



NET+Works with GNU Tools BSP Porting Guide

Making
DEVICE NETWORKING
easy™

NET+Works with GNU Tools BSP Porting Guide

Operating system/version: 6.3
Part number/version: 90000726_B
Release date: March 2006
www.digi.com

©2006 Digi International Inc.

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

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

Information 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.

Contents

| | |
|---|--------------|
| Chapter 1: Introduction | 1 |
| Overview | 2 |
| Application development | 2 |
| What is the board support package? | 3 |
| Why does the target BSP need to change from the NET+ARM development board BSP? | 3 |
| What are the benefits of following the NET+ARM reference design? | 4 |
| What's the best way to add my target hardware BSP platform? | 4 |
| NET+OS tree structure | 5 |
| bsp | 6 |
| examples | 6 |
| bin | 6 |
| h | 7 |
| gnusrc | 7 |
| smicng | 8 |
| arm7 | 8 |
| arm9 | 8 |
| debugger_files | 8 |
| docs | 8 |
| Chapter 2: NET+OS BSP for ARM7 | 9 |
| Overview | 10 |
| Platforms | 10 |

| | |
|---|----|
| Initialization | 11 |
| Initializing hardware | 11 |
| Initialization sequence | 11 |
| C library startup | 11 |
| NABoardInit | 12 |
| ROM bootloader | 12 |
| BSP tree structure | 13 |
| Top-level directory | 13 |
| bootloader subdirectory | 13 |
| devices directory | 14 |
| platforms directory | 14 |
| Customizing the BSP for application hardware | 15 |
| Follow the reference design | 16 |
| Verify the features your hardware supports | 16 |
| Task 1: Purchase and assign Ethernet MAC addresses | 16 |
| Task 2: Create a new platform subdirectory | 17 |
| Task 3: Building and modifying the BSP Makefile | 17 |
| Task 4: Modify the linker scripts | 18 |
| Task 5: Modify BSP configuration files | 20 |
| Task 6: Modify the new BSP to start up the required drivers | 24 |
| Task 7: Modify the format of BSP arguments in NVRAM | 25 |
| Task 8: Modify error and exception handlers | 27 |
| Task 9: Verify the debugger initialization files | 28 |
| Task 10: Debug the initialization code | 29 |
| Task 11: Modify the startup dialog | 34 |
| Task 12: Modify the POST | 34 |
| Task 13: Modify the ACE | 34 |
| Other BSP customizing | 35 |
| BSP_NVRAM_DRIVER | 35 |
| TCP/IP stack | 36 |
| File system | 37 |

| | |
|--|----------------|
| Chapter 4: Linker Files | 75 |
| Overview | 76 |
| Linker files provided for sample projects..... | 76 |
| Basic GNU Tools section of the linker files | 77 |
| NET+OS section of the linker files..... | 77 |
| Address mapping (ARM9 only) | 78 |
| NET+OS memory map (ARM9 only) | 80 |
| Memory aliasing in NET+OS (ARM7 only) | 81 |
| Chapter 5: Adding Flash | 83 |
| Flash table data structure..... | 84 |
| Supporting larger flash..... | 88 |
| Chapter 6: Device Drivers | 89 |
| Overview | 90 |
| Adding devices | 90 |
| deviceInfo structure..... | 90 |
| Device driver functions | 91 |
| Return values..... | 100 |
| Modifications to Cygwin's standard C library and startup file..... | 101 |
| Modifying the libc.a library and crt0.o startup file | 102 |
| NET+OS device drivers | 103 |
| Device driver interface | 103 |
| Chapter 7: Hardware Dependencies for ARM7-based Platforms | 105 |
| Overview | 106 |
| DMA channels..... | 106 |
| Ethernet PHY | 107 |
| ENI controller | 107 |
| Serial ports | 107 |
| Software watchdog..... | 108 |

| | |
|------------------------------------|-----|
| Endianness | 108 |
| System clock..... | 108 |
| BSP_CLOCK_SOURCE..... | 108 |
| XTAL1_FREQUENCY | 109 |
| CRYSTAL_OSCILLATOR_FREQUENCY..... | 109 |
| PLL Control register setting | 109 |
| System timers | 110 |
| Memory map..... | 111 |

Chapter 8: Hardware Dependencies for ARM9-based Platforms

113

| | |
|---------------------------------------|-----|
| Overview | 114 |
| DMA channels..... | 114 |
| Ethernet PHY | 114 |
| Endianness | 115 |
| General purpose timers..... | 115 |
| System timers | 115 |
| All other general purpose timers..... | 116 |
| Interrupts | 116 |
| System clock..... | 117 |
| Chip selects..... | 117 |
| Memory map..... | 118 |

Chapter 9: Porting NET+OS v6.0 Applications to NET+OS v6.3

119

| | |
|-------------------------------------|-----|
| Overview | 120 |
| BSP Makefile..... | 120 |
| Application Makefiles | 121 |
| Linker scripts..... | 121 |
| Bootloader files | 121 |
| Cache API | 122 |
| Embedded Networking Interface | 122 |
| ISR API | 122 |

| | |
|---|-----|
| RAM API | 122 |
| Real Time Clock driver..... | 123 |
| SYSCLK API..... | 123 |
| GPIO configuration | 123 |
| SPI API | 124 |
| Stack sizes for exception handlers..... | 124 |
| Interrupt priorities | 124 |

Chapter 10: Porting NET+OS v6.1 Applications to NET+OS v6.3

| | |
|---|-----|
| Overview | 125 |
| BSP Makefile..... | 126 |
| Application Makefiles | 126 |
| Linker scripts | 127 |
| Bootloader files | 127 |
| Client parallel driver | 127 |
| I2C driver | 128 |
| Interrupt Service Routine (ISR) API | 128 |
| MMU API | 129 |
| PLL functions | 129 |
| Real time clock driver | 130 |
| GPIO configuration | 130 |
| Timer driver | 130 |
| SPI API | 131 |
| Network heap caching | 131 |
| USB host API | 131 |

Library Makefile System

| | |
|-------------------------------------|-----|
| Overview | 136 |
| Makefile hierarchy..... | 137 |
| Building all libraries | 138 |
| Building individual libraries | 138 |
| Library directory structure | 139 |

| | |
|---|------------|
| Library Makefile variables | 139 |
| Adding a new libraries to the system | 140 |
| Cleaning libraries | 140 |
| Bootloader Makefile | 140 |
| Example: using the Makefile | 142 |
| Appendix B: Application Makefile | 143 |
| Building applications | 144 |
| Application Makefiles | 144 |
| Sections of the Makefile | 145 |
| Makefile hierarchy | 147 |
| Makefile targets | 148 |
| Building an application | 148 |
| Cleaning an application | 148 |
| Porting an application to a new platform | 149 |
| Appendix C: Customizing the SPI Bootloader | 151 |
| Overview | 152 |
| SPI bootloader application images | 153 |
| ROM image | 153 |
| RAM image | 153 |
| Application image structure | 154 |
| Application image header | 154 |
| boothdr utility | 156 |
| spibootldr utility | 157 |
| Generating an image | 158 |
| Configuration file | 158 |
| General bootloader limitations | 159 |
| Customizing the SPI bootloader utility | 160 |
| Customization hooks | 160 |

| | |
|---|------------|
| Appendix D: Customizing the ROM Bootloader | 169 |
| Overview | 170 |
| Bootloader application images | 170 |
| ROM image | 170 |
| RAM image | 171 |
| Application image structure | 172 |
| Application image header | 172 |
| boothdr utility..... | 174 |
| Generating an image | 174 |
| Configuration file | 175 |
| General bootloader limitations..... | 176 |
| Overview of customizing | 176 |
| Customization hooks | 177 |
| Appendix E: Customizing the Address Configuration | |
| Executive | 185 |
| Overview | 186 |
| Configuring ACE | 186 |
| Setting the static IP configuration | 187 |
| Setting DHCP configuration | 188 |
| AUTOIP configuration | 189 |
| Stopping ACE | 189 |
| Appendix F: Processor Modes and Exceptions | 191 |
| Overview | 192 |
| Vector table | 192 |
| IRQ handler | 194 |
| Servicing AHB interrupts in ARM9 based NET+ARM processor. | 195 |
| Servicing Bbus interrupts in ARM9 based NET+ARM processor | 195 |
| Changing interrupt priority | 196 |
| Interrupt service routines | 203 |
| Installing an ISR | 203 |
| Disabling and removing an ISR | 203 |
| ARM9 FIQ handlers..... | 203 |
| ARM7 FIQ handlers..... | 204 |

| | |
|--|-----|
| Overview | 206 |
| Block pools..... | 206 |
| Byte pools..... | 206 |
| Network heap application tuning | 207 |
| Memory usage in TCP connections | 207 |
| Active close of a TCP connection | 208 |
| Time wait state of a TCP connection..... | 209 |
| Using a connection reset instead of an orderly close | 209 |
| Maximum number of sockets..... | 210 |

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 6.3 and how to use it as part of your development cycle. Part of the NET+Works integrated product family, NET+OS is a network software suite optimized for the NET+ARM.

Software release

This guide supports NET+OS 6.3. By default, this software is installed in the `C:/netos63_gnu/` directory.

Who should read this guide

This guide is for software engineers and others who use NET+Works for NET+OS.

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

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

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 → option | Menu commands. The first word is the menu name; the words that follow are menu selections. |
| monospaced type | File names, pathnames, and code examples. |

What's in this guide

This table shows where you can find information this guide:

| To read about | See |
|--|--|
| An overview of the board support package | Chapter 1, “Introduction” |
| Using the board support package to create a platform for your customized hardware for ARM7-based platforms | Chapter 2, “NET+OS BSP for ARM7” |
| Using the board support package to create a platform for your customized hardware for ARM9-based platforms | Chapter 3, “NET+OS BSP for ARM9” |
| The linker files that are provided for sample projects | Chapter 4, “Linker Files” |
| How to update flash memory | Chapter 5, “Adding Flash” |
| Device drivers and device definition | Chapter 6, “Device Drivers” |
| NET+OS hardware dependencies for platforms that use the NS7520 and NET+50 processor | Chapter 7, “Hardware Dependencies for ARM7-based Platforms” |
| NET+OS hardware dependencies for platforms that use the NS9360 and NS9750 processors | Chapter 8, “Hardware Dependencies for ARM9-based Platforms” |
| The differences between the APIs in NET+OS 6.0 and NET+OS 6.3 | Chapter 9, “Porting NET+OS v6.0 Applications to NET+OS v 6.3” |
| The differences between the APIs in NET+OS 6.1 and NET+OS 6.3 | Chapter 10, “Porting NET+OS v6.1 Applications to NET+OS v 6.3” |

In addition, a series of appendixes provide information about:

- Customizing the SPI bootloader, the ROM bootloader, and the Address Configuration Executive (ACE)
- BSP Makefiles and application Makefiles
- Processor modes and exceptions
- Memory usage

Related documentation

- *NET+Works Quick Installation Guide* describes how to install the hardware.
- *NET+Works with GNU Tools Getting Started Guide* provides a brief, hands-on exercise.
- *NET+Works with GNU Tools Programmer's Guide* describes how to use NET+OS to develop programs for your application and hardware.
- The NET+Works online help describes the application program interfaces (APIs) that are provided with NET+OS. The online help is located in `C:\netos63_gnu\Documentation`.

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

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

Documentation updates

Digi occasionally provides documentation updates on the Web site.

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

Customer support

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

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



Introduction



C H A P T E R 1

This chapter provides an overview of the Board Support Package (BSP) software, describes how this software is segmented from higher layer application software, and provides hardware design guidelines to minimize the cost of the software effort. In addition, this chapter describes the NET+OS tree structure.

Overview

After you complete a system analysis that includes data throughput, I/O and processing requirements and select the NET+ARM processor as the target processor, you can begin two efforts: hardware design and software development.

Hardware design might require a complete new board design, reusing or modifying a previous design, or using an off-the-shelf NET+ARM module. Target hardware often is unavailable to software developers for weeks - and sometimes even months. To minimize product time-to-market, you can begin software development immediately by partitioning the effort into two distinct tasks: application development and the board support package (BSP).

Application development

Application development involves piecing together hardware-independent, high-level software components, while the BSP provides hardware-specific services along a standardized application programming layer (API) to the application software.

By using a NET+ARM development board and its associated BSP, you can begin software development immediately. NET+OS is delivered with BSPs to support all NET+ARM development board platforms and all DIGI Connect products. Each BSP is tailored to support the development board's specific target processor (for example, the NS9360 or NS7520) and the components that surround the processor (memory and PHY).

The development board is ideal for prototyping general network services, including Web pages, private management information bases (MIBs), FTP servers, SMTP clients, or network startup characteristics such as DHCP or Auto IP. In addition, you can prototype non-volatile system configuration, I/O protocols, field upgrade mechanisms, or file system requirements effectively with a NET+ARM development board.

Alternatively, the BSP enables you to create the platform-specific software needed to support a hardware platform. Because the BSP is hardware-specific, completing this software requires the target hardware - and so must wait until the target hardware is debugged and available.

When the hardware target becomes available, you can create the BSP and port the application to the target hardware. Because application software maintains the BSP standardized API, it reduces the effort required to port the application to the new target hardware and BSP. Minimizing software development cost and time-to-market is an important design goal.

This guide describes best practices for modifying a standard NET+ARM development board BSP platform to support your target hardware needs and operational characteristics.

Note that throughout this document, the terms *BSP* and *platform* are used interchangeably.

What is the board support package?

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 Transmission Control Protocol/Internet Protocol (TCP/IP) 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.

Why does the target BSP need to change from the NET+ARM development board BSP?

The NET+ARM development boards are generic designs that contain a broad range of hardware, including RAM, flash, serial line drivers, and an Ethernet PHY, and the circuitry needed to support the specific target processor peripherals (such as PCI clock circuitry or a USB PHY). Overall, the NET+ARM development boards were designed to maximize the range of applications that can be prototyped, and not to minimize cost or maximize performance.

Most commercial products would not need all the parts options on a NET+ARM development board or might require changes to the development board design. For example, the development board might include an unnecessarily large (and more expensive) flash, or the application might need special processing that requires a larger SDRAM. Alternatively, different components can be used, such as faster SDRAMs for higher performance, or slower SDRAMs for lower cost. Some applications might require more extensive modifications that include special peripherals, such as a wireless compact flash or a cryptographic accelerator.

All modifications to the development board require special BSP software support.

What are the benefits of following the NET+ARM reference design?

The NET+ARM processors have many possibilities for connecting addressable peripherals; a good example is the use of chip selects and memory. When board designers connect SDRAM to a NET+ARM processor, they can use any chip select that supports dynamic RAM. From a hardware perspective, any chip select is as good as another, and the choice might even be arbitrary. From a software perspective, however, not all chip selects are equal, and an arbitrary board design decision might have major implications on software.

To reduce the software development cost of modifying and maintaining a BSP, and to reduce the cost of future upgrades to NET+OS, Digi strongly recommends that you follow the NET+ARM development board reference design.

What's the best way to add my target hardware BSP platform?

Digi recommends that you use a preexisting functional BSP as a template for new target system BSPs. For best results, use these general steps:

- 1 Determine the closest matching NET+ARM development board BSP.
- 2 Copy the BSP platform that best matches your target platform, and paste it in your `platforms` directory.

For example, to create a new ns9360 platform, copy the `ns9360_a` platform and paste it in `custom9360`.

- 3 Update the BSP `Makefile` to support the new platform.
- 4 Build the new BSP platform.
- 5 Compile and link an application using the new BSP.

- 6 Test the debugger with an application using the new BSP.
- 7 Test flash-based images using the new BSP.
- 8 Apply custom hardware modifications to the new BSP.

This procedure provides the most reliable set of instructions needed to create a new BSP platform. Use these steps to create a template for porting BSPs from previous versions of NET+OS.

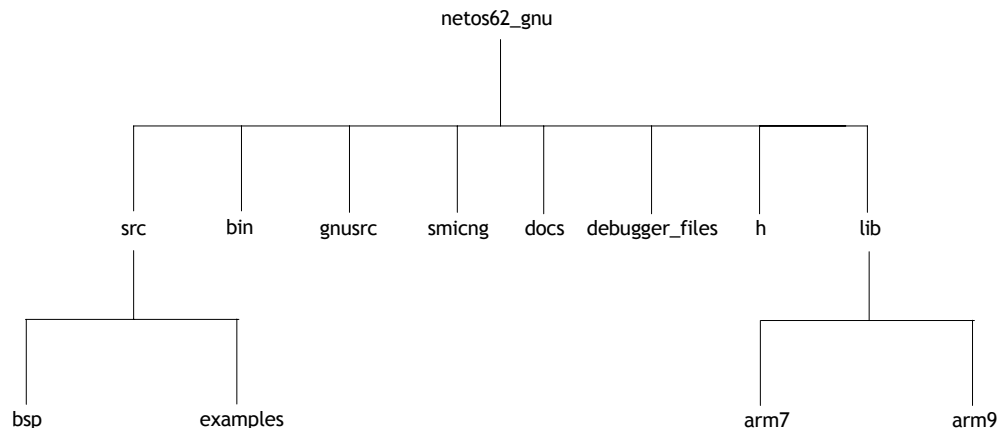
How does the NET + OS structure support multiple BSP platforms?

In previous releases of NET+OS, the tree could support only one version of a BSP. This version, however, can support multiple BSPs. This has been achieved by providing better fanning out of the `lib` (library) tree, including an `arm7` and `arm9` sub-tree, and by fanning out individual BSP directories under these sub-trees.

Additionally, previous releases could support only one compilation of the bootloader because this folder was located under the BSP sub-tree. The `rom.bin` bootloader image has been moved to the `BSP platforms` folder.

NET+OS tree structure

The NET+OS tree structure is divided into subdirectories, with `netos63_gnu` as the root directory. This figure shows how the tree is set up:



The next sections describes the subdirectories under `netos63_gnu`:

- `bsp`
- `examples`
- `bin`
- `h`
- `gnusrc`
- `smicng`
- `arm7`
- `arm9`
- `debugger_files`
- `docs`

bsp

The BSP is located in `netos63_gnu/src/bsp`. All the initialization code, device drives, and platform-specific configuration files are stored in subdirectories under the `BSP` directory.

examples

The sample applications are located in `netos63_gnu/src/examples`. These applications demonstrate how to use the APIs for the NET+OS software libraries.

Be aware that some of the sample applications require platform-specific hardware and will not compile if the required hardware is not available. For example, the USB-related sample application compile and work only for NS9750 and NS9360 processors.

bin

The binary files that are executable on a PC and used by NET+OS are located in `netos63_gnu/src/bin`. Some of the most commonly used files are:

- `spiboothdr.exe` - Uses the `netos63_gnu/src/bsp/platforms/"my platform"/spibootldr.dat` configuration file for SPI devices.
- `smicng.exe` - MIB compiler for SNMP MIBs written in either the SMI v1 or SMI v2 formats.

- `compress.exe` - Compress the application image's `.bin` file to save memory in flash.
- `boothdr.exe` - Inserts a header at the beginning of the image based on information read from the `netos63_gnu/src/bsp/platforms/my_platform/boothdr.dat` configuration file.

This program 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`:

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

h

The public API header files are located in `netos63_gnu/h`. 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.

gnusrc

These files allow interfacing the GNU C library I/O functions to the file systems and the C library time functions to the real time clock driver. The GNU I/O and time driver interface functions are located in `netos63_gnu/gnusrc`.

smicng

`smicng` subdirectories consist of MIBS that are written in either the SNMP v1 or SNMP v2 formats. The files are located in `netos63_gnu/smicng`.

arm7

The netos libraries and the BSPs for ARM7 devices are located in subdirectories of `netos63_gnu/lib/arm7`.

arm9

The netos libraries and the BSPs for ARM9 devices are located in subdirectories of `netos63_gnu/lib/arm9`.

debugger_files

This file contains sample gdb initialization scripts and configuration setting files for the Raven. In addition, the file contains the `gdbThreadX` script, which sets up macros to view ThreadX structures. This file is located in `netos63_gnu/debugger_files`.

docs

All the NET+OS hardware- and software-related documentation is located in `netos63_gnu/docs`. This directory contains the online help for the NET+OS APIs and PDF versions of the hardware and software guides.



NET+OS BSP for ARM7



C H A P T E R 2

This chapter describes how to create a platform for your customized hardware using the NET+OS board support package (BSP) for ARM7-based platforms such as the NET+50 and NS7520.

Overview

The board support package (BSP) contains the drivers, the board-specific software, and a customizable directory for each supported platform. When you port a new platform to NET+OS 6.3, you typically need to modify the files in the `platforms` directory. If you are using a standard development kit, you can use one of the existing platforms with no modifications.

This chapter describes the overall structure of the NET+OS BSP, how to add in a new platform, and how to debug a new platform.

Platforms

This table shows the list of supported platforms provided with NET+OS 6.3. If you are adding a new platform to NET+OS, start with a platform that is similar to yours.

| Platform | CPU type |
|-------------|----------|
| net50bga_a | NET+50 |
| net50_d | NET+50 |
| ns7520_a | NS7520 |
| connectme | NS7520 |
| connectem | NS7520 |
| connectwime | NS7520 |
| connectwiem | NS7520 |
| connectsp | NS7520 |

For a description of your platform, see the hardware reference for the processor you are using and the jumpers and components guide for your development board.

Initialization

This section describes the powerup and initialization of NET+OS. In general, you do not need to modify the initialization code.

Initializing hardware

The hardware initialization code is located in `src/bsp/init/arm7`. The `main()` routine is located in `src/bsp/common/main.c`.

Initialization sequence

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

- 1 Initialize supervisor mode and disable interrupts.
- 2 Initialize the PLL (NET+50 only).
- 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.

C library startup

After hardware initialization, the C library `START` routine is called by the `Reset_Handler`, which is located in the `INIT.S` file.

The size of the stack for the C library is specified in the `customize.ldr` file. The default stack size for the C library is 12K. If you are not using C++, you can reduce this size to 8K. The `main()` routine is located in `src/bsp/common/main.c`.

The `main()` routine must perform these steps:

- 1 If the power-on self-test (POST) is enabled, execute it.
- 2 Set up the vector table.
- 3 Call `NABoardInit` (described in the next section).
- 4 Perform the first level device driver initialization.
This step performs low-level device driver initialization and is executed before the OS is loaded.
- 5 If C++ is enabled, initialize the C++ libraries.
- 6 Start ThreadX.

NABoardInit

This routine completes the hardware initialization that was started in `INIT.s`.

The `NABoardInit` routine must do these steps:

- 1 Read the chip revision and store it in `g_NAChipRevision`.
- 2 Initialize the low level flash interface.
- 3 Set up non-volatile random access memory (NVRAM).

ROM bootloader

The NET+OS ROM bootloader is a small program that is programmed into ROM. The application also is programmed into flash in a compressed format. At power-up, the bootloader decompresses the application into RAM, and then executes it from RAM.

The advantage of using the ROM bootloader is twofold:

- Less flash memory is required because the application image is compressed.
- The applications generally run faster from SDRAM.

The bootloader is built as part of the BSP. The ROM image for the bootloader is in the specific platform's directory and is called `rom.bin`. For details about the bootloader, see Appendix D, "Customizing the ROM Bootloader."

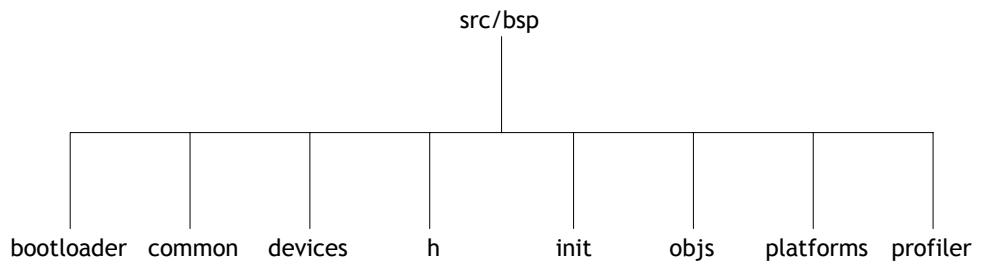
BSP tree structure

These sections describe and illustrate the BSP tree structure.

Top-level directory

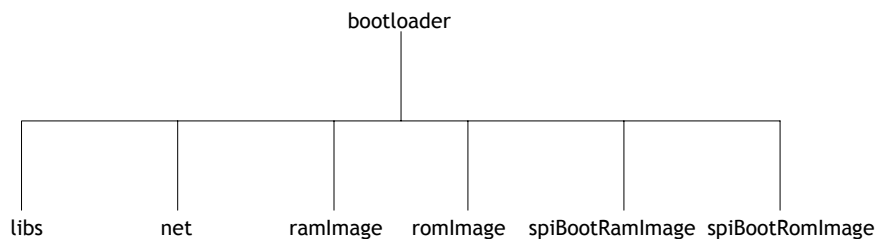
The NET+OS BSP is located in the `src/bsp` directory. The top level directory contains the Makefile for the BSP and the Makefile for the bootloader.

This figure shows the top level directory:



bootloader subdirectory

The bootloader subdirectory contains the source code for the SPI and ROM-based bootloaders. This figure shows the `bootloader` subdirectory:



The `bootloader` has two parts: the ROM image and the RAM image.

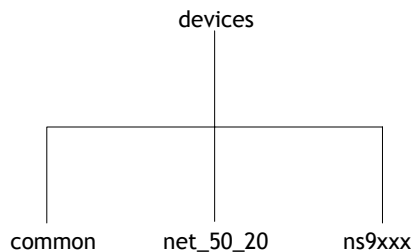
Because the `bootloader` size is kept to less than 64K, the `libs` directory contains the libraries that are linked into the `bootloader`. The `bootloader` does not link in the standard NET+OS libraries.

The `bootloader` directory has six subdirectories:

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

devices directory

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

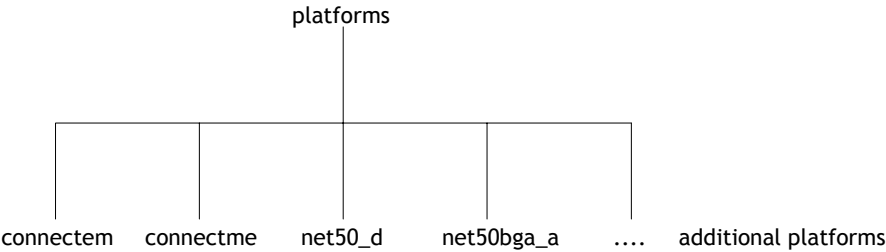


The device drivers are separated into three directories:

- `common` - Contains the device drivers that are common to all processors, such as serial and Ethernet
- `net_50_20` - Contains the drivers for the NS7520 and the NET+50
- `ns9xxx` - Contains the drivers for the NS9360 and NS9750

platforms directory

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



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

Customizing the BSP for application hardware

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

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

| Task | Action |
|------|--|
| 1 | Purchase Ethernet media access controller (MAC) addresses from the IEEE. |
| 2 | Create a new platform directory. |
| 3 | Modify the BSP Makefile. |
| 4 | Modify the linker scripts. |
| 5 | Modify the BSP configuration files to support your application hardware. |
| 6 | Modify the BSP to start up the required drivers. |
| 7 | Modify the format of BSP arguments in NVRAM. |
| 8 | Modify the error and exception handlers. |
| 9 | Verify the debugger initialization files. |
| 10 | Debug the initialization code. |
| 11 | Modify the startup dialog. |

| Task | Action |
|------|---|
| 12 | Modify the power-on self-test (POST) routines. |
| 13 | Modify the Address Configuration Executive (ACE), which controls TCP/IP configuration on startup. (For details about ACE, see the online help.) |

Follow the reference design

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

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

Verify the features your hardware supports

Make sure your hardware supports these features:

- Flash at CS0
- RAM (32-bit wide) at CS1
- NVRAM at CS3
- A JTAG port, which allows you to use an in-circuit emulator (ICE) to debug the hardware and software. This feature is essential when you are bringing up a new board.
- Extra serial port to send diagnostic messages for debugging.
- Enough RAM to run your entire application, even if your product runs out of ROM. Being able to run an application from RAM greatly simplifies debugging.
- A way to disable flash ROM. This feature is necessary because flash can be accidentally overwritten; in this situation, the NET+ARM CPU executes garbage instructions when you start it up.

Task 1: Purchase and assign Ethernet MAC addresses

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

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

Task 2: Create a new platform subdirectory

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

The `src/bsp/platforms` directory contains a set of subdirectories for each supported platform. Each subdirectory contains all the code for a particular development board. For example, the `ns7520_a` subdirectory contains the code needed to support the NS7520 development board.

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

► To create the new platform subdirectory:

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

Modify the subdirectory to interoperate with your hardware.

Be aware that it is unusual to modify any code outside the platform tree unless you are adding your own drivers or modifying existing devices.

If you believe you must modify files outside the platform tree, contact Digi technical support for confirmation.

Task 3: Building and modifying the BSP Makefile

The next step is to build the BSP.

► To build the BSP:

- 1 Double-click the X-Tools icon on your desktop.
- 2 Change to the directory in which you installed NET+OS.

3 Add these lines to the `Makefile.inc` file:

```

ifeq "$(PLATFORM)" "my_platform"
CHIP=NS7520
PROCESSOR=arm7
endif
- If you are using a ns7520_a, enter:
  CHIP=NET50
  ifeq '$(PLATFORM)' 'my_platform'
  CROSS_DEFS_PROCESSOR += -DNS7520
  endif
- if you are using a net50bga_a, enter:
  CHIP=NET50
  ifeq '$(PLATFORM)' 'my_platform'
  CROSS_DEFS_PROCESSOR += -DNET50
  endif

```

4 Do one:

```

- Either enter:
  make PLATFORM=platform
- Or enter:
  export PLATFORM=platform
  make

```

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

This builds the new BSP and the `bootloader`.

Task 4: Modify the linker scripts

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

Constants you might need to change

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

| Constant | Description |
|--------------------------|--|
| RAM_SIZE | <p>The size of the RAM part on the board.</p> <p>The linker generates an error if the application is too large to fit in RAM.</p> |
| FLASH_SIZE | <p>The size of the flash part on the board.</p> <p>The linker generates an error if a ROM-based application is too large to fit in ROM.</p> |
| FLASH_START | The starting address of flash. For the NS7520 and NET+50 processors, this address is typically 0x2000000. |
| RAM_START | The starting address of RAM. |
| FILE_SYSTEM_SIZE | The number of bytes to be allocated for the file system in flash. |
| BOOTLOADER_SIZE_IN_FLASH | <p>The amount of flash ROM to be reserved for the <code>bootloader</code>. You also can use this constant to calculate where the application image starts in flash. The bootloader shipped with NET+OS fits into one sector of flash, typically 64 K. It is important that this is large enough to fit the bootloader ROM image, which is in the <code>src/bsp/platforms/your platform/rom.bin</code> directory.</p> |
| MAX_CODE_SIZE | <p>The largest possible size of the uncompressed application image. Use this constant to reserve enough RAM to hold the application image.</p> <p>The bootloader uses this constant to reserve a section of memory to hold the application.</p> <p>When you create your application, use <code>MAX_CODE_SIZE</code> to reserve memory in uncached memory so that the alias of the application does not collide with RAM used for data storage.</p> <p>The compression algorithm used by the bootloader generally achieves 2:1 compression. A good rule of thumb is to set this constant to twice the amount of flash available to hold the compressed application image.</p> <p>The linker generates an error if an application image is larger than this value.</p> |

| Constant | Description |
|------------------|---|
| INIT_DATA_START | Determines where the init data is stored in RAM. The init data section stores information read by the initialization code that needs to be accessed later. Enough space must be left from the start of RAM to hold the vector table, and possibly a FIQ routine, if you decide to write one. |
| INIT_DATA_SIZE | The size of the memory area used to hold the start of switches and buttons read at powerup. |
| CODE_START | The start of ROM code. |
| NVRAM_FLASH_SIZE | Determines how much flash ROM is reserved for NVRAM storage. Set this constant to 0 if flash is not used for NVRAM. |

Bootloader considerations

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

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

You must set the `ramAddress` in the `bootldr.dat` to the value of `BOOTLOADER_CODE_START` in the `customize.ldr` file.

Task 5: Modify BSP configuration files

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

The next sections describe the files you must modify to support your hardware. You may find it helpful to review the Memory Controller information in the hardware reference for the processor you are using.

Phase Lock Loop (PLL)

The PLL generates a clock when a crystal is used instead of an external oscillator.

The PLL must be configured to generate the correct clock speed. On the NS7520, the PLL is configured through pull-up and pull-down resistors; on the NET+50, the PLL is configured by the BSP. The PLL settings are stored in a table in `bsp.c` in the `platforms` directory. The default settings in the table configure the NET+50 PLL to run at 44 MHz and assume an 18.432 MHz crystal input. Modify the values in this table if your platform is different.

For more information on this table, see the online help.

Note: The NS7520 development board BSP assumes that an external oscillator will be used. To configure the BSP to use the PLL, see the online help.

bsp.c file

The `NANetarmInitData` array in `bsp.c` in the `platforms` directory holds the timing settings for the memory parts. The timing settings control the number of wait states and idle cycles. The default values in the table work for commonly used parts. Verify that the settings are correct for the memory parts on your board, and make any necessary adjustments.

Interrupt tables

You change the system interrupt priority by updating the `NAInterruptPriority` array. This allows flexible prioritization for all the NET+ARM interrupts that drive the ARM processor IRQ. The table prioritization requires lower priority interrupts early in the array and higher priority interrupts toward the end of the array. For example, the `NAInterruptPriority` array defaults to bit 0, `PORTC_PC0`, as the system's lowest priority interrupt, and bit 31, `DMA1`, as the system's highest priority interrupt.

Chip select settings

The next table lists the customization hooks in `cs.c`. This file contains the routines that configure the NET+ARM chip selects to support memory parts. You need to modify the code in these routines to support your application hardware.

| Customization hooks | Hardware feature and default values set |
|----------------------------------|--|
| <code>customizeGetRamSize</code> | Returns the total amount of RAM on the system. The default implementation does this by examining the configuration set for CS1 and CS2. It therefore assumes that RAM is connected to CS1 and, optionally, CS2. |
| <code>customizeGetScr</code> | <p>Returns the value to write to the System Control register (SCR). The return value must leave the <code>CACHE</code>, <code>CINIT</code>, <code>DMATST</code>, <code>LENDIAN</code>, and <code>SMARST</code> bits unchanged.</p> <p>These hardware features are set by <code>customizeGetScr</code>:</p> <ul style="list-style-type: none"> ■ Bus speed. Default is to run at full bus speed. ■ Bus Monitor Timer. Enabled and set for 128 clocks. ■ User mode access to ASIC registers. Enabled. ■ External bus master access to ASIC registers. Disabled. ■ Internal/External System Bus Arbiter. Internal. ■ DMA test mode. Must leave enabled during initialization. ■ Use of TEA pin. Use only for error indications. ■ Misaligned bus transfer abort. Do not generate an abort exception for misaligned transfers. ■ TA input synchronization. One state synchronization. |
| <code>customizeSetupCS0</code> | Configures CS0. The default implementation configures it to support a flash part. The timing parameters are set according to values in <code>NANetarmInitData</code> . The processor will be executing code in flash when this function is called after a power-on reset. Therefore, you must carefully write this function so that the chip select remains valid at all times while the function configures it. |
| <code>customizeSetupCS1</code> | Configures CS1. The default implementation assumes that RAM will be connected to this chip select. It automatically detects the RAM type and size and sets up the chip select accordingly. The default implementation sets the timing parameters according to the values in <code>NANetarmInitData</code> and generates a fatal error if no RAM is detected on this chip select. |
| <code>customizeSetupCS2</code> | Configures CS2. The default implementation assumes that RAM may be connected to this chip select. It automatically detects the RAM type and size and sets up the chip select accordingly. The default implementation sets the timing parameters according to the values in <code>NANetarmInitData</code> and disables the chip select if no RAM is detected. |

| Customization hooks | Hardware feature and default values set |
|---------------------|--|
| customizeSetupCS3 | Configures CS3. On development boards that support EEPROM, the default implementation sets up the chip select to support an 8K EEPROM. Otherwise, the default implementation disables the chip select. |
| customizeSetupCS4 | Configures CS4. The default implementation disables the chip select. |
| customizeSetupMMCR | <p>Sets up the Memory Management Control register (MMCR). The MMCR controls the dynamic RAM (DRAM) refresh timing and special functions for pins A25, A26, and A27. Default functionality for A26 and A27 is set by the hardware through pull-down resistors on pins A23 and A24. Usually, the software leaves the powerup settings for these pins alone.</p> <ul style="list-style-type: none">■ DRAM refresh rate. The default is to refresh at 67 KHz.■ Use of pins A25, A26, and A27. The default is to use A25 as an address line, and leave pins A26 and A27 as configured by hardware at powerup.■ Address multiplexor. The default setting is to use the internal address multiplexor and not to use GPIO port C3 to support the DRAM RAS/CAS signals. |

gpio.h file

The NS7520 has 16 pins that are multiplexed with various functions including GPIO functionality. These pins can be rapidly configured using the definitions in this file. The functions multiplexed include serial, DMA, Ethernet CAM, external IRQs, and GPIO.

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

For information about how pins are multiplexed, see the gpio.h file and the hardware reference for the processor you are using. The gpiomux_def.h public header contains definitions used by the gpio.h file.

For a detailed description of the GPIO customization, see the online help.

mii.c file

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

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

customizeLed.c file

The `customizeLed.c` file contains the structure `NALedTable` 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.

These functions determine what the system should do in case of a reset or restart request. This is where you place application-specific code 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, you modify this driver.

The driver is located in the `simpleSerial.c` file in the `devices/net_50_20/serial` directory.

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

You must configure the `bsp.h` file to enable the drivers that you want to run with your application. The default configuration works with a development board. Note that drivers that use the same GPIO pins cannot properly function at the same time. For details on all the defines in `bsp.h`, see the online help.

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

1284 controller

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

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

In addition, you must modify the development board to support the 1284 controller. For information about modifying the board to support this interface, see the jumpers and components guide for the board you are using.

To set up all the necessary GPIO settings, see the instructions in the `ReadMe` file of the `naparaclient` example.

Serial ports

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

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

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

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

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

Task 7: Modify the format of BSP arguments in NVRAM

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

You must modify these customization hooks to support your application:

| Customization hook | Description |
|--|---|
| <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.</p> <p>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.</p> <p>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. It is not used by the API libraries or in any part of the BSP except the dialog.</p> <p>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> |
| <code>customizeUseDefaultParameters</code> | <p>Determines default configuration values and returns them in a buffer.</p> <p>The default implementation determines the default values through constants set in <code>appconf.h</code>. You must modify this routine to support your application.</p> |
| <code>customizeReadDevBoardParams</code> | <p>Reads the configuration from NVRAM into a buffer. You must modify this routine to support your application.</p> |

| Customization hook | Description |
|---|--|
| <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>customizeGetIPParameters</code> | Reads the IP-related configuration values from NVRAM. |
| <code>customizeSaveIPParameters</code> | Writes the IP-related configuration values to NVRAM. |

Task 8: Modify error and exception handlers

The `errhndlr.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.h`, you can configure the default error handler to either:

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

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

Exception handler

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

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

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

- Handle these exceptions by resetting the unit.

- Blink an error code on LEDs.
- Continue execution at the point at which the exception returned.

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

For details about error and exception handlers, see Appendix F, “Processor Modes and Exceptions.”

Task 9: Verify the debugger initialization files

When you use the debugger, you must 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 are executed by the debugger before the application is downloaded and executed.

NET+OS ships with debugger scripts that initialize the supported development boards. You must create one to initialize your application hardware.

NET+Works supports the Macraigor Raven.

► To create a debugger initialization file:

- 1 Copy the debugger script for the development board that is closest to your hardware platform, and give it an appropriate name.

The debugger scripts are located in the `debugger_files` directory.

- 2 Edit the debugger script with a text editor. You see several sequences of commands like these:

```
monitor long ffc00020 = 0x0
monitor long ffc00024 = 0xf3000070
monitor long ffc00020 = 0x0000022d
monitor long ffc00028 = 0x00000001
```

These commands write values to registers in the NET+ARM.

- 3 Modify the script so that the NET+ARM is properly set up for your application hardware:
 - Set up the communications port for the Raven.
 - Configure the PLL on the NET+50 to the correct clock speed by setting PLLCR.
 - Configure the System Control register to set the correct bus speed and endianness, and disable the watchdog timer.
 - 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 has to know whether to configure the RAM chip selects, perform a memory test, and turn on cache.
- Set up the memory controller to perform the synchronous dynamic RAM (SDRAM) refresh functions.
- Set up the chip selects used for RAM, because the application code will be loaded into RAM.

The debugger initialization scripts, which are in the debugger files directory, are labeled `gdbns7520.raven` for the NS7520 and `gdbnet50.raven` for the NET+50. The debugger reads these scripts when you start to download code to the board using `gdb`.

If you are using a different type of SDRAM, you must modify the settings in these scripts. The debugger script programs the registers in the memory controller. For a detailed description of these registers, see the hardware reference for the processor you are using.

Task 10: 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 Raven 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 either the `root` directory or the `bsp` directory, rebuild the BSP with your changes.
 - a Change to the BSP directory:


```
cd src/bsp
```
 - b Enter this command:


```
make PLATFORM=my_platform
```

 where *my_platform* is the name of your platform.
 (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. Make sure that `stdio` is directed to the correct serial port. The default is `/com/0`.
- 4 Build the application:
 - a Copy the template application, which is located in:


```
src/apps/template.
```
 - b In the `src/apps/template/32b` directory, enter:


```
make clean
make all
```
- 5 Copy the debugger file that corresponds to your processor into the `src/apps/template/32b` and name it `.gdbinit`. This is read by `gdb` when it starts.
 For example, if you are using the ns7520, copy the `debugger_files/gdbns7520.raven` file.
- 6 Click `ocdRemote` for ARM7TDMI.
- 7 From the `/src/apps/template/32b` directory, enter this command:


```
gdbtk -se image.elf
```
- 8 To load your image, from the `gdb` console window, enter:


```
lo image.elf
```


- 9 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.
- 10 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.
 - The PLL registers are properly programmed for the crystal on your application hardware. The PLL should be set by the debugger script on NET+50 processors, and by pull-up and pull-down resistors on the NS7520.
 - You can read and write RAM on your application board.
- 11 Debug the initialization code by stepping through it, as described in the next section.

Debugging the initialization code

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

- 1 `INIT.s` file
- 2 `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 mods.
- 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 specific 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, which is 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 Put a breakpoint on the `eth_reset` routine (in `eth_reset.c`) and let the program run until you reach the breakpoint.
- 2 Step into the `customizeMiiReset` routine (in the `mii.c` file) and then into `customizeMiiIdentifyPhy`.
- 3 Verify that:
 - `customizeMiiIdentifyPhy` returns a value not equal to `0xffff`.
 - `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.

Task 11: Modify the startup dialog

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

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

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

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

Task 12: Modify the POST

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

You may want to create other POST routines that test additional hardware on your board.

Task 13: Modify the ACE

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

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

aceCallbacks.c

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

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

aceParams.c

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

- The `dhcp_desired_params` array. Contains a list of the Dynamic Host Configuration Protocol (DHCP) options that you want the client to request from the server.
Add any other DHCP options you want the client to request from the server.
- `NADefaultEthInterfaceConfig`. Contains the configuration that ACE uses if none is stored in NVRAM. This configuration controls which protocols are used to get an IP address and the options used with them. The default configuration uses all protocols to get an IP address. Customize this configuration as needed.

For details about these functions, see the online help.

Other BSP customizing

This section describes additional customizing you may want to do.

BSP_NVRAM_DRIVER

The `BSP_NVRAM_DRIVER` constant in `bsp.h` defines the non-volatile memory type used to store the configuration information. The next table describes the settings:

| Constant | Description |
|------------------|--|
| BSP_NVRAM_DRIVER | <p>This constant in <code>bsp.h</code> defines the non-volatile memory type used to store the configuration information. Here are the settings:</p> <ul style="list-style-type: none"> ■ <code>BSP_NVRAM_NONE</code> - No NVRAM driver is to be built ■ <code>BSP_NVRAM_LAST_FLASH_SECTOR</code> - The last sector of flash memory to be used for NVRAM ■ <code>BSP_NVRAM_SEEPROM</code> - The serial EEPROM driver is to be built ■ <code>BSP_NVRAM_SEEPROM_WITH_SEMAPHORES</code> - The serial EEPROM driver with semaphore protection is built ■ <code>BSP_NVRAM_LAST_SFLASH_SECTOR</code> - 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. These functions and constants are used for configuring the TCP/IP stack.

| Function or constant | Description |
|---------------------------|--|
| BSP_LOW_INTERRUPT_LATENCY | <p>This constant in <code>bsp.h</code> determines how the TCP/IP stack implements its critical section:</p> <ul style="list-style-type: none"> ■ To use a semaphore for the TCP/IP critical section, set <code>BSP_LOW_INTERRUPT_LATENCY</code> to <code>TRUE</code>. ■ To disable processor interrupts to implement the TCP/IP critical section, set <code>BSP_LOW_INTERRUPT_LATENCY</code> to <code>FALSE</code>. |
| BSP_WAIT_FOR_IP_CONFIG | <p>This constant in <code>bsp.h</code> determines whether the BSP waits for the stack to be configured before starting the application by calling the <code>applicationStart()</code> function. Previous versions of NET+OS always waited for the stack to be configured.</p> <p>Your application should not use any network resources until the stack has been configured by setting an IP address on at least one interface. You can use the <code>customizeAceGetInterfaceAddrInfo()</code> function to determine whether an IP address has been assigned to an interface.</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> |

| Function or constant | Description |
|------------------------------------|--|
| BSP_ENABLE_FAST_IP | <p>This constant in <code>bsp.h</code> enables Fast IP:</p> <ul style="list-style-type: none"> ■ To enable Fast IP, set <code>BSP_ENABLE_FAST_IP</code> to <code>TRUE</code>. ■ To disable Fast IP, set <code>BSP_ENABLE_FAST_IP</code> to <code>FALSE</code>. <p>Fast IP is not supported for low interrupt latency.</p> |
| BSP_ENABLE_ADDR_CONFLICT_DETECTION | <p>This constant in <code>bsp.h</code> enables IP address conflict detection, during initial IP address configuration.</p> <p>If <code>BSP_ENABLE_ADDR_CONFLICT_DETECTION</code> is defined to <code>TRUE</code>, the ACE subsystem sends ARP probes to detect IP address conflict for BOOTP, RARP, Ping ARP, and static IP address protocols. IP address conflict detection must also be enabled on a network device. You can retrieve the device configuration for IP address conflict detection by using the <code>NAGetAddrConflictData</code> function.</p> |
| NAIpSetKaInterval | <p>This function in <code>naip_global.c</code> overrides the default value for the TCP keepalive interval, which by default 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 overridden on a particular socket by the <code>IP_TTL</code> socket option.</p> |
| NAIpSetTcpMsl | <p>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 will be set to <code>(tcp_msl * 2)</code>.</p> |
| APP_NET_HEAP_SIZE | <p>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.</p> |

File system

The BSP can be configured to interface the C library file I/O functions to the file systems. NET+OS 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 | Set this constant in <code>bsp.h</code> to <code>TRUE</code> to include the native file system in the C library and create a RAM and flash volume as part of the BSP initialization process. |
| BSP_NATIVE_FS_MAX_INODE_BLOCK_LIMIT | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the percentage of the maximum number of inode blocks that can be allocated to store inodes for a volume. This constant allows specifying the upper limit of the number of blocks reserved to store inodes. Valid values are from 1 to 100. For more information, see the native <code>NAFSinit_volume_cb</code> file system API function in the online help. |
| BSP_NATIVE_FS_MAX_OPEN_DIRS | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the maximum number of open directories that the file system will track. A directory is considered open if there are open files in the directory. Valid values are from 1 - 64. For more information, see the native <code>NAFSinit_volume_cb</code> file system API function in the online help. |
| BSP_NATIVE_FS_MAX_OPEN_FILES_PER_DIR | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the maximum number of open files per directory that the file system will track. Valid values are from 1 to 64. For more information, see the native <code>NAFSinit_volume_cb</code> file system API function in the online help. |
| BSP_NATIVE_FS_BLOCK_SIZE | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the block size used for the volume. Valid values are: <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_RAMO_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_FLASHO_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> <p>NAFS_MOST_DIRTY_SECTOR - Uses the default sector transfer algorithm that selects the sector with the most dirty blocks. If no sector transfer algorithm is specified, or if multiple sector transfer algorithms are specified, the default algorithm is used.</p> <p>NAFS_RANDOM_DIRTY_SECTOR - Uses the alternative sector transfer algorithm that randomly selects a sector with dirty blocks.</p> <p>NAFS_TRACK_SECTOR_ERASES - Enables tracking the number of sector erases for each sector of a flash volume.</p> <p>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> <p>For more information, see the NAFSinit_volume_cb native file system API function in the online help.</p> |
| BSP_NATIVE_FS_FLASHO_COMPACTING_THRESHOLD | <p>If the BSP_NATIVE_FS_FLASHO_OPTIONS constant includes NAFS_BACKGROUND_COMPACTING, 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 NAFSinit_volume_cb native file system API function in the online help.</p> |

NET+OS BSP for ARM9

C H A P T E R 3

This chapter describes how to create a platform for your customized hardware using the NET+OS board support package (BSP) for ARM9-based platforms such as the NS9750 and NS9360.

[illegible]

The board support package (BSP) contains the drivers, board-specific software, and a customizable directory for each supported platform. When you port a new platform to NET+OS 6.3, you typically need to modify the platform directory. If you are using a standard development kit, you can use one of the existing platforms with no modifications.

This chapter describes the overall structure of the NET+OS BSP, how to add in a new platform, and how to debug a new platform.

This table shows the list of supported platforms provided with NET+OS 6.3. If you are adding a new platform to NET+OS, start with a platform that is similar to yours.

| Platform name | CPU type | Description |
|---------------|----------|--------------------------|
| ns9360_a | ARM9 | NS9360 development board |
| ns9750_a | ARM9 | NS9750 development board |

For a description of your platform, see the jumpers and components guide for your development board.

.....

This section describes the power-up and initialization of NET+OS. In general, you do not need to modify the initialization code. Instructions about how to modify customizable parameters on your board are provided in the next section.

Initializing hardware

The hardware initialization code is contained in `src/bsp/init/arm9` for ARM9-based CPUs. The `main()` routine is `src/bsp/common/main.c`.

Initialization sequence

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

- 1 Determine whether the application is booting from SPI:
 - If the application is booting from SPI, the initialization code sets a flag that is read later. This skips over the code that initializes the memory controller, because this is already done during the SPI boot.
 - If the application is not booting from SPI, the initialization code initializes the memory controller so the application can run from SDRAM.
- 2 Take the BBUS out of reset.
- 3 Test a section of RAM that will be used as a stack for the rest of the initialization code.
- 4 Jump to the `nccInit` 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 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.

C library startup

After hardware initialization, the C library `START` routine is called by the `Reset_Handler`, which is located in the `INIT.arm` file. The size of the stack for the C library is specified in the `customize.ldr` customizable file. The default stack size for the C library is 12K. If you are not using C++, you can reduce this size to 8K. The `main()` routine is located in `src/bsp/common/main.c`.

The `main()` routine must perform these steps:

- 1 If the power-on self-test (POST) is enabled, execute it.
- 2 Set up the vector table.
- 3 Enable the Memory Management Unit (MMU).
- 4 Call `NABoardInit` (described in the next section).
- 5 Perform the first level device driver initialization.
This step performs low-level device driver initialization before the OS is loaded.
- 6 If C++ is enabled, initialize the C++ libraries.
- 7 Start ThreadX.

NABoardInit

This routine completes the hardware initialization that was started in `INIT.arm`.

The `NABoardInit` routine must do these steps:

- 1 Read the chip revision and stores it in `g_NAChipRevision`.
- 2 Initialize the low level flash interface.
- 3 Set up non-volatile random access memory (NVRAM).

ROM bootloader

The NET+OS ROM bootloader is a small program that is programmed into ROM. The application image is stored in flash in a compressed format. At startup, the bootloader decompresses it to RAM, and executes it from RAM.

The advantage of using the ROM bootloader is twofold:

- Less flash memory is required because the application image is compressed.
- Applications generally run faster from SDRAM.

The ROM bootloader is built as part of the BSP. The ROM image for the bootloader is contained in the `platforms` directory and is called `rom.bin`. When you run from the debugger, the bootloader is not use.

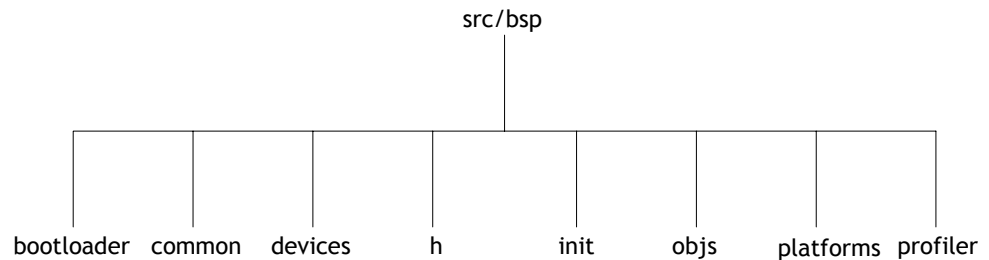
For details about the ROM bootloader, see Appendix D, “Customizing the ROM Bootloader.”

BSP tree structure

These sections describe and illustrate the BSP tree structure.

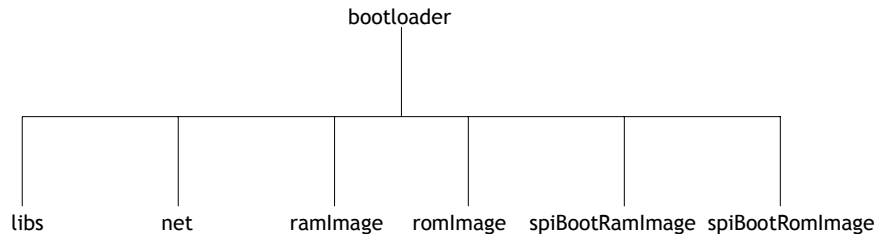
Top-level directory

The NET+OS BSP is located in the `src/bsp` directory. The top level directory, shown next, contains the `Makefile` for the BSP and the `Makefile` for the `bootloader`:



bootloader subdirectory

The `bootloader` subdirectory contains the source code for the SPI and ROM-based bootloaders. This figure shows the `bootloader` subdirectory:



The `bootloader` has two parts: the ROM image and the RAM image.

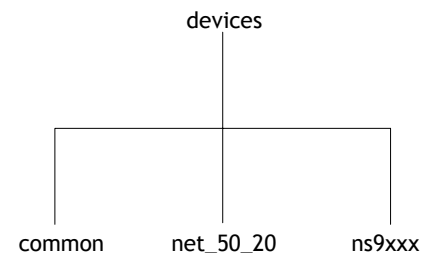
Because the `bootloader` size is kept to less than 64K, the `libs` directory contains the libraries that are linked into the bootloader. The bootloader does not link in the standard NET+OS libraries.

The `bootloader` directory has six subdirectories. This table lists the subdirectories and their contents:

| Subdirectory | Contents |
|--|---|
| <code>libs</code> | Libraries that are specific to the <code>bootloader</code> |
| <code>net</code> | Network-related code for the BSP |
| <code>ramImage</code> | The code and Makefile for the portion of the <code>bootloader</code> that runs from RAM |
| <code>romImage</code> | The Makefile and code for the portion of the <code>bootloader</code> that runs from ROM |
| <code>spiBootRamlImage</code> and <code>spiBootRomImage</code> | The SPI bootloader |

devices directory

The `devices` directory, which contains the NET+OS device drivers, is shown here:

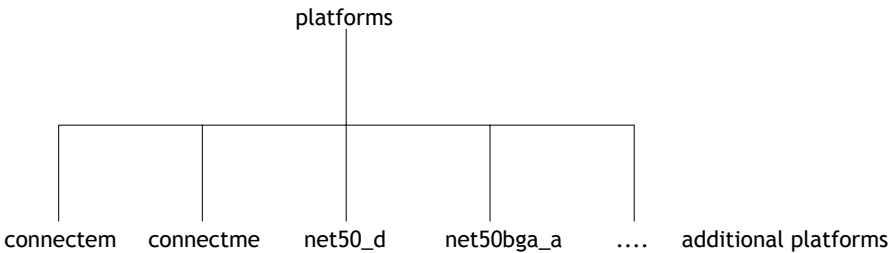


The device drivers are separated into three directories:

- `common` - Contains the device drivers that are common to all processors, such as serial and Ethernet
- `net_50_20` - Contains the drivers for the NS7520 and the NET+50
- `ns9xxx` - Contains the drivers for the NS9360 and NS9750

platforms directory

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



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

Customizing the BSP for application hardware



This section describes how to customize the NET+OS board support package (BSP) for your application hardware. In addition, this section provides general information about the BSP and presents the tasks for porting the BSP to a new hardware platform.

This table lists and briefly describes the basic tasks for porting the BSP to your application hardware.

You may find it helpful to print this table and use it as a checklist as you port the BSP.

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

Follow the reference design

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

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

Verify the features your hardware supports

Make sure your hardware supports these features:

- **Flash at CS1.** The NS9750 and the NS9360 boot from this flash on powerup.
- **RAM (32-bit wide) at CS4.** If you are using multiple chip selects for SDRAM, you must put the largest SDRAM on CS4. CS4 is mapped to address 0 after the Memory Controller is enabled.

The BSP autodetects and configures additional SDRAM memory on the other chip selects.

- **JTAG port.** This port, which allows you to debug the hardware and software, is essential for bringing up a new board.
- **Extra serial port.** This port is used to display standard out messages for debugging. You can easily communicate diagnostic information to the debugging engineer using the standard I/O `printf`.
- **Enough RAM to run your entire application, even if your product runs from ROM.** Running an application from RAM greatly simplifies debugging.

Task 1: Purchase and assign Ethernet MAC addresses

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

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

Task 2: Create a new platform subdirectory

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

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

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

► To create the new platform subdirectory:

- 1 Determine which development board platform is closest to your application hardware.
- 2 Copy the platform's subdirectory and all its contents to your `platform` directory:

```
cd src/bsp/platforms/my_platform
```

You modify the relevant files in the directory to interoperate with your hardware.

Be aware that it is unusual to modify any code outside the platform tree unless you are adding your own drivers or modifying existing devices.

If you believe you must modify files outside the platform tree, contact Digi technical support for confirmation.

Task 3: Building and modifying the BSP Makefile

The next step is to build the BSP.

► **To build the BSP:**

- 1 Double-click the X-Tools icon on your desktop.
- 2 Change to the directory in which you installed NET+OS.
- 3 Add these lines to the `Makefile.inc` file:

```
ifeq "$(PLATFORM)" "my_platform"
CHIP=NS9360
PROCESSOR=arm9
endif
```

- If you are using a `ns9750_a`, enter:

```
CHIP=NS9750
ifeq '$(PLATFORM)' 'my_platform'
CROSS_DEFS_PROCESSOR+= -DNS9750
endif
```

- If you are using a `ns9360_a`, enter:

```
CHIP=NS9360
ifeq '$(PLATFORM)' 'my_platform'
CROSS_DEFS_PROCESSOR+= -DNS360
endif
```

- 4 Do one:

- *Either* enter this command:

```
make PLATFORM=my_platform
```

- *Or* enter these commands:

```
export PLATFORM=my_platform
make
```

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

Task 4: Modify the linker scripts

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

Constants you may need to change

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

| Constant | Description |
|--------------------------|---|
| RAM_SIZE | The size of the RAM part on the board. The linker generates an error if the application is too large to fit in RAM. |
| FLASH_SIZE | The size of the flash part on the board. The linker generates an error if a ROM-based application is too large to fit in ROM. |
| FLASH_START | The starting address of flash. For the NS9750 and NS9360 processors, this address is typically 0x50000000. |
| RAM_START | The starting address of RAM. |
| FILE_SYSTEM_SIZE | The number of bytes to be allocated for the file system in flash. |
| BOOTLOADER_SIZE_IN_FLASH | The amount of flash ROM to be reserved for the bootloader. You also can use this constant to calculate where the application image starts in flash. The bootloader shipped with NET+OS fits into one sector of flash that is typically 64 K. It is important that this is large enough to fit the bootloader ROM image, which is in the <code>src/bsp/platforms/my_platform/rom.bin</code> directory. |

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

Bootloader considerations

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

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

You must set the `ramAddress` field to the value of `BOOTLOADER_CODE_START` in the `customize.ldr` file.

Task 5: Modify BSP configuration files

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

The next sections describe the files you must modify to support your hardware. You may find it helpful to review the Memory Controller information in the hardware reference for the processor you are using.

sysclock.h file

The `sysclock.h` file defines the value for the external oscillator or crystal that supplies the input frequency defined in the `sysclock.h` *platforms* file. This line defines the input frequency for the ns9750 platform:

```
#define NA_ARM9_INPUT_FREQUENCY 398131200
```

The value 398131200 is the input frequency to the NS9750 development boards and 29491200 for the NS9360 development boards. If your input frequency is different, you must modify this value.

bsp.c file

The `bsp.c` file contains tables you must update:

- **Static memory table.** The `MCStaticMemoryTable` array in the `bsp.c` file in the *platforms* directory holds the timing settings for the SRAM (flash) memory parts. The values in the table correspond to the SRAM register settings for the NS9750/NS9360 memory controller.

For more information, see the hardware reference for the processor you are using.

The data structure that corresponds to this table is defined in the `bsp.h` header file and described in the online help. The values in the table correspond to the SRAM part supplied on the development board. The online help also has a description of this table.

If you are using a flash part that is different from what's on the standard NS9750/NS9360 development boards, you may need to modify this table.

- **Interrupt tables.** When you change the system interrupt priority, you must update these tables:

- `NABbusPriorityTab` - This array in the `bsp.c` file in the `platforms` directory contains the priority of each interrupt in the Bbus. The `NABbusPriorityTab` allows flexible prioritization for all BBUS interrupts in the NET+ARM 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 in the `bsp.c` file in the `platforms` directory contains the priority of each interrupt in the AHB Bus. The `NAAhbPriorityTab` allows flexible prioritization for all the AHB interrupts in the NET+ARM that drive the ARM processor IRQ.
The table is configured with interrupts of 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.

init_settings.h file

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

The register settings supplied in the NS9750/NS9360 platforms are for the PC133 parts supplied on the development board. The register settings in this file are described in detail the hardware reference. For a description of these settings, see the online help. *It is important you verify that these values are correct for your memory type.*

cs.c file

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

The next table lists the customization hooks in the `cs.c` file, which contains the routines that configure the NET+ARM chip selects to support memory parts. You need to modify the code in these routines to support your application hardware.

Note that CS4 is already programmed in the initialization code with the parameters in `init_settings.h`. CS1 is connected to flash.

| Customization hook | Hardware feature/default values set |
|----------------------------------|---|
| <code>customizeGetRamSize</code> | Returns the total amount of RAM on the system and calls the customizable <code>customizeGetCSSize</code> routine. |
| <code>customizeGetCSSize</code> | Returns the total number of bytes of memory the chip select is configured to support by examining the address mask in the CS mask register. |
| <code>customizeSetupCS0</code> | CS0 has its power up value when this function is called. The table in the <code>bsp.c</code> file is used to set the registers in the Memory Controller. CS0 has an optional SRAM device connected to it. |
| <code>customizeSetupCS1</code> | Sets up CS1 (flash ROM). CS1 contains the flash code, which the processor initially starts executing on bootstrap. CS1 is preconfigured through the use of strapping pins and must always be connected to flash. The size and starting address of flash come from the linker directive file that is created from <code>customize.ldr</code> . |
| <code>customizeSetupCS2</code> | Sets up CS2 (Static Memory). The table in the <code>bsp.c</code> file is used to set the registers in the Memory Controller. |
| <code>customizeSetupCS3</code> | Must set up CS3 (Static Memory). CS3 has its powerup value when this function is called. The table in <code>bsp.c</code> is used to set the registers in the Memory Controller for this chip select. |
| <code>customizeSetupCS4</code> | Called to customize CS4 (SDRAM). CS4 is initially set up in <code>init.s</code> with the parameters from <code>init_settings.h</code> . The size of the RAM on CS4 must be specified in the <code>customize.ldr</code> file. CS4 gets mapped to address zero after the memory controller is enabled. |
| <code>customizeSetupCS5</code> | Must set up CS5 (optional SDRAM) and fill in the size of the amount of RAM detected on this chip select. This is then used to create the memory map. |
| <code>customizeSetupCS6</code> | Must set up CS6 (optional SDRAM) and fill in the size of the amount of RAM detected on this chip select. This is then used to create the memory map. |
| <code>customizeSetupCS7</code> | Must set up CS7 (RAM) and fill in the size of the amount of RAM detected on this chip select. This is then used to create the memory map. |

| Customization hook | Hardware feature/default values set |
|----------------------------------|---|
| <code>customizeSetupMMCR</code> | <p>Sets up the memory management control register (MMCR), which controls the SDRAM refresh timing. This routine sets up the MMCR for the NET+OS development board. The refresh rate is calculated from the <code>BSP_MPMC_REFRESH_RATE</code> define in <code>cs.c</code>.</p> <p>You need to adjust this value to equal the refresh rate of the SDRAM part you are using. A default refresh rate already has been set up, but you may want to optimize this value.</p> |
| <code>customizeGetRamSize</code> | Returns the total amount of RAM on the system and calls the customizable <code>customizeGetCSSize</code> routine. |

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

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

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

gpio.h file

The NS9750 has 50 pins and the NS9360 has 73 GPIO pins that are multiplexed with functions that include GPIO functionality. You can quickly configure these pins using the definitions in the `gpio.h` file. The multiplexed functions include serial, LCD, Timers, DMA, 1284, USB, Ethernet, external IRQs, and GPIO.

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

For information about how pins are multiplexed, see the `gpio.h` file and the hardware reference for the processor you are using. The `gpiomux_def.h` public header contains definitions used by the `gpio.h` file.

For a detailed description of the GPIO customization, see the online help.

mii.c file

The Ethernet PHY driver is located in the `mii.c` file in the `platforms` directory. If your hardware does not use a supported PHY, you must modify the driver to support it. For more information about supported Ethernet PHYs, see either Chapter 7, “Hardware Dependencies for ARM7-based Platforms” or Chapter 8, “Hardware Dependencies for ARM9-based Platforms.”

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

customizeCache.c file

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

You must update this table if:

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

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

By default, NET+OS uses write back cache. If you are writing NET+OS drivers, you need to make sure that they can handle cache coherency.

pci.c file

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

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

customizeButtons.c file

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

customizeLed.c file

The `customizeLed.c` file contains the `NALedTable` table global data structure, 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 section 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.

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

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

You must configure the `bsp.h` file to enable the drivers that you want to run with your application. The default configuration works with a development board. Note that drivers that use the same GPIO pins cannot properly function at the same time. For details on all the defines in `bsp.h`, see the online help.

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

USB device controller

The BSP is configured by default to support the USB device. You must modify the development board to support this interface. For information about modifying the board, see the hardware reference for the processor you are using.

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

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

To test USB device functionality on the NS9750 board, use the instructions in the development board’s jumpers and components guide. To modify the development board, see the `ReadMe` file in the `nausbdevapp` example. The USB device example uses GPIO pin 17 to set up plug-and-play (pnp) functionality. Your system uses this pin to detect whether the device is active and ready to receive commands.

To test USB device functionality on the NS9360 board, see the `ReadMe` file in the `nausbdevapp` example.

1284 controller

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

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

Edit the `1284.h` file in your `platforms` directory to set the number and size of the receive and transmit buffers used by the 1284 driver. The default values usually are sufficient unless you want to tune your application for performance or memory usage.

In addition, you must modify the development board to support the 1284 controller. For information about modifying the board to support this interface, see the hardware reference for the processor you are using.

To configure the GPIO MUX to support the parallel port, set `BSP_GPIO_MUX_1284` to `BSP_GPIO_USE_PRIMARY_INTERFACE`. Be aware that those pins are shared by several other functions, and you will need to disable those functions in `gpio.h`. If conflicts occur, the BSP `Makefile` will output compiler errors that tell you which functions you need to disable.

I2C controller

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

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

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

LCD controller

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

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

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

PCI driver

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

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

Serial ports

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

To set a serial port to a mode other than those already set up by the standard NET+OS release (such as SPI), modify the `gpio.h` file to ensure that correct GPIO pins are set to the correct value. Set `BSP_SERIAL_PORT_X` to one of these values in `bsp.h`:

- `BSP_SERIAL_NO_DRIVER`
- `BSP_SERIAL_UART_DRIVER`
- `BSP_SERIAL_SPI_DRIVER`
- `BSP_SERIAL_SPI_SLAVE_DRIVER`

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

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

RTC

The BSP supports a real time clock on NS9360 board platforms:

- To enable the real time clock, set the `BSP_INCLUDE_RTC_DRIVER` define to `TRUE`.
- To disable the RTC set the `BSP_INCLUDE_RTC_DRIVER` define to `FALSE`.

Task 7: Modify the format of BSP arguments in NVRAM

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

You must modify these customization hooks to support your application:

| Customization hook | Description |
|---------------------------------------|---|
| <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.</p> <p>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.</p> <p>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> |

| Customization hook | Description |
|--|--|
| <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. It is not used by the API libraries or in any part of the BSP except the dialog.</p> <p>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> |
| <code>customizeUseDefaultParameters</code> | <p>Determines default configuration values and returns them in a buffer.</p> <p>The default implementation determines the default values through constants set in <code>appconf.h</code>. You must modify this routine to support your application.</p> |
| <code>customizeReadDevBoardParams</code> | <p>Reads the configuration from NVRAM into a buffer. You must modify this routine to support your application.</p> |
| <code>customizeWriteDevBoardParams</code> | <p>Writes the configuration to NVRAM.</p> <p>The default implementation accepts the current configuration as a buffer and writes the buffer into NVRAM.</p> |
| <code>customizeGetIPParameters</code> | <p>Reads the IP-related configuration values from NVRAM.</p> |
| <code>customizeSaveIPParameters</code> | <p>Writes the IP-related configuration values to NVRAM.</p> |

Task 8: Modify error and exception handlers

The `errhndlr.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.h`, you can configure the default error handler to either:

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

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

Exception handler

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

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

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

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

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

For details about error and exception handlers, see Appendix F, “Processor Modes and Exceptions.”

Task 9: Verify the debugger initialization files

This section provides instructions for both the Raven and the MAJIC debuggers.

Using the MAJIC/MAJICO probe

When you use the EPI MAJIC/MAJICO probe, you must initialize hardware registers on the board that the BSP ROM startup code normally sets up. Debugger initialization scripts are set up as part of the installation procedure for NET+OS 6.3. The scripts contain commands that the debugger executes before the application is downloaded and executed.

The MDI server is the interface between the gdb debugger and the MAJIC. The debugger initialization script, `ns9xxx.cmd`, is located in the directory from which the MDI server is executed.

During the installation procedure, you are prompted for the name of this directory. The MDI server reads this script when you start to download code to the board using gdb.

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

| File name | Contents |
|---------------------------|--|
| <code>startice.cmd</code> | The JTAG settings and reads in the <code>ns9xxx_cmd</code> file to initialize the target board |
| <code>ns9xxx.cmd</code> | The sequence of commands to initialize SDRAM |
| <code>epimdi.cfg</code> | MAJIC settings, including the network parameters |

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

If you are using a different type of SDRAM, you must modify the settings in the `ns9xxx.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.

Raven debugger

When you use the Raven debugger, you must initialize hardware registers on the board that the BSP ROM startup code would normally set up. The scripts contain commands that the debugger executes before the application is downloaded and executed. The debugger initialization scripts are contained in the directory `debugger` files and are labeled `gdbns9xxx.raven`. The debugger reads these scripts when you start to download code to the board using gdb.

If you are using a different type of SDRAM, you must modify the settings in these scripts. The debugger scripts program the registers in the memory controller. For a detailed description of these registers, see the hardware reference for the processor you are using.

Task 10: 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 code.

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

The next sections provide Instructions the MAJIC/MAJICO probes and the Raven debugger.

Preparing to debug the initialization code using the MAJIC/MAJICO

The instructions in this section apply to both the MAJIC and the MAJICO debuggers.

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

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

```
ping IP_ADDR
```

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

If you do not get a response, verify that the Ethernet cable is connected to the MAJIC and that the status light on the MAJIC is green.

- 2 Rebuild the BSP with your changes:

- a Change to the bsp directory:

```
cd src/bsp
```

- b Enter this command:

```
make PLATFORM=my_platform
```

where *my_platform* is the name of your platform.

- 3 Disable the POST by setting the `APP_POST` constant in the `root.c` file to 0.

Carefully review all the settings in the `appconf.h` file. Make sure that `stdio` is directed to the correct serial port. The default is `com/0`.

- 4 Build the application:
 - a Copy the template application, which is located in this directory:
`src/apps/template`
 - b In the `src/apps/template32b` directory, enter:
`make PLATFORM=my_platform clean`
`make PLATFORM=my_platform all`
- 5 Start the MDI server by clicking the MDI server icon.
- 6 Load the application with either `gdb` or `gdbtk`.
 - To set up a `gdbinit` file in the `32b` directory for the template application, enter:
`make gdbinit`
 - To start the debugger, enter:
`gdbtk -se image.elf`
- 7 Set up the debugger to view assembler instructions, and then step one instruction. This leaves the program counter (PC) at the beginning of the startup code.
- 8 Verify that the debugger initialization file has configured the application board such that:
 - The Chip Select registers for ROM and RAM are set up to support the parts and memory map.
 - You can read and write RAM on your application board.
- 9 Debug the initialization code by stepping through it, as described in the next section.

Preparing to debug the initialization code using the Raven

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

- 1 In 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` and then enter `make` to build the BSP.)

- 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. Make sure that `stdio` is directed to the correct serial port. The default is `/com/0`.
- 4 Build the application:
 - a Copy the template application, which is located in this directory:
`src/apps/template`
 - b In the `src/apps/template/32b` directory, enter:

```
make clean
make all
```
- 5 Copy the debugger file that corresponds to your processor into the `src/app/template/32b` directory.
 When `gdb` starts, it reads this file.
 For example, if you are using the NS9750, copy the `debugger_files/gdbtk` `-se image.elf` file into `src/apptemplate 32b` and rename it `gdbinit`.
- 6 Click the `ocdremote -C ARM926EJ-S` icon.
- 7 From the `/src/apps/template/32b` directory, enter this command:
`gdbtk -se image.elf`
- 8 To load your image, from the `gdb` console window, enter:
`lo image.elf`
- 9 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.
- 10 Verify that the debugger initialization file has configured the application board such that:
 - The Chip Select registers for ROM and RAM are set up to support the parts and memory map.
 - You can read and write RAM on your application board.
- 11 Debug the initialization code by stepping through it, as described next.

Debugging the initialization code

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

Note: 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 the `Reset_Handler` routine 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, you can 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, which is located in `src/bsp/init/arm9`, provides some low-level initialization routines for flash and NVRAM. Step through the initialization code in the `nambrd.c` file to verify that the NVRAM APIs are initialized to support the NVRAM on your application hardware. You can configure the board to use a flash sector as NVRAM.

Debug the Ethernet driver startup► **To debug the Ethernet driver startup:**

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

Task 11: Modify the startup dialog

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

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

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

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

Task 12: Modify the POST

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

The POST routines that ship with NET+OS test the NS9750 /NS9360 processor. You may want to create other POST routines that test additional hardware on your board.

Task 13: Modify the ACE

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

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

aceCallbacks.c

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

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

aceParams.c

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

- **The `dhcp_desired_params` array.** Contains a list of the Dynamic Host Configuration Protocol (DHCP) options you want the client to request from the server. Add any other DHCP options you want the client to request from the server.
- **`NADefaultEthInterfaceConfig`.** Contains the configuration that ACE uses if none is stored in NVRAM. This configuration controls which protocols are used to get an IP address and the options used with them. The default configuration uses all protocols to get an IP address. Customize this configuration as needed.

For details about these functions, see the online help.

Other BSP customizing

This section describes additional BSP customizing you may want to do.

BSP_NVRAM_DRIVER

The `BSP_NVRAM_DRIVER` constant in `bsp.h` defines the non-volatile memory type used to store the configuration information. Here are the settings:

| Constant | Description |
|-------------------------------|--|
| <code>BSP_NVRAM_DRIVER</code> | <p>This constant in <code>bsp.h</code> defines the non-volatile memory type used to store the configuration information. Here are the settings:</p> <ul style="list-style-type: none">■ <code>BSP_NVRAM_NONE</code> - No NVRAM driver is to be built.■ <code>BSP_NVRAM_LAST_FLASH_SECTOR</code> - The last sector of flash is to be used for NVRAM.■ <code>BSP_NVRAM_SEEPROM</code> - The serial EEPROM driver is to be built.■ <code>BSP_NVRAM_SEEPROM_WITH_SEMAPHORES</code> - The serial EEPROM driver with semaphore protection is to be built.■ <code>BSP_NVRAM_LAST_FLASH_SECTOR</code> - