIR	List of source directories.
LOCAL LIB	Name of the library.
INCDIRS	List of include paths with -I prefix.
C OBJ	List of C object files.
OBJ	List of assembly object files.
OBJ	List of C++ objects (*.cc and *.cxx).
MY_CFLAGS	List of C flags (such as -Werror, -Os). To turn on debug information, set the -g flag here.
MY_DEFINES	List of defines with the -D prefix

Adding new libraries to the system

► To add new libraries:

- 1 Copy a child Makefile from `netos/src/flash`, and use it as a template.
- 2 Carefully set up the variables described in the previous table.
- 3 Add this directory to the list of libraries in the `netos/Makefile.lib` file, which has the list of directories that will be built when you enter `make`.

You don't need to change any other top-level Makefiles.

Cleaning libraries

When you clean the libraries, temporary files such as objects and dependency files are deleted.

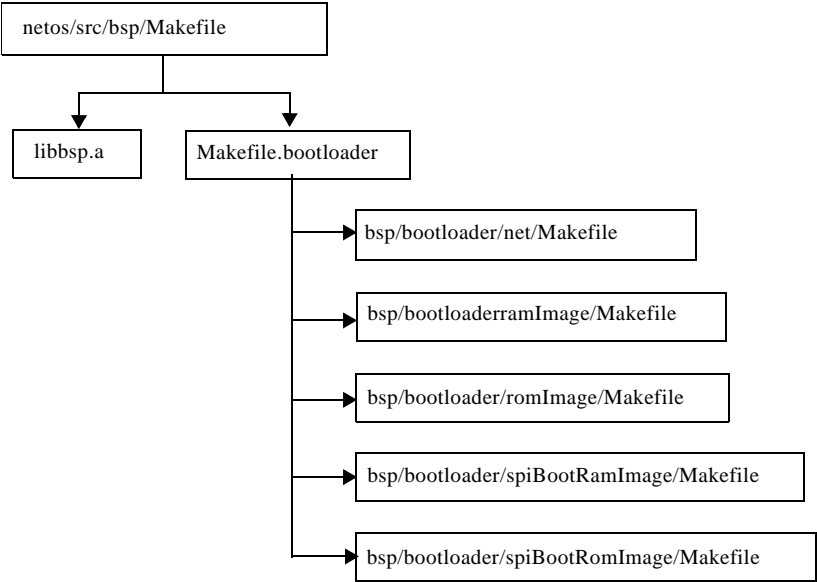
To clean libraries, enter this command:

```
make PLATFORM=my_platform clean
```

The next time you build after a `clean`, all the object files are rebuilt and archived into the library.

Bootloader Makefile

Because bootloader Makefiles are called from `netos/src/bsp/Makefile`, you always build the bootloader as part of the BSP. The next diagram shows how the bootloader Makefile system is organized.



This table describes the directories:

Directory	Description
netos/src/bsp/bootloader/net/Makefile	Generates the libnet.lib directory under bootloader/libs/gnu/my_platform/
netos/src/bsp/bootloader/ramImage/Makefile	Generates blram.bin to a converted file that is linked to rom.bin
netos/src/bsp/bootloader/romImage/Makefile	Generates rom.bin under netos/src/bsp/platforms/my_platform/
netos/src/bsp/bootloader/spiBootRamImage/Makefile	Generates spi_bram.bin to a converted file that is linked to spi_rom.bin (ARM9 only)
netos/src/bsp/bootloader/romImage/Makefile	Generates spi_rom.bin under netos/src/bsp/platforms/my_platform/ (ARM9 only)

Example: using the Makefile

This example shows how to build the NET+OS libraries and BSP for your platform:

- 1 Open a bash shell and change to this directory:

```
cd c:/netos
```

- 2 Enter these commands:

```
export PLATFORM=my_platform export
```

```
MODE=verbose
```

```
make
```

where you replace `my_platform` with the name of your platform.

For example, if you are using the `connectcorewi9c_a` platform, you would replace `my_platform` with `connectcorewi9c_a`.

The system rebuilds any NET+OS libraries that have changed, including the BSP.

Application Makefile

C H A P T E R 1 2

This chapter describes the application `Makefile` hierarchy and its sections. This chapter also describes the procedures for building, cleaning, and porting an application to a new platform.

Building applications

To allow ease of use and rapid prototyping, an application Makefile hierarchy is supplied with each sample application, located under the sample application's 32b folder. (For example, the Makefile for the sample application program `naftpapp` is in the `src/naftpapp/32b` directory and is called `Makefile`.) This Makefile allows application writers to rapidly assemble complex applications.

Application Makefiles

Each application has its own Makefile in the `applications/32b` directory.

For example, the Makefile for the `naftpapp` application example is in the `src/naftpapp/32b` directory and is called `Makefile`. This Makefile includes a master Makefile, which is in the `./src/linkerscripts` directory

You do not need to modify the master Makefiles; to create a new application, copy an existing Makefile and use it as a template.

The application Makefile generates these files:

File	Description
<code>image.bin</code>	Application image. Can be downloaded into flash, decompressed by the bootloader, and executed from RAM
<code>image.elf</code>	Application image in ELF format. Contains debug symbols; you use it to debug your RAM-based application.
<code>image.map</code>	Linker map. Contains size and location information about RAM-based application symbols.
<code>image.sym</code>	Symbol table. Contains information on the location of RAM-based application symbols.
<code>rom.bin</code>	ROM-based application image. Can be downloaded into and executed from flash

File	Description
rom.elf	ROM-based application image in ELF format. Contains debug symbols you use to debug ROM-based application
rom.map	Linker map. Contains information (size and location) about ROM-based application symbols
rom.sym	Symbol table. Contains information on the location of ROM-based application symbols

These variables are defined in the `Makefile` application:

Variable	Description
APP_DESCRIPTION	A string that contains a brief description of this application.
APP_INCDIRS	List of include paths with <code>-I</code> prefix
APP_C_FILES	List of C
APP_CC_FILES	List of C++ files
APP_ASM_FILES	List of assembly files
APP_C_FLAGS	List of C flags to pass to the compiler; for example, you can turn on debug information (<code>-g</code>) or optimization here
APP_C_FLAGS	List of C flags to pass to the compiler for C++ files
APP_ASM_FLAGS	List of C flags (such as <code>-Werror</code> , <code>-Os</code>), to pass to the assembler
APP_LIBS	List of libraries to link into this application

Definitions of the Makefile

The application `Makefile` includes these definitions:

Section	Description
NETOS_DIR	Defines the root of the NET+OS tree (where this version of NET+OS is installed).
APP_DESCRIPTION	Description of the application used to identify the image in the standard output window.
vpath	Defines the search path for all the application source files.

Section	Description
BUILD_RAM_IMAGE	<p>When enabled (non-zero), builds the RAM-based and debug images, <code>image.bin</code> and <code>image.elf</code>, respectively. The RAM-based <code>image.bin</code> image can be loaded into flash, along with the <code>bootloader rom.bin</code>, and decompressed into high speed SDRAM at startup, for faster execution and higher performance. Because the image must be decompressed, the time to boot is longer. If your application has a critical boot time requirement, you might want to use the application ROM image (or modify the decompression scheme in the bootloader.</p> <p><code>BUILD_RAM_IMAGE</code> option is enabled by default.</p>
BUILD_ROM_IMAGE	<p>When enabled (non-zero), builds the ROM-based image <code>rom.bin</code>, which can be loaded into flash in place of the <code>bootloader rom.bin</code>, and includes the bootstrap process and the application image. This image is executed from ROM (instead of RAM), which eliminates the need to decompress the application image from ROM to RAM and reduces the boot time.</p> <p>Because of the slower speeds of ROM memory access (and lack of memory burst capability), program execution speed overall is reduced. <code>BUILD_ROM_IMAGE</code> option is enabled by default.</p> <p>Be aware that applications you build to execute from flash cannot:</p> <ul style="list-style-type: none"> ■ Use flash-based file systems. ■ Store configuration parameters in flash. ■ Update firmware in flash.
APP_C_FILES	<p>List of application ANSI C files to be compiled using the ARM GNU GCC compiler.</p> <p>These files should have a <code>.c</code> file extension.</p>
APP_CC_FILES	<p>List of application C++ files to be compiled using the ARM GCC compiler. For proper C++ compilation, include the <code>-xc++</code> flag in the <code>APP_CC_FLAGS</code> group.</p> <p>These files should have a <code>.C</code>, <code>.cxx</code>, or <code>.cc</code> file extension.</p>
APP_ASM_FILES	<p>List of application assembler files to be assembled using the ARM GCC assembler.</p> <p>These files should have a <code>.s</code>, <code>.S</code>, or <code>.arm</code> file extension.</p>
APP_C_FLAGS	<p>Compiler directives and defines passed through to the GCC compiler and used during compilation of the <code>APP_C_FILES</code> application files. This must be a recognizable GCC option; for example, adding <code>-DAPP_DEBUG</code> defines the <code>APP_DEBUG</code> symbol in your application.</p>
APP_CC_FLAGS	<p>Compiler directives and defines passed through to the GCC compiler and used during compilation of the <code>APP_CC_FILES</code> application files.</p>

Section	Description
APP_ASM_FLAGS	Assembler directives and defines passed through to the GCC assembler and used during assembly of the APP_ASM_FILES application files.
APP_INCDIRS	Application-specific include directories, in GCC compatible format. (Uses the -I prefix and is passed through directly to GCC.) For example, to add the directory above the 32b directory, set this symbol to -I .
APP_LIBS	Libraries required for this specific application. Requires full path and file name.

Makefile hierarchy

This section describes the hierarchy that the Makefile uses.

Path name	Description
linkerScripts/Makefile.app.inc	This Makefile is included directly in the application Makefile and includes the linkerScripts/Makefile.appcc.common and the linkerScripts/Makefile.appbuild Makefile based on the platform selected. Sets the BUILD_METHOD based on platform, which is then used to pull from the appropriate linkerscripts directory.
linkerScripts/Makefile.appcc.common	This Makefile is included in the linkerScripts/Makefile.app.inc and assigns values to the GNU toolset, DEBUG_FLAG, WARN_FLAGS, C_DEFS, CC_DEFS, ASM_DEFS, C_FLAGS, CC_FLAGS, and ASM_FLAGS. These settings are platform-independent.
Makefile.appbuild.ns9360	This Makefile is included in linkerScripts/Makefile.app.inc. Includes all the NS9360 specific build and .gdbinit settings for the make.
Makefile.appbuild.ns9750	This Makefile is included in linkerScripts/Makefile.app.inc. Includes all the NS9750 specific build and .gdbinit settings for the make.
Makefile.appbuild.original	This Makefile is included in linkerScripts/Makefile.app.inc when any other processor type is used.
Makefile.appbuild.ns9215	This Makefile is included in linkerScripts/Makefile.app.inc. Includes all the NS9210 or NS9215 specific build and .gdbinit settings for the make.

Makefile targets

In addition to the targets for the files listed earlier in this appendix (for example, either `image.bin` or `rom.bin`), these targets exist in the Makefile:

Target	Description
<code>clean</code>	Removes all objects, images, map files, symbol files, and dependencies.
<code>all</code>	Builds all objects, images, map files, symbol files, and creates a dependency file.
<code>gdbinit</code>	Creates a <code>.gdbinit</code> file used for the gdb debugger.

Building an application

NET+OS ships with prebuilt libraries. If you modify the BSP or a library, you must rebuild the libraries before you build your application.

For example, to build the `naftpapp` application, you would use this procedure:

To build your application:

- 1 Change to this directory:

```
cd netos/source/examples/naftpapp/32b
```
- 2 Do one:
 - *Either* enter this command:

```
make PLATFORM=my_platform DEBUGGER=jtaglink
```
 - *Or* enter these commands:

```
export PLATFORM=my_platform
export DEBUGGER=jtaglink
make all
```

Creating `.gdbinit` files for your debugger

NET+OS creates a `.gdbinit` file as part of the application make process. This is done by building the `all` target.

The supported debugger options are `jtaglink`, `raven`, and `majic`.

Cleaning an application

When you clean an application, temporary files such as objects and dependency files are deleted.

To clean an application, enter this command:

```
make PLATFORM=my_platform clean
```

Porting an application to a new platform

Some applications cannot run on all platforms. Makefiles for these applications check the value of the `PLATFORM` variable and will terminate with an error message if you try to build the application on an unsupported platform.

If you create a new platform, you must update the application Makefiles to make the application build on the new platform. To update an application's Makefile to support a new platform, look for statements in the Makefile similar to these:

```
ifneq $(findstring $(PLATFORM), ns9750_a ns9360_a_eng),)
```

```
else
```

```
$(error This application is not supported for the specified platform, check readme for more information)
```

```
endif
```

The supported platforms are listed after `ifneq $(findstring $(PLATFORM);` in this case, the platforms are `ns9750_a ns9360_a_eng`. Edit the Makefile, and add the name of your platform to the list.

If a Makefile does not have these statements, the application is supported on all platforms.

Part 4: Building Web Pages

Using the Advanced Web Server Utility

C H A P T E R 1 3

This chapter describes the Advanced Web Server (AWS) utility, which you use to convert HTML Web pages into usable, compilable C source code.

Overview

The Advanced Web Server (AWS) utility, also known as PBuilder, is a program that converts HTML Web pages into usable, compilable C source code.

The utility uses several inputs — in particular, Web content — to generate source files to be used and linked with the Advanced Web Server. You can maintain or update an HTML page, rerun the PBuilder utility, and recompile the application program to generate updated images.

Working in the way, you can directly edit the web page and debug edits with a standard web browser, rather than update source code generated from a tool.

The PBuilder utility

The PBuilder utility, a component of AWS, converts HTML Web pages into usable, compilable C source code. The HTML pages are stored as linked lists of smaller data structures that AWS requires.

Digi strongly discourages generating these structures manually. The structures are complex, and their internal structure is beyond the scope of this guide.

PBuilder understands *comment tags* (described in the next section), which are special proprietary annotations. The comment tags are within HTML comment syntax, so they have no effect on the Web page, and they are absent when the page is served by the AWS. However, comment tags allow you to generate and modify hooks (function stubs) with the present dynamic content inserted.

Comment tags

The most important component of the PBuilder utility is the comment tags you insert into the HTML Web pages. You can use comment tags to link dynamic data fields with the Web page to specific application variables or functions.

Each comment tag begins with `<!-- tagname -->` and ends with

```
<!-- RpEnd -->
```

The Web content within a comment tag (that is, the HTML between `<!-- RpFormInput -->` and `<!-- RpEnd -->`) is not used, nor is it required. Digi recommends that you include the HTML, however, to assist when you create Web pages.

The Advanced Web Server Toolkit documentation for the PBuilder utility describes comment tags in detail. Digi strongly recommends that you carefully review the `nahttp_pd` application and read the comment tag section in the PBuilder documentation.

About the Advanced Web Server Toolkit documentation

The Advanced Web Server Toolkit documentation, included on the NET+OS CD, describes how to annotate HTML Web content with comment tags to pass dynamic content through the server. The documentation also provides examples.

A portion of the documentation describes the internal workings of the AWS. These structures and routines are considered private and can be changed at any time. A section also is included that describes the PBuilder utility and how the phrase dictionary is used for Web content compression.

Digi has created 8 white papers describing the use of different aspects of the AWS. Digi recommends that you also review these pieces of documentation.

Running the PBuilder utility

You run the PBuilder utility from a DOS prompt by entering this command:

```
pbuilder list.bat
```

A window that looks similar to this opens:


```

C:\WINNT\System32\cmd.exe
C:\nahttp_ph\pbuilder>dir
Volume in drive C has no label.
Volume Serial Number is AC15-F5B2

Directory of C:\nahttp_ph\pbuilder

08/10/00 10:24a    <DIR>      -
08/10/00 10:24a    <DIR>      ..
08/10/00 10:24a    <DIR>      html
08/02/00 10:52a             81 list.bat
08/02/00 09:36a             4,155 netarm1_v.c
02/08/00 05:43p             1,650 PbSetUp.txt
08/04/00 08:24a             5,302 RpUsrDct.txt
               ? File(s)             11,188 bytes
               1,645,022,208 bytes free

C:\nahttp_ph\pbuilder>dir html
Volume in drive C has no label.
Volume Serial Number is AC15-F5B2

Directory of C:\nahttp_ph\pbuilder\html

08/10/00 10:24a    <DIR>      -
08/10/00 10:24a    <DIR>      ..
08/02/00 10:53a             504 formreply.htm
08/02/00 12:55p             340 netarm1.htm
08/04/00 07:31a             2,989 netarm2.htm
08/02/00 09:19a             7,262 netsilicon.gif
               6 File(s)             11,095 bytes
               1,645,022,208 bytes free

C:\nahttp_ph\pbuilder>type list.bat
html/netarm1.htm
html/netarm2.htm
html/netsilicon.gif
html/formreply.htm

C:\nahttp_ph\pbuilder>pbuilder list.bat
PageBuilder version 3.06b2
Setting CreateSingleSourceFile flag
Replacing ExplicitVoidPointers -- old = "", new = "<void *> "
Setting UseFileNameForUrl flag
Converting html\netarm1.htm
Converting html\netarm2.htm
Converting html\netsilicon.gif
Converting html\formreply.htm
C:\nahttp_ph\pbuilder>

```

Directory list

pbuilder list.bat command

The window shows a directory list followed by a PBuilder execution and the contents of list.bat. The list.bat file contains all the Web pages used for the nahttp_pd application. The Web page file (that is, list.bat) needs either a .bat OR .txt extension.

The Web pages within the files are located in the \html directory. The list.bat file, however, requires the Web pages to be listed with a forward slash; for example, html/netarm1.htm.

You need these additional files to run the PBuilder utility:

- PbSetUp.txt — Copy this file from the nahttp_pd application directory, and use it to configure the PBuilder utility.
Do not change this file.
- RpUsrDct.txt — Contains definitions for Web content compression and is used to generate the RpUsrDct.c and RpUsrDct.h files.
You can update the RpUsrDct.txt file to include common phrases used in the application's Web pages.

The output of this PBuilder execution — netarm1.c and netarm1_v.c — is located in the \html directory and is the source code representation of the Web pages:

- The netarm1.c file contains the linked list structures.
Never update or modify this file.
- The html\netarm1_v.c file contains the stubs used for dynamic content. This file was copied to the working directory (.\\) and fleshed out for this application.

It is good practice to move the v.c file to a different directory. Otherwise, when you run the PBuilder utility again, the fleshed-out version of the file will be overwritten.

This PBuilder execution also produces the RpPages.c file, which contains a structure - gRpMasterObjectList - that contains all the application Web pages.

You must compile and link these files for this application:

- pbuilder\html\netarm1.c
- pbuilder\netarm1_v.c
- pbuilder\RpPages.c
- pbuilder\RpUsrDct.c

Because pbuilder\RpUsrDct.h is required, you need to add the \pbuilder\ path to the build's include path.

Linking the application with the PBuilder output files

When you build an application, you include the AWS library in the final link of the application. You also need to compile and include three additional files in the build:

- `security.c`
- `file.c`
- `cgi.c`

These files are in the appropriate application directory. You can either leave the files as they are or update them based on Web application requirements.

For examples of overwriting the files, see the `nahttp_pd` or `nafcgi` sample application on the NET+OS CD.

security.c file

Using the `security.c` file, you can add up to eight security realms. You can then use the realms to password-protect Web pages.

For more information, see the `nahttp_pd` sample application or the Advanced Web Server Toolkit documentation for the PBuilder utility.

cgi.c and file.c files

You use the `cgi.c` and `file.c` files to handle external CGI and to add or simulate a file system. The file system method was used for uploading and retrieving the file used in the `nafcgi` sample application.

For more information about using external CGI, see the `nafcgi` sample application, the *NET+OS API Reference*, or the Advanced Web Server Toolkit documentation for the PBuilder utility.

Creating Web pages

The Management API Interface to the Advanced Web Server (MAW) API integrates the Advanced Web Server and the Management API. You use the MAW API to construct Web pages that access management variables.

The Advanced Web Server has a built-in way to support a custom interface to system variables. The interface has been adapted to access variables through the management API, allowing you to use the standard AWS mechanism for embedding dynamic data into Web pages. This program demonstrates how to create Web pages that display and change management variables.

AWS custom variables

Using AWS, you can create Web pages that can display the current value of variables and prompt users for new values. To create these pages, you insert comment tags in your HTML source code. These tags identify variables to AWS and tell it how to access them.

For example, these HTML comment tags, when run through the PBuilder utility, generate C source code that AWS uses to display the variable *Username*:

```
<!-- RpNamedDisplayText Name=Username RpTextType=ASCII RpGetType=Function
RpGetPtr=GetUsername -->
<!-- RpEnd -->
```

The HTML comment tag starts with the `RpNamedDisplayText` keyword, which identifies the comment tag as an AWS command to insert the current value of a variable into a Web page.

This table describes the tags:

This tag	Tells AWS that
<code>Name=Username</code>	The variable is named <i>Username</i> .
<code>RpTextType=ASCII</code>	The variable is an ASCII string.
<code>RpGetType=Function</code>	A function has been supplied to read the variable.
<code>RpGetPtr=GetUsername</code>	The function is named <code>GetUsername</code> .

When PBuilder encounters this comment tag, it converts the HTML to C code and includes a call to the `getUsername` function, which returns an ASCII string that AWS inserts into the Web page. For more information about using comment tags, see the Advanced Web Server Toolkit documentation for the PBuilder utility.

AWS normally accesses variables directly through either pointers or functions that you write. In addition, AWS has a built-in mechanism to support customized access to variables.

Comment tags that use the custom interface for accessing variables are similar, with these exceptions:

- You must set the `RpGetType` and `RpSetType` tags to `Custom`.
- The `RpGetPtr` and `RpSetPtr` tags are no longer needed. Setting the type tag to `Custom` tells AWS to call a customizable routine to get the value of the variable.

Through modifications to the AWS's customizable routines to access management variables, AWS and the management API have been integrated. So, for example, if the variable `Username` were registered with the management API, the AWS comment tag to display its value would be:

```
<!-- RpNamedDisplayText Name=Username RpTextType=ASCII
      RpGetType=Custom -->
<!-- RpEnd -->
```

Data types

This table shows how AWS data types are mapped to management API data types:

AWS type	Management type
ASCII	MAN_CHAR
ASCIIExtended	MAN_CHAR
ASCIIFixed	MAN_CHAR
HEX	WORD8
HEXColonForm	WORD8
DotForm	WORD8
Signed8	INT8

AWS type	Management type
Signed16	INT16
Signed32	INT32
Unsigned8	WORD8
Unsigned16	WORD16
Unsigned32	WORD32

Displaying variables

To display the values of management variables in Web pages, use the AWS `RpNamedDisplayText` comment tag. The comment tag takes this form:

```
<!-- RpNamedDisplayText Name=name RpTextType=type RpGetType=Custom-->
<!-- RpEnd -->
```

where you replace:

- *name* with the name of the management variable to display.
- *type* with the AWS type of the variable.

For example, assume that `monthString` is a character string, `yearInt32` is a 32-bit integer, and `dayWord8` is an 8-bit word, and that all the variables have been registered with the management API. The HTML code to display them would be:

The date is

```
<!-- RpNamedDisplayText Name=monthString RpTextType=ASCII RpGetType=Custom -->
<!-- RpEnd -->
<!-- RpNamedDisplayText Name=dayWord8 RpTextType=Unsigned8 RpGetType=Custom -->
<!-- RpEnd -->
<!-- RpNamedDisplayText Name=yearInt32 RpTextType=Signed32 RpGetType=Custom -->
<!-- RpEnd -->
```

Changing variables

You use HTML forms to prompt users for input. AWS comment tags are embedded in the HTML form commands to tell AWS how to transfer the user's input into application variables.

RpFormInput

To prompt users for a numeric value or a string, use the `RpFormInput` tag. The format of this tag is:

```
<!-- RpFormInput TYPE=promptType RpTextType=dataType NAME=name RpGetType=Custom  
RpSetType=Custom MaxLength=length Size=size -->
```

html code

```
<!-- RpEnd -->
```

where you replace:

- *promptType* with the type of prompt for this input field (text, password, hidden, check box, or option button).
- *dataType* with the AWS data type for the variable.
- *name* with the name the variable was registered under.
- *length* with the maximum length for the variable.
- *size* with the size of the input field.

For example, this HTML code prompts for a value for `maxTemperature`, which is a 16-bit integer. The prompt is a 15-character-wide text field.

```
<!-- RpFormInput TYPE=text RpTextType=Signed16 NAME=maxTemperature RpGetType=Custom  
RpSetType=Custom MaxLength=15 Size=15 -->  
<!-- RpEnd -->
```

RpFormTextAreaBuf

Use `RpFormTextAreaBuf` to prompt users for a string value with a multi-line text box. Use this form:

```
<!-- RpFormTextAreaBuf NAME=name RpGetType=Custom RpSetType=Custom ROWS=height  
COLS=width -->
```

```
<!-- RpEnd -->
```

where you replace:

- *name* with the variable's name
- *height* with the height of the text box
- *width* with the width of the text box

For example, this HTML code prompts users with a 4 x 50 text box to enter a new value for the string `postalAddress`:

```
<!-- RpFormTextAreaBuf NAME=postalAddress RpGetType=Custom RpSetType=Custom ROWS=4
COLS=50 -->
<!-- RpEnd -->
```

RpFormSingleSelect and RpSingleSelectOption

You can use a select list to prompt for a numeric value to be written into an 8-bit word using the `RpFormSingleSelect` and `RpSingleSelectOption` tags:

- `RpFormSingleSelect` sets up a select list.
- `RpSingleSelectOption` sets up individual items in the select list.

RpFormSingleSelect

The `RpFormSingleSelect` tag has this form:

```
<!-- RpFormSingleSelect NAME=name RpGetType=Custom RpSetType=Custom Size=size -->
option list
<!-- RpEnd -->
```

where you replace:

- *name* with the variable's name
- *size* with the number of visible lines in the select list
- *option list* with a list of `RpSingleSelectOption` tags

RpSingleSelectOption

The `RpSingleSelectOption` tag has this form:

```
<!-- RpSingleSelectOption value="text label"
RpItemNumber=numericValue -->
<!-- RpEnd -->
```

where:

- *text label* is a label for this option.
- *numericValue* is the corresponding numeric value to be assigned to the variable if the user selects this option.

The next example sets up a select list that prompts users to choose a day of the week. The variable *dayOfTheWeek* is set to a value between 0 and 6, depending on which day a user chooses.

```
<!-- RpFormSingleSelect NAME=dayOfTheWeek RpGetType=Custom RpSetType=Custom Size=7
-->
<!-- RpSingleSelectOption value="Sunday" RpItemNumber=0 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Monday" RpItemNumber=1 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Tuesday" RpItemNumber=2 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Wednesday" RpItemNumber=3 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Thursday" RpItemNumber=4 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Friday" RpItemNumber=5 -->
<!-- RpEnd -->
<!-- RpSingleSelectOption value="Saturday" RpItemNumber=6 -->
<!-- RpEnd -->
<!-- RpEnd -->
```

Security

AWS allows you to associate a username and password with a group of Web pages. The combination of the username, password, and list of Web pages is called a *realm*. The AWS requires users to supply a username and password whenever they access any page in the realm. You can create up to eight realms.

Exceptional cases

You may need to write special-purpose code to access management variables. In these cases, you can specify the AWS function type in the comment tags, and then supply functions to perform the access.

In this example, the `appGetDate` and `appSetDate` functions are defined to access a management variable:

```
<!-- RpFormInput TYPE=text NAME=dateString RpGetType=Function RpGetPtr=appGetDate
RpSetType=Function RpSetPtr=appSetDate MaxLength="31" Size="31" -->
<!-- RpEnd -->
```

Controlling the MAW module

You can configure the MAW module to control:

- The timeout that is used to access management variables
- The array subscripts that are used when accessing management variables that are arrays
- How error conditions are handled

This table shows the default configuration settings:

Setting	Default action
Semaphore timeout	Wait forever for semaphores to unlock.
Array subscripts	If the variable is a one-dimensional character array, read or write the entire variable.
Error handling	Halt system on errors.

Setting the semaphore timeout

Management variables can be protected by one or more semaphores. When the MAW module accesses a management variable, it specifies the maximum amount of time it will wait for the semaphores to unlock. By default, the timeout is infinity.

Applications change the timeout value with the `mawSetAccessTimeout` function, which is defined as:

```
void mawSetAccessTimeout (MAN_TIMEOUT_TYPE timeout);
```

The `timeout` argument specifies the new timeout value.

Applications also can specify different timeouts for each variable. To do so, use `mawInstallTimeoutFunction` to register a function that is passed the name of each variable being accessed and returns the appropriate timeout. This function is defined as:

```
void mawInstallTimeoutFunction (mawTimeoutFn appFunction);
```

The *appFunction* argument is a pointer to a function supplied by the application that controls the timeouts used for each variable.

mawTimeoutFn type

The type `mawTimeoutFn` is defined as:

```
typedef MAN_TIMEOUT_TYPE (*mawTimeoutFn)(char *varName);
```

The *varName* argument specifies the name of the variable being accessed, and the function returns an appropriate timeout value.

Array subscripts

If Web pages access management variables that are arrays, the application must register a function to specify the subscripts to use when the arrays are accessed. You do this by calling the function `mawInstallSubscriptsFunction`, which is defined in this way:

```
void mawInstallSubscriptsFunction (mawSubscriptsFn appFunction);
```

The *appFunction* argument is a pointer to the application-supplied function that determines the subscripts to use.

mawSubscriptsFn type

The `mawSubscriptsFn` type is defined in this way:

```
typedef int * (*mawSubscriptsFn)(char *varName, INT16 *indices, int *dimensions, int numberdimensions,  
    AwsDataType htmlType);
```

where:

- *varName* is a pointer to the variable being accessed.
- *indices* is a pointer to an array of integers that are the current loop indices being used by the HTTP server.
- *dimensions* is a pointer to an array of integers that specify the dimensions of the management variable.

- *numberDimensions* is the number of dimensions the management variable has.
- *htmlType* is the data type the HTTP server is expecting.

The function must return a pointer to an integer array that contains the subscripts of the array element to be accessed.

If the variable is an array of characters, the function can return `NULL` to indicate that the entire array is to be read or written.

Error handling

Applications can use the `mawInstallErrorHandler` function to install an error handler. This function is defined as:

```
void mawInstallErrorHandler (mawErrorFn appFunction);
```

The *appFunction* argument is a pointer to the application's error handler.

mawErrorFn type

The `mawErrorFn` type is defined in this way:

```
typedef void * (*mawErrorFn)(char *varName, AwsDataType htmlType, MAW_ERROR_TYPE error);
```

where:

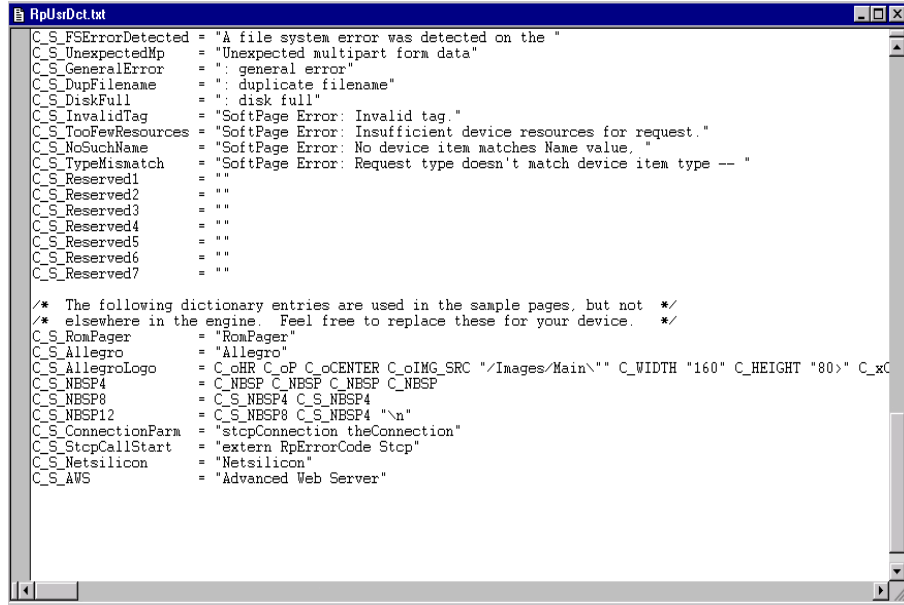
- *Varname* is a pointer to the variable being accessed.
- *HtmlType* is the data type expected by the HTTP server.
- *Error* is the error condition that is identified.

The function either halts the system or returns a value the HTTP server can use.

Phrase dictionaries and compression

The AWS uses a phrase dictionary technique to provide compression for static ASCII text strings with the HTML Web content. The PBuilder utility uses the `RpUsrDct.txt` file as input and builds its data structures to point to common phrases in the dictionary instead of repeating strings.

This figure shows the content of the `RpUsrDct.txt` file for the `nahttp_pd` application:



```

C_S_FSErrorDetected = "A file system error was detected on the "
C_S_UnexpectedMp     = "Unexpected multipart form data"
C_S_GeneralError     = ": general error"
C_S_DupFilename      = ": duplicate filename"
C_S_DiskFull         = ": disk full"
C_S_InvalidTag       = "SoftPage Error: Invalid tag."
C_S_TooFewResources  = "SoftPage Error: Insufficient device resources for request."
C_S_NoSuchName       = "SoftPage Error: No device item matches Name value, "
C_S_TypeMismatch     = "SoftPage Error: Request type doesn't match device item type -- "
C_S_Reserved1        = ""
C_S_Reserved2        = ""
C_S_Reserved3        = ""
C_S_Reserved4        = ""
C_S_Reserved5        = ""
C_S_Reserved6        = ""
C_S_Reserved7        = ""

/* The following dictionary entries are used in the sample pages, but not */
/* elsewhere in the engine. Feel free to replace these for your device. */
C_S_RonPager         = "RonPager"
C_S_Allegro          = "Allegro"
C_S_oHR C_oP C_oC C_oCENTER C_oIMG_SRC "/Images/Main/" C_WIDTH "160" C_HEIGHT "80" C_xC
C_S_NBSP4            = C_NBSP C_NBSP C_NBSP C_NBSP
C_S_NBSP8            = C_S_NBSP4 C_S_NBSP4
C_S_NBSP12           = C_S_NBSP8 C_S_NBSP4 "\n"
C_S_ConnectionPara   = "stopConnection theConnection"
C_S_StopCallStart    = "extern RpErrorCode Stcp"
C_S_Netsilicon       = "Netsilicon"
C_S_AWS              = "Advanced Web Server"
    
```

You add common phrases in all the application Web pages (for example, a company name that is used several times).

In the sample file, note this definition, which is used several times in the application Web pages:

`C_S_AWS` = "Advanced Web Server"

Search the `\pbuilder\html\netarm1.c` file. The `C_S_AWS` string is used consistently throughout the file.

Maintaining and modifying Web content

After you generate application source files, the best way to maintain and update Web content is through the HTML pages. Digi recommends that you maintain these files and include them in source control.

If a Web page requires a change or a new page, you can either update the HTML, add a new page to the `list.bat` file, or do both. You can add new phrases to the dictionary at any time.

For the changes to take effect, rerun the PBuilder utility. The application or image is automatically rebuilt.

Sample applications

Two sample applications are included in the application directory:

- `nahttp_pd` — This application shows examples of using comment tags, overwrites the `security.c` file to use a password-protected page, and shows an example of the phrase dictionary.
- `nafilecgi` — This application shows how a file can be uploaded and served, and it overwrites the `cgi.c` and `file.c` files to external CGI.

Part 5: Miscellaneous

Porting NET+OS v6.x Applications to NET+OS v7.x

C H A P T E R 1 4

This chapter describes the differences between the APIs in NET+OS 6.x and NET+OS 7.x.

Changes to the flash driver

The NET+OS flash driver has been changed to make it possible to select which parts are supported by the driver. This change reduces the memory requirements because table entries for unused parts not compiled into the driver.

The `flashparts.h` file in the `platform` directory determines which parts are supported by each platform. The file contains a set of macro definitions, and each definition corresponds to a specific flash part. For example, `NAFLASH_WANT_TO_SUPPORT_AM29DL323DTB` determines whether the AMD AM29DL323D part is supported.

- To build in support for a part, set the macro definition to `TRUE`.
- To drop support for a part, set the macro definition to `FALSE`.

When you port a platform to NET+OS 7.x that you created under NET+OS 6.x, you must copy the `flashparts.h` file into the `platform` directory and edit it to enable support for the flash parts your hardware uses.

IAM and ACE

In NET+OS 7.x, the Internet Address Manager (IAM) replaces the Address Configuration Executive (ACE), which was used in NET+OS 6.x and previous releases of NET+OS.

Changes to the sockets API

The NET+OS sockets API has been updated to support both IP version 6 and IP version 4. Most of these changes, described next, make the API more compatible with BSD sockets:

- In previous releases, `sockaddr_in` structures were used to pass addresses to the `accept`, `bind`, `connect`, `getpeername`, `getsockname`, `recvfrom`, and `sendto` functions. NET+OS 7.x uses the `sockaddr` structure. The compiler emits a warning message if you use `sockaddr_in` instead of `sockaddr`.

- The Fast Socket API has been replaced with the Zero Copy API. You need to rewrite applications that used the Fast Socket API to use the Zero Copy API. This API is documented in the online help, in the “Zero Copy Socket API” section.
- The `NAIpSetKaGarbage` function is no longer supported.
- These IP level options are no longer supported:

<code>IP_O_FRAG</code>	<code>IP_O_RR</code>
<code>IP_O_SECURE</code>	<code>IP_O_STREAM</code>
<code>IP_O_LSRR</code>	<code>IP_O_TIME</code>
<code>IP_O_SSR</code>	<code>IP_MULTICAST_LOOP</code>

- These TCP level options are no longer supported:

<code>TCP_O_SEQNO</code>	<code>TCP_SET_RCV_MSS</code>
<code>TCP_O_SEQNO</code>	<code>TCP_ACKNSEG</code>
<code>TCP_ENABLE_PAWS</code>	<code>TCP_PERMIT_SACKS</code>
<code>TCP_ENABLE_PAWS disable</code>	<code>TCP_SEND_SACK</code>
<code>TCP_MAXSEG</code>	<code>TCP_FAST_RETR_RECOV</code>
<code>TCP_USE_PEER_MSS_OPTION</code>	<code>TCP_ENABLE_TRANSACTION_TCP</code>

- These socket level options are no longer supported:

<code>SO_THROUGHPUT</code>	<code>SO_RXDATA</code>
<code>SO_EXPEDITE</code>	<code>SO_ADD_FDEST</code>
<code>SO_NOCHKSUM</code>	<code>SO_DEL_FDEST</code>
<code>SO_MAXMSG</code>	<code>SO_FSOCK_CALLBACK</code>
<code>SO_MYADDR</code>	<code>SO_NUM_FBUFFS</code>

Changes to SNMP

The SNMP agent has been replaced. You must reimplement any MIBs that you implemented using MIBMAN and NET+OS 6.x

netos/src/bsp/customize directory

The `netos/src/bsp/customize` directory has been added to the BSP. This directory contains files that are shared by all platforms, and which you may want to customize. Changes to files in this directory affect all BSP platforms. The BSP Makefile is set up as the source path to search first for files in the `platform` directory, and then in the `netos/src/bsp/customize` directory.

To change a file in `netos/src/bsp/customize` for only one platform, copy that file into the `platform` directory, and make the changes to the copy. The copy in the `platform` directory takes priority over the copy in the `customize` directory.

Changes to Makefile variables and defines

The Makefiles set up internal variables that indicate the processor type. The Makefiles also create a symbol that identifies the processor type by defining a macro in the command line to the C compiler. These symbols have changed in NET+OS 7.x.

- In NET+OS 6.x, `PROCESSOR` was set to either `arm9` or `arm7`. Starting in NET+OS 7.x, `PROCESSOR` is set to indicate the type of processor supported:
 - `net50`
 - `ns7520`
 - `ns9750`
 - `ns9360`
 - `ns9210`
 - `ns9215`

In addition, the processor type is set to 1, and all other processor type are set to 0. For example, when you build the ConnectME platform,

which uses an ns7520, `PROCESSOR` is set to ns7520, and ns7520 is set to 1. The other processor types — ns9360, for example — are set to 0.

- In previous versions of NET+OS, `CHIP` was set to indicate the type of processor the platform supported. As of NET+OS 7.x, `CHIP` is no longer defined by the `Makefiles`. It has been replaced with `PROCESSOR`.
- Starting in NET+OS 7.x, the `CPU` symbol is defined to indicate the internal CPU type. `CPU` is set to either `arm9` or `arm7`, and the selected CPU type is set to 1. For example, when you build the ConnectME, which uses an ARM7 CPU, `CPU` is set to `arm7`, `arm7` is set to 1, and the `arm9` symbol is not defined.

Automatic RAM sizing

When you used the NS9750 or NS9360 with NET+OS 6.x, you had to set the size of the SDRAM configuration files.

NET+OS 7.x adds support for automatically determining the size of RAM on power up. This feature is supported *only* when you boot from regular flash or ROM; it is not supported when you boot from SPI flash.

To use this feature:

- 1 Set the `RAM_SIZE` constant (defined in either `customize.lx` or the RAM that will be available.
- 2 Edit `init_settings.h` in the platform directory, and set the timing parameters for the type of SDRAM you plan to use.

Note that all the SDRAM you use must use the same timing parameters. The BSP cannot automatically sense the timing.

- 3 Edit `init_settings.h` in the platform directory and set the values for `FIRSTSECTORRAM`, `SECONDSECTORRAM`, `THIRDSECTORRAM`, and `FOURTHSECTORRAM`. These values are used to set the SDRAM mode register.

The settings in `init_settings.h` are documented in the online help.

The BSP automatically detects the RAM size at powerup. If the amount of RAM is greater than the value set in `RAM_SIZE`, the extra RAM is added to the heap.

Porting pre-NETOS 7.x PPP applications

Point-to-Point (PPP) is a communications protocol that allows devices to perform network communications through a serial communications line.

NET+OS 7.x has a new TCP/IP stack that includes a new PPP layer. The sections that follow show how standard PPP functions were implemented in NET+OS 6.x and how they can be ported using NET+OS 7.x

Adding a route

Adding routes allows IP packets for different networks to be routed through the interface. These functions add a static route for the specified PPP interface.

NET+OS 6.x

```
int PPPAddRoute(unsigned long destination, unsigned long mask, unsigned long gateway, int commPort);
```

NET+OS 7.x

```
int PPPAddStaticRoute(PPPUUserInterface interfaceHandle, const struct sockaddr_storage * destIpAddrPtr, int prefixLen, const struct sockaddr_storage * gatewayIpAddrPtr, int hops);
```

Deleting a route

These functions are for deleting an added route.

NET+OS 6.x

```
int PPPDelRoute(unsigned long destination, unsigned long mask);
```

NET+OS 7.x

```
int PPPDelStaticRoute(const struct sockaddr_storage * destIpAddrPtr, int prefixLen);
```

Adding PAP user/password or adding CHAP ID and secret key pair

These functions set the authentication modes for the specified PPP interface using either the CHAP or PAP protocols. In addition, these functions set the specified PPP interface's PAP username/password and CHAP ID/secret key pairs.

NET+OS 6.x

```
int PPPDeviceSetAuth(char * devname, char * papname, char * pappassword, char * chapname, char *
chapsecret, int authentication);
```

NET+OS 7.x

```
int PPPSetOption(PPPUUserInterface interfaceHandle, int protocolLevel, int remoteLocalFlag, int optionName,
const char * optionValuePtr, int optionLength);
```

Use PPP_PAP_PROTOCOL and PPP_CHAP_PROTOCOL as *protocolLevel*.

Checking link status

This function provides a method for reporting the PPP link status. A callback feature has been added to notify your application of changes in time.

NET+OS 6.x

```
int PPPCheckLink(unsigned int commPort);
```

NET+OS 7.x

```
Int PPPLinkInterfaceToDevice(PPPUUserInterface interfaceHandle, char * deviceName, PPP_USER_FUNCS *
userFuncs);
```

Use *linklayer_callback_fn* to check the link status.

Creating the interface

This function creates and configures a PPP device on the specified port where the mode argument is used to select either a modem or a direct serial connection. The device IP parameters are configured based on the specified IP address and subnet mask.

NET+OS 6.x

```
int PPPCreateDevice(unsigned int commPort, unsigned int mode, unsigned int ipAddress, unsigned int
subnetMask)
```

NET+OS 7.x

```
int PPPAddInterface(int pppMode, PPPUserInterface * interfaceHandle);
```

```
int PPPSetOption(PPPUUserInterface interfaceHandle, int protocolLevel, int remoteLocalFlag, int optionName,
const char * optionValuePtr, int optionLength);
int PPPLinkInterfaceToDevice(PPPUUserInterface interfaceHandle, char * deviceName, PPP_USER_FUNCS *
userFuncs);
```

To configure the IP address, use the `PPPSetOptions` call.

To configure serial settings, use the `configure_device_fn` callback function.

Getting the peer assigned local address

This function gets the peer-assigned IP address of the PPP interface.

NET+OS 6.x

```
int PPPGetPeerAssignedAddress( unsigned int commPort, unsigned long * ipAddress );
```

NET+OS 7.x

```
int PPPGetLocalPeerIpAddress(PPPUUserInterface interfaceHandle, struct sockaddr_storage *
localPeerIpAddressPtr,int addressFamily, unsigned int multiHomeIndex);
```

To get the peer IP address, use:

```
Int PPPGetRemotePeerIpAddress(PPPUUserInterface interfaceHandle, struct sockaddr_storage *
remotePeerIPAddress);
```

Closing the interface

In the case of a modem, this function hangs up the phone line, resets the modem, and closes the modem connection on the specified port.

In the case of a serial, this function closes the direct serial port.

NET+OS 6.x

```
int PPPModemClose(unsigned int commPort);
```

```
int PPPSerialClose(unsigned int commPort);
```

NET+OS 7.x

```
int PPPCloseInterface(PPPUUserInterface interfaceHandle);
```

Setting authentication and compression

Use these functions to:

- Set the authentication mode options for the specified PPP interface (that is, CHAP or PAP)
- Set the specified PPP interface's PAP username/password and CHAP ID/secret key pairs.
- Enable/disable Van Jacobson (VJ) compression or IP header compression.

NET+OS 6.x

Either of these:

```
int PPPSetAuth(unsigned int commPort, char * papname, char * pappassword, char * chapname, char * chapsecret, int authentication);
```

```
int PPPSetVJ(unsigned int commPort, int require_VJ)
```

NET+OS 7.x

```
int PPPSetOption(PPPUUserInterface interfaceHandle, int protocolLevel, int remoteLocalFlag, int optionName, const char * optionValuePtr, int optionLength);
```


Initializing serial port configuration

This function opens the serial port to the specified baud rate.

In the case of a modem, this function initializes the modem layer.

NET+OS 6.x

```
int PPPSerialInit(unsigned int commPort, unsigned int mode, unsigned int baud);
```

```
int PPPModemInit(unsigned int commPort, unsigned int mode, unsigned int baud, char * init_string);
```

The serial port baud rate is set up during the process to start the direct serial or modem PPP connection. Flow control and data/stop/parity are hard coded to RTS/CTS and 8N1 respectively.

NET+OS 7.x

```
Int PPPLinkInterfaceToDevice(PPPUUserInterface interfaceHandle, char * deviceName, PPP_USER_FUNCS * userFuncs);
```

To configure the serial settings, use the `configure_device_fn` callback function, which is called from `PPPOpenInterface`.

You can set flow control and data/stop/parity to whatever you want.

Setting the ring count

This function sets the ring count before the modem answers an incoming call.

NET+OS 6.x

```
int PPPSetModemAutoAnswer(unsigned int commPort, unsigned int auto_answer_rings);
```

NET+OS 7.x

```
int PPPUseDialer(PPPUUserInterface interfaceHandle);
```

```
int PPPDialerAddSendExpect(PPPUUserInterface interfaceHandle, char * sendString, char * expectString, char * errorString, int numRetries, int timeout, unsigned char flags)
```

Dial string settings***NET+OS 6.x***

Either of these:

```
int PPPSetModemDialStr(unsigned int commPort, char * dial_string);
```

```
int PPPSetModemDialString(char * dial_string);
```

NET+OS 7.x

```
int PPPUseDialer(PPPUUserInterface interfaceHandle);
```

```
int PPPDialerAddSendExpect(PPPUUserInterface interfaceHandle, char * sendString, char * expectString, char * errorString, int numRetries, int timeout, unsigned char flags);
```

Processor Modes and Exceptions

C H A P T E R 1 5

This chapter describes the modes NET+OS supports and how NET+OS handles interrupts.

Overview

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

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)	<ul style="list-style-type: none">■ 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)	<ul style="list-style-type: none">■ 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.

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
- Software 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 ARM9-based processors (such as the NS9360 and NS9750) allow external interrupts to trigger a fast interrupt, ARM7-based processors do not. Applications for both ARM7- and ARM9-based processors always 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 “ARM7 FIQ handlers” or “ARM9 FIQ handlers,” later in this chapter.

IRQ handler

An *interrupt request* is generated when one or more devices assert their interrupt signal. For ARM9-based processors, the BSP provides an *IRQ handler*, which 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, described next:

- 26 interrupt signals support AHB devices that are internal to the NS9750 and NS9360.
- Four interrupt signals support Bbus devices that are internal to the NS9750. In the NS9360, several of the BBus signals are moved up to the AHB interrupt vector table, including USB device, USB host, BBUS DMA and I2C. These changes speed up the interrupt response from those peripherals.

Several timer interrupts that are supported in the AHB interrupt vector table in the NS9750 have been combined in the NS9360 to make room for the BBus interrupts described in the previous paragraph.

- Four interrupt signals support external devices.
- One interrupt signal is not used and is considered reserved.
- For the NS9210 and NS9215 processors, the interrupt system is a two-tier priority scheme, where two lines access the CPU core and can interrupt the processor: IRQ (normal interrupt) and FIQ (fast interrupt). FIQ has a higher priority than IRQ. The IRQ interrupts come from several different sources in the processor and are managed using the Interrupt Config registers. IRQ interrupts can be enabled or disabled on a per-level basis using the Interrupt Enable registers. These registers serve as masks for the different interrupt levels.

ARM7-based processors have two interrupt signals. For more information, see the `bsp.c` file and the hardware reference for the processor you are using.

Application software can selectively Install, uninstall, enable, or disable any of the interrupt signals with `naIsrInstall`, `naIsrUninstall`, `naInterruptEnable`, and `naInterruptDisable`, respectively.

In ARM9-based processors, 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 in ARM9 based NET+ARM processor

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 `NAIrqHandler` in the `NA_isr.c` file, which reads the ISRADDR register to determine which device interrupt to process.
- 8 `NAIrqHandler` saves the current interrupt mask word and enables interrupts from higher priority devices.
- 9 `NAIrqHandler` calls the ISR that was registered for the device with the `naIsrInstall` routine.
- 10 The ISR services the device and acknowledges the interrupt.
- 11 Control returns to `NAIrqHandler`, which restores the interrupt mask word and returns.

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

Servicing Bbus interrupts in ARM9 based NET+ARM processor

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 `NAIrqHandler` in `mc_isr.c` calls `BBUS_IrqHandler`, which is installed as an ISR, to service the BBUS interrupt.
- 3 In a loop, `Bbus_IrqHandler` masks all lower priority interrupts, enables interrupts, and calls the function registered during the `NAInstallIsr` 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 returned back to `NAIrqHandler`.

Changing interrupt priority

You can change the interrupt priority level by changing the order of the `NAAhbPriorityTab` and `NABbusPriorityTab` arrays in the `bsp.c` file. The tables in the next sections, “AHB interrupts in ARM9-based processors” and “Bbus interrupts in ARM9-based processors,” 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 and calls `customizeErrorHandler`.

AHB interrupts: ARM9-based processors

The priority of each interrupt in the AHB Bus is controlled by software. 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.

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

You specify the priority for each AHB source interrupt in the `NAAhbPriorityTab` array in the `bsp.c` file.

This table lists the supported interrupt sources in the AHB Bus and the associated software directives for the NS9750:

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
Timer 2	TIMER2_INTERRUPT
Timer 1	TIMER1_INTERRUPT
Timer 0	TIMER0_INTERRUPT
Reserved	AHB_PERIPH15_INTERRUPT
I2C	I2C_INTERRUPT
PCI External 3	PCI_EXTERNAL3_INTERRUPT
PCI External 2	PCI_EXTERNAL2_INTERRUPT

AHB interrupt source	Software directive
PCI External 1	PCI_EXTERNAL1_INTERRUPT
PCI External 0	PCI_EXTERNAL9_INTERRUPT
PCI Arbiter	PCI_ARBITER_INTERRUPT
PCI Bridge	PCI_BRIDGE_INTERRUPT
LCD	CD_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

This table lists the supported interrupt sources in the AHB Bus and the associated software directives for the NS9360:

AHB Interrupt source	Software directive
External 3	EXTERNAL3_INTERRUPT
External 2	EXTERNAL2_INTERRUPT
External 0	EXTERNAL0_INTERRUPT
IEEE_1284	IEEE_1284_INTERRUPT
USB_DEVICE	USB_DEVICE_INTERRUPT
USB_HOST	USB_HOST_INTERRUPT
RTC	RTC_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
BBUS_DMA	BBUS_DMA_INTERRUPT
I2C	I2C_INTERRUPT
SER3TX	SER3TX_INTERRUPT
SER3RX	SER3RX_INTERRUPT
SER2TX	SER2TX_INTERRUPT
SER2RX	SER2RX_INTERRUPT
SER1TX	SER1TX_INTERRUPT
SER1RX	SER1RX_INTERRUPT
LCD	LCD_INTERRUPT
Ethernet PHY	ETH_PHY_INTERRUPT
Ethernet Transmit	ETH_TRANSMIT_INTERRUPT
Ethernet Receive	ETH_RECEIVE_INTERRUPT
Reserved	N/A
BBUS Aggregate	ANY BBUS INTERRUPT DIRECTIVE
AHB Bus Error	AHB_BUS_ERROR_INTERRUPT
Watchdog	WATCHDOG_INTERRUPT

This table lists the supported interrupt sources and associated software directives for the NS9210/NS9215:

Interrupt source	Software directive
0	WATCHDOG_INTERRUPT
1	AHB_BUS_ERROR_INTERRUPT
2	EXTERNAL_DMA_INTERRUPT
3	CPU_WAKE_INTERRUPT
4	ETH_RECEIVE_INTERRUPT
5	ETH_TRANSMIT_INTERRUPT
6	ETH_PHY_INTERRUPT
7	UARTA_INTERRUPT
8	UARTB_INTERRUPT
9	UARTC_INTERRUPT
10	UARTD_INTERRUPT
11	SPI_INTERRUPT
12	Reserved (IOP)
13	Reserved (IOP)
14	ADC_INTERRUPT
15	EARLY_POWER_LOSS_INTERRUPT
16	I2C_INTERRUPT
17	RTC_INTERRUPT
18	TIMER0_INTERRUPT
19	TIMER1_INTERRUPT
20	TIMER2_INTERRUPT
21	TIMER3_INTERRUPT
22	TIMER4_INTERRUPT
23	TIMER5_INTERRUPT
24	TIMER6_INTERRUPT
25	TIMER7_INTERRUPT

Interrupt source	Software directive
26	TIMER8_INTERRUPT
27	EXTERNAL9_INTERRUPT
28	EXTERNAL0_INTERRUPT
29	EXTERNAL1_INTERRUPT
30	EXTERNAL2_INTERRUPT
31	EXTERNAL3_INTERRUPT

Bbus interrupts: ARM9-based processors

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 returns.
- 5 The interrupt driver checks for more pending interrupts. If any interrupts are found, their ISRs are called as well.
- 6 When all pending interrupts are 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 `NABbusPriorityTab` 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_DMA14_INTERRUPT	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 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

Bbus interrupt source	Software directive
Serial 1 transmit	SER2TX_INTERRUPT
USB	USB_INTERRUPT
Bbus DMA	BBUS_DMA_INTERRUPT

System interrupts: ARM7-based platforms

You set the priority for interrupts using the `NAInterruptPriority` table in the `bsp.c` file of its corresponding platform.

When a device signals an interrupt, these steps occur:

- 1 The hardware sets bits in the Interrupt Status register.
- 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 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 returns.
- 5 The interrupt driver checks for more pending interrupts. If any interrupts are found, their ISRs are called as well.
- 6 When all pending interrupts are processed, the NET+OS interrupt driver returns control to the application.

The next table lists the supported interrupt sources in the ARM7 based NET+ARM processor. Interrupt sources with a higher-numbered priority level can interrupt the service routines of devices with lower-numbered priority levels.

Interrupt source	Software directive
DMA1	DMA1_INT
DMA2	DMA2_INT
DMA3	DMA3_INT
DMA4	DMA4_INT
DMA5	DMA5_INT
DMA6	DMA6_INT
DMA7	DMA7_INT
DMA8	DMA8_INT
DMA9	DMA9_INT
DMA10	DMA10_INT
ENI/PORT1	ENI/PC_PORT1_INT
ENI/PORT2	ENI/PC_PORT2_INT
ENI/PORT3	ENI/PC_PORT3_INT
ENI/PORT4	ENI/PC_PORT4_INT
ENETRX	ENETRX_INT
ENETTX	ENETTX_INT
SER1RX	SER1RX_INT
SER1TX	SER1TX_INT
SER2RX	SER2RX_INT
SER2TX	SER2TX_INT
11 - 7	Reserved
WATCHDOG	WATCHDOG_INT
TIMER1	TIMER1_INT
TIMER2	TIMER2_INT
PCPC3	PCPC3_INT
PCPC2	PCPC3_INT
PCPC1	PCPC1_INT
PCPC0	PCPC0_INT

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 `NAInstallIsr`. 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 `NAUninstallIsr`. This routine disables the interrupt and uninstalls the ISR handler.

ARM9 FIQ handlers

Because a fast interrupt (FIQ) is a higher priority interrupt than an IRQ, an FIQ can interrupt an IRQ at any time. The default handler installed by the BSP treats a FIQ exception as an error and calls `customizeExceptionHandler`.

Use `naIsrSetFiq` to program an interrupt source to generate an FIQ interrupt, and then call `naIsrInstall` to install the interrupt handler for the FIQ.

For ARM9-based processors only:

- Unlike an IRQ, only one interrupt can be configured for an FIQ, and it must be the first one in the `NAAhbPriorityTab` array.
- To disable and remove a FIQ, call `NAUninstallIsr`.

ARM7 FIQ handlers

On ARM7 based-processors, you can configure the watchdog timer and the two general-purpose timers to 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 hardware reference for the processor you are using.

► To install an ARM7 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 to install and use FIQs, see `bsp/profiler/profilerAPI.c`.



Device Drivers



C H A P T E R 1 6

This chapter describes device driver functions.

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.

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
<i>name</i>	Pointer to a null-terminated string that is the device channel's name. The name must be unique for each device.
<i>channel</i>	Channel number for the device name. This number is passed to the device driver for all I/O requests.
<i>deviceEnter</i>	Pointer to the driver's first-level initialization routine for the channel. <code>DDIFirstLevelInitialization</code> calls this routine once, during initialization, when the C library initializes its I/O library. Kernel services are not available at this point.
<i>deviceInit</i>	Pointer to the driver's second-level initialization routine for the channel. <code>DDISecondLevelInitialization</code> calls this routine once, at startup, after the kernel has been loaded.
<i>deviceOpen</i>	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 <i>flags</i> field indicates whether the channel: <ul style="list-style-type: none"> ■ Was opened for read, write, or read/write mode ■ Operates in blocking or non-blocking mode
<i>deviceClose</i>	Pointer to the driver's close routine for the channel. This routine is called at the end of every session.
<i>deviceRead</i>	Pointer to the driver's read routine for the channel.
<i>deviceWrite</i>	Pointer to the driver's write routine for the channel.
<i>deviceIoctl</i>	Pointer to the driver's I/O control routine for the channel.
<i>flags</i>	Bit field that indicates which bits are valid in the <i>flags</i> 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 shows 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 function 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
<i>channel</i>	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 function for the device channel.

After the kernel has loaded, the device driver table is scanned, and the `deviceInit` functions 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
<i>channel</i>	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
<i>channel</i>	Channel number as set in the channel’s device table entry
<i>flags</i>	Bit field formed by ORing together one or more of these values: <ul style="list-style-type: none">■ O_RDONLY■ O_WRONLY■ O_RDWR■ O_NONBLOCK

For this routine’s return values, see the table in the section “Return values.”

deviceClose

Notifies the device driver that the application is closing its session. This function 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 Does one of these steps:
 - Sets the channel semaphore
 - 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
<i>channel</i>	Channel number as set in the channel’s device table entry

For this function’s return values, see the table in the section “Return values.”

deviceRead

Reads data from the device to the caller's buffer. This function 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 Sets the channel semaphore or returns `EBUSY` if the semaphore already is set.
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`.
- 5 Copies the data from the driver buffers until either all the data has been copied or the caller's buffer has been filled.
- 6 Updates *bytesRead*.
- 7 Releases the channel semaphore.
- 8 Returns a completion code.

Format

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

Arguments

Argument	Description
<i>channel</i>	Channel number as set in the channel's device table entry
<i>buffer</i>	Pointer to caller's receive buffer
<i>length</i>	Length of caller's receive buffer (number of bytes)
<i>bytesRead</i>	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 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
<i>channel</i>	Channel number as set in the channel's device table entry
<i>buffer</i>	Pointer to caller's buffer; not necessarily aligned
<i>length</i>	Length of caller's receive buffer (number of bytes)
<i>bytesWritten</i>	Pointer to <code>int</code> 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
<i>channel</i>	Channel number as set in the channel's device table entry
<i>request</i>	Commands encoded as integers
<i>arg</i>	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) functions 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 **bytesRead* and **bytesWritten* arguments, 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, which is located 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:/netos/lib/32b/gnu`

directory. All the sample applications that are provided with NET+OS 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 `Makefiles` 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:/netos/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:/netos/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:/netos/gnusrc` directory and enter:

```
make all
```

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

Note that the `crtbegin.o`, `crtend.o`, `crti.o`, and `crtm.o` files in the `C:/netos/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.

NET+OS device drivers configure and control the components of the Digi chips, such as serial, Ethernet, USB, and so on. These drivers are part of the NET+OS operating system, and depending on the defines in the `bsp_drivers.h` and `bsp_serial.h` file for your platform, are loaded on startup.

NET+OS device drivers

This table lists the device drivers that are supported as part of NET+OS:

Driver	Description	Supported platforms
Ethernet	Ethernet	All
SPI master	SPI master	All
SPI slave	SPI slave	NS9360, NS9750, NS9215, NS9210
Serial UART	Serial UART	All
NVRAM	Non- volatile RAM	All
System clock	System clock interface routines	All
Timer	Timer	All
MMU	Memory Management Unit	NS9360, NS9750, NS9215, NS9210
GPIO	General purpose I/O	All
IEEE — 1284	Parallel driver	NET+50, NS9360
I2c	Inter-IC	NS9360, NS9750, NS9215, NS9210
LCD	LCD routines	NS9360, NS9750
USB device	USB device	NS9360, NS9750
USB host	USB Host	NS9360, NS9750
PWM	Pulse Width Modulator	NS9360, NS9215, NS9210
RTC	Real Time Clock	NS9360, NS9215
PCI	PCI Bus	NS9750
AES	AES H/W Accelerator	NS9215, NS9210
A/D	Analog-to-Digital Converter	NS9215, NS9210
QUAD	Quadrature Driver	NS9215
SD/SDIO	Secure Digital Driver	NS9215
Comparator	Comparator	NS9215
Scratchpad	Scratchpad Memory	NS9215

Device driver interface

NET+OS device drivers are based on the standard Device Driver Interface (DDI) and use a layered model to implement device drivers. Within this model, all API calls are made through the DDI interface.

Some drivers (such as Timer and GPIO) do not use the DDI interface. Because they do not fit into a read/write type of model, they have a separate interface.

Part 6: Troubleshooting

Troubleshooting

C H A P T E R 1 7

This chapter describes how to diagnose errors you may encounter when you are working with NET+OS. This chapter also describes how to reserialize a module.

Diagnosing errors

These sections tell you how to diagnose two types of errors:

- Fatal errors
- Unexpected exceptions

Diagnosing a fatal error

Code in the BSP and NET+OS API libraries calls the `customizeErrorHandler` routine when a *fatal error* — one from which the software cannot recover — is encountered.

The default version of `customizeErrorHandler` blinks the LEDs on the module in a pattern that indicates the type of error that occurred.

► To determine where in the code an error occurred:

- 1 Stop the program in the debugger.
- 2 Examine the call stack.
The call stack lists each function frame on the stack. To go to any of these functions, double-click the function name in the call stack display.
- 3 To continue execution from the point where the error occurred, set the `naCustomizeErrorHandlerClearToContinue` variable to 0.

Be aware that because a fatal error has occurred, the results are unpredictable.

Diagnosing an unexpected exception

The `customizeExceptionHandler` routine is called whenever an unexpected exception occurs. This table describes the exceptions:

Exception type	Triggered when
Data abort	Software attempts to access memory that doesn't exist, or attempts to perform a misaligned address.
Prefetch abort	The processor attempts to fetch an instruction from memory that doesn't exist.

Exception type	Triggered when
Fast interrupt	The FIQ pin is toggled by hardware, or when internal devices such as the watchdog timer in the NET+ARM are programmed to generate it.
Software interrupt	The processor executes a software interrupt (SWI) instruction.
Undefined interrupt	The processor executes an undefined instruction.

The value of the `BSP_HANDLE_UNEXPECTED_EXCEPTION` constant in `bsp_sys.h` controls the default version of `customizeExceptionHandler`. (For details, see the *NET+OS API Reference*.) Usually, `customizeExceptionHandler` either resets the unit or blinks the LEDs in a pattern that indicates the type of exception that occurred. During development, you can continue execution from where the exception occurred.

► To diagnose an unexpected exception:

- 1 Put a breakpoint on `customizeExceptionHandler`.
- 2 When the breakpoint is reached, step into the routine until it sets `customizeExceptionHandlerClearToContinue` to `TRUE`.
- 3 Set `customizeExceptionHandlerClearToContinue` to 0.
- 4 Step through the routine until just before it returns.
- 5 Switch the debugger display to show assemble instructions.
- 6 Step through the code assembler instructions one at a time until the processor returns to the source of the exception.

Reserializing a module

The Digi Connect and ConnectCore modules ship with a boot ROM application programmed in flash memory. This application allows you to configure the module.

Observing the LEDs

Be aware of the LEDs whenever you power cycle the module. The LEDs provide information you can use to monitor the module's status at all times.

Assigning a MAC address to the module

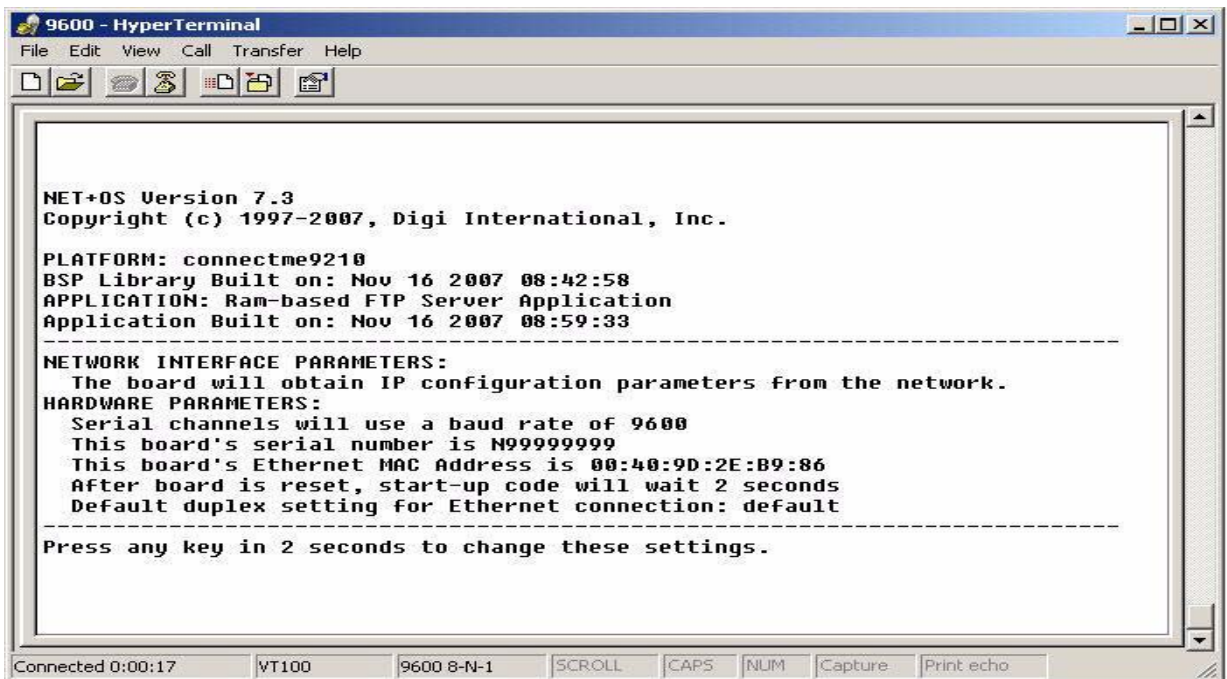
Each device on the network must have a unique Ethernet media access controller (MAC) address. The module comes preconfigured with a factory-set MAC address that is printed on a sticker on the module.

The MAC address can be lost if NVRAM is corrupted by an application under test. In such a case, you must restore the MAC address to make sure that the module can communicate over the network. The module ships with an application written in flash memory that you can use to restore the MAC address. From the debugger, you also can use any sample application built with the configuration dialog enabled.

► To restore a module's original Ethernet MAC address:

- 1 Connect the module to a serial port on your system.
- 2 Start a HyperTerminal session on the serial port.
- 3 Power up the module.

A message similar to this one appears after a brief pause:



The screenshot shows a HyperTerminal window titled "9600 - HyperTerminal". The window contains the following text:

```

NET+OS Version 7.3
Copyright (c) 1997-2007, Digi International, Inc.

PLATFORM: connectme9210
BSP Library Built on: Nov 16 2007 08:42:58
APPLICATION: Ram-based FTP Server Application
Application Built on: Nov 16 2007 08:59:33
-----
NETWORK INTERFACE PARAMETERS:
The board will obtain IP configuration parameters from the network.
HARDWARE PARAMETERS:
Serial channels will use a baud rate of 9600
This board's serial number is N99999999
This board's Ethernet MAC Address is 00:40:9D:2E:B9:86
After board is reset, start-up code will wait 2 seconds
Default duplex setting for Ethernet connection: default
-----
Press any key in 2 seconds to change these settings.
  
```

At the bottom of the window, the status bar shows: "Connected 0:00:17", "VT100", "9600 8-N-1", "SCROLL", "CAPS", "NUM", "Capture", and "Print echo".

- 4 Enter the configuration dialog by pressing a key before the timeout expires.
- 5 At the prompt, enter the system password:
password
- 6 Enter the values for the IP stack configuration settings and serial port baud rate.
- 7 At the prompt, enter the Ethernet MAC address that appears on the sticker on the module.
- 8 Respond to the prompts to set up the remaining configuration settings.

This is a sample dialog:

Enter the root password: *****

Reset configuration to default values (Y/N)? Y

For each of the following questions, you can press <Return> to select the value shown in braces, or you can enter a new value.

NETWORK INTERFACE PARAMETERS:

Should this target obtain IP settings from the network? [N] y

SECURITY PARAMETERS:

Would you like to update the Root Password? [N]

HARDWARE PARAMETERS:

Set the baud rate of Serial channels [9600]?

The new baud rate is 9600

The baud rate will be changed on next power up

Please set the baud rate for your terminal accordingly

Each development board must have a unique serial number

Set the board's serial number [N99999999]? N12345678

The board's new serial number is N12345678

Each development board must have a unique Ethernet MAC address.

Set the board's Ethernet MAC Address [00:40:9D:BA:DB:AD]?

00:40:9D:12:34:56

This board's new Ethernet MAC address is 00:40:9D:12:34:56

How long (in seconds) should CPU delay before starting up [5]?

Normally the board will automatically negotiate with the network hub (or switch) to determine the Ethernet duplex setting; however some hubs do not support autonegotiation.

What duplex setting should be used in this case (Full or Half)? [Full Duplex]

Saving the changes in NV memory...Done.

Restoring the contents of flash memory

Digi Connect and ConnectCore modules ship with a boot ROM program written in flash memory. The boot ROM program implements support for debugging and provides an FTP server that you can use to update flash memory.

You restore the original boot ROM program by using a procedure in which you:

- 1 Configure the target and JTAG debugger.
- 2 Build the BSP, making sure that you specify the correct platform.
The BSP `Makefile` automatically builds the bootloader. The bootloader image is stored in `rom.bin` in the `platform` directory.
- 3 Using the same platform, build the `naftpapp` application image in `src/examples/naftpapp`.
- 4 Run the `naftpapp` application using the JTAG debugger. Using a standard FTP client, log into the module using the `root` account.
The default password of the `root` account is `password`.
- 5 Download `rom.bin` from the BSP `platform` directory in binary mode, and then quit. Wait for the standard output message saying that the module will reset.
- 6 Rerun the `naftpapp` application using the JTAG debugger. Using the standard FTP client, download `image.bin` from the `naftpapp/32b` sample application in binary mode. Wait for the standard output that says the module will reset.
- 7 Remove the JTAG debugger and recycle power. Verify that the module boots with `naftpapp`.

The next sections provide details about each step in the procedure.

Note: Be aware that the order of the tasks for restoring the contents of flash memory is important. You *must* do the tasks in the order in which this document presents them.

Step 1: Configure the module and the debugger

- To set up the module and the **debugger**:
 - 1 Connect the JTAG debugger, Ethernet cable, and serial cable as described in the *Quick Start Guide*.
 - 2 Disable flash on the module.
 - 3 Power up the target.
 - 4 Start a HyperTerminal session.

Step 2: Building the bootloader

- To build the bootloader if rom.bin does not already exist in the BSP platform directory:
 - 1 Edit the bsp_bldr.h, bsp_drivers.h, bsp_net.h, bsp_serial.h and bsp_sys.h files and make sure the configuration settings are correct.
 - 2 Using the Makefile in the c:/netos/src/bsp directory, build the BSP.
 - 3 To select your platform, use the command-line option and enter:
`make PLATFORM = platform-name`

Step 3: Building the application image and starting naftpapp

- To build the application image and start the naftpapp application:
 - 1 Change to the naftpapp application directory, c:/netos/src/examples/naftpapp.
 - 2 Edit the appconf.h file for naftpapp, and make sure the application is configured to generate a configuration dialog.
 To generate a dialog, set the constant BSP_DIALOG_PORT in the platform bsp_sys.h.
 (For details about BSP_DIALOG_PORT in bsp_sys.h, see the *NET+OS API Reference*.)

- 3 Rebuild the `naftpapp` application.
- 4 Start the debugger and load `naftpapp`.
- 5 Enable flash on your module.
- 6 Start the application.
`naftpapp` prompts you with the standard NET+OS configuration dialog box (unless you have disabled this feature).
- 7 Verify that the network settings are correct, and change them if necessary.

Step 4: Sending `rom.bin` to the module

- To send `rom.bin` of the bootloader to the module:

- 1 Open a command shell.
- 2 Change to this directory:
`c:/netos/src/bsp/your-platform`
 where you replace `your_platform` with the name of your platform.
- 3 To start the Windows FTP client, enter this command:
`FTP a.b.c.d`
 and press Enter.
 where `a.b.c.d` is your unit's IP address.
- 4 When you are prompted for a username and password, use the `root` account. The default `root` password is `password`.
- 5 To put `ftp` in binary mode, enter:
`bin`
- 6 To download `rom.bin`, enter this command:
`put rom.bin`
- 7 When the transfer is complete, enter:
`quit`
- 8 When the application output reports it is resetting, exit from the debugger.

Step 5: Verifying the boot ROM image on the module

At this point, the bootloader has been written into the boot sector of flash. Now you need to write the application into flash.

► To write an application into flash:

- 1 Restart the `naftpapp` application in the debugger.
- 2 Change to this directory:
`c:/netos/src/examples/naftpapp/32b`
- 3 To start the Windows FTP client, type this command and press Enter:
`ftp a.b.c.d`
where `a.b.c.d` is your module's IP address.
- 4 When you are prompted for a username, use the `root` account.
The default `root` password is `password`.
- 5 To put ftp into binary mode, enter:
`bin`
- 6 To download `rom.bin`, enter this command:
`put image.bin`
- 7 When the transfer is complete, enter:
`quit`
- 8 When the application output reports that it resetting, exit from the debugger.

Step 6: Verify the contents of flash

To verify the contents of flash:

- 1 Remove the debugger.
- 2 Recycle power.
The `naftpapp` application now boots from flash.

Index



A

- accept function 126
- adding devices 153
- adding new libraries to the system 96
- Advanced Web Server Toolkit
 - documentation 109
- appconf.h file 32, 177
- application image
 - components of 42
 - header 43, 44
 - structure 42
- application samples
 - naficgi 112, 123
 - nahttp_pb 123
 - nahttp_pd 112
- ARM7 platforms, initialization
 - sequence 17
- ARM7-based hardware dependencies
 - DMA channels 82
 - endianness 83
 - interrupts 84
 - RS-232-style communications 85
 - serial ports 83
 - software watchdog 83

- system timers 83
- ARM9 platforms, initialization
 - sequence 18
- ARM9-based hardware dependencies
 - DMA channels 87
 - endianness 88
 - general purpose timers 88
 - interrupts 89
 - system clock 89
 - system timers 88
- array subscripts and MAW module 119, 120
- automatic RAM sizing 129
- AWS
 - comment tags 115
 - custom variables 113
 - customizable routines 114
 - data types of 114
 - function type, specifying 118
 - phrase dictionary technique 121

B

- bind function 126
- blerror.c file 48

- blmain.c file 48
- board support package. See *BSP*.
- boardParams.h 30
- boot ROM
 - and restoring contents of flash memory 176
 - application 173
 - image, verifying 179
- boothdr utility 41
- boothdr.dat file 6
- boothdr.exe 5
- bootldr.dat file 46
- bootloader Makefiles
 - organization of 96
- bootloader utility
 - limitations of 47
- BSP 3
 - and NET+OS 3
 - configuration files, modifying 28
 - customizing for application hardware 25
 - defined 11
 - tree structure 12
- bsp_drivers 26
- bsp_sys 32, 33, 173
- bsp.c file 89
- building
 - all libraries 94
 - an individual library 94

C

- cgi.c file 112, 123
- cleaning libraries 96
- close function 153
- comment tags 108, 108
- compress.exe 5
- compression and phrase dictionaries 121
- configuration file 46
- configuring the TCP/IP stack 33
- connect function 126
- contents of flash memory, restoring 176
- creating a new platform
 - copying a similar platform 21
 - creating a new platform directory 21
 - updating Makefile.bsp 22
 - updating Makefile.example 22
 - updating Makefile.inc 22
 - updating Makefiles in linkerscripts directory 23
 - wireless platform changes 23
- creating Web pages 113
- crt0.o file 167
- crt0.S file 167
- custom variables and AWS 113
- customization hooks 48
- customizeGetMACAddress function 50
- customizeLed.c file 29
- customizeReset.c file 29
- customizing the BSP for application hardware 25
- Cygwin standard C library
 - and device drivers 153
 - and startup crt0.o file 166
 - modifying 166

D

- data abort 172

- data passing functions 165
- ddi.h file 153
- DDIFirstLevelInitialization 154
- DDISecondLevelInitialization 154
- debugger initialization scripts 60
- debugger_files file 6
- debugging initialization code
 - ARM7 platforms 64
 - ARM9 platforms 67
- default configuration file 46
- device driver functions
 - deviceClose 159
 - deviceEnter 156
 - deviceInit 157
 - deviceIoctl 164
 - deviceOpen 158
 - deviceRead 160
 - deviceWrite 162
- deviceClose function 159
- deviceEnter function 156
- deviceInfo structure 153
- deviceInfo structures 153
- deviceInit function 157
- deviceIoctl function 164
- deviceOpen function 158
- deviceRead function 160
- devices
 - adding 153
 - drivers and ThreadX kernel 167
- devices.c file 153
- deviceTable array 153
- deviceWrite function 162
- DHCP/BOOTP client 47
- diagnosing errors 172

- dialog.c file 32
- dictionary, adding new phrases 122
- Digi JTAG Link debugger 63
- downloadImage routine 48, 56
- drivers, enabling 26

E

- enabling drivers 26
- error
 - fatal 172
 - handler, installing 121
- error and exception handlers,
 - modifying 31
- Ethernet MAC address, restoring
 - original 174
- exceptions
 - diagnosing 173
 - types of 172

F

- fast interrupt 173
- fatal error 172
- file.c file 112, 123
- files 177
- flash driver
 - adding or dropping support for a
 - part 126
- flash memory, restoring contents of 176
- flashparts.h file 126
- function stubs 108

G

- generating
 - an image 45
- getDefaultFilename routine 48, 55
- getMacAddress routine 48, 50
- getpeername function 126
- getsockname function 126
- getUsername function 114
- GPIO configuration, setting 25
- gpio.h file 25, 27

H

- h files 7
- hard-coding the MAC address 50
- hardware dependencies for ARM7-based modules
 - DMA channels 82
 - endianness 83
 - interrupts 84
 - RS-232-style communications 85
 - serial ports 83
 - software watchdog 83
 - system timers 83
- hardware dependencies for ARM9-based modules
 - DMA channels 87
 - endianness 88
 - general purpose timers 88
 - interrupts 89
 - system clock 89
- hardware dependencies of ARM9-based modules
 - system timers 88
- hierarchy of Makefile 93

- hooks, customization 48
- hooks. See *function stubs*.
- HyperTerminal 177

I

- image, generating 45
- INIT.s file 17
- Initialization sequence
 - ARM7 platforms 17
 - ARM9 platforms 18
- Internet Address Manager (IAM) 3, 126
- interrupt tables 28
- ioctl function 153
- isImageValid routine 48, 51

K

- keyword/value pairs in configuration file 47

L

- LEDs
 - and troubleshooting 173
 - observing during power cycle 173
- libc.a file and library 167
- libraries
 - adding new 95
 - adding new to the system 96
 - building all 94
 - building an individual 94
 - cleaning 96
- library makefile variables 95
- library directory structure 95

limitations of the bootloader utility 47
list.bat file 110, 122

M

MAC address 48
 and hard-coding 50
 locating on module 175
 restoring to a module 174
Macraigor Raven debugger 63
maintaining Web content 122
MAJIC/MAJICO probe 61
Makefile 177
 hierarchy of 92
 using 98
Makefile.bsp file 22
Makefile.examples file 22
Makefile.inc file 22
MAW module
 and array subscripts 120
 and semaphore timeout 119
 error handling 121
mawInstallErrorHandler function 121
mawInstallSubscriptsFunction
 function 120
mawInstallTimeoutFunction
 function 120
mawSetAccessTimeout routine 119
memory aliasing (ARM7) 70
memory map (ARM9) 72
Mentor Graphics MAJIC/MAGICO
 debuggers 63
MIB compiler 5
Microsoft HyperTerminal 177
modifying Cygwin's standard C library

 and startup file 166
modifying error and exception
 handlers 31
modifying the POST 33
modifying Web content 122
module
 reserializing 173
 restoring Ethernet MAC address 174
 verifying boot ROM image on 179

N

NABIReportError routine 49
NABIReportError routine 48
nafcgi sample application 112, 123
naftpapp application 176, 177, 179
nahttp_pd sample application 112, 123
NET+OS
 device driver interface (DDI) 165
 supported platforms 16
 tree structure 4
netarm1_v.c file 111
netarm1.c file 111
new libraries, adding to the system 96
new platform
 configuring 25
new platform, creating 21

O

observing LEDs during power cycle of
 module 173
open function 153
original Ethernet MAC address,
 restoring 174

P

- PbSetUp.txt file 111
- PBuilder
 - linking the application 112
- PBuilder utility
 - described 108
 - sample applications 123
- PBuilder,running 109
- pci.c file 28
- pci.h public header file 29
- phrase dictionaries and compression
 - example 121
- platform 21
- platforms supported by NET+OS 16
- porting pre-NETOS 7.x PPP applications 130
- POST, modifying 33
- power cycling a module, LEDs and 173
- Power On Self Test (POST) 18, 19
- PPP applications
 - adding a route 130
 - adding PAP user/password 130
 - checking link status 131
 - closing interface 133
 - creating interface 131
 - deleting a route 130
 - dial string settings 135
 - getting peer assigned local address 132
 - initializing serial port configuration 134
 - setting authentication and compression 133
 - setting the ring count 134
- prefetch abort 172

private structures and routines 109

R

- RAM image and bootloader utility 40
- read function 153
- recvfrom function 126
- reserializing a module 173
- restoring the contents of flash memory 176
- return values for NET+OS DDI functions 165
- ROM image and bootloader utility 40
- rom.bin, sending to module 178
- RpFormInput tag 116
- RpFormSingleSelect tag 117
- RpFormTextAreaBuf tag 116
- RpPages.c file 111
- RpSingleSelectOption tag 117
- RpUsrDct.txt file 111, 121

S

- sample applications 112
 - naficgi 112, 123
 - nahttp_pd 112, 123
- security 118
 - security realms, defined 112
- security.c file 112, 123
- semaphore timeout and MAW module 119
- sendto function 126
- setting the GPIO configuration 25
- setup functions 165
- shouldDownloadImage routine 48, 53

simpleSerial.c file 29
smidump.exe 5
software interrupt 173
spi_blmain.c file 48
spiboothdr.exe 5
structure of the library directory 95
supported platforms 16

T

TCP/IP stack, configuring 33
TFTP client and bootloader utility 47
ThreadX kernel
 and NET+OS device drivers 167
tree structure
 BSP 12
 NET+OS 4

U

undefined interrupt 173
unexpected exceptions 172
User Datagram Protocol (UDP) stack and
 bootloader utility 47

V

v.c file 111

W

Web content
 maintaining and modifying 122
Web page
 creating 113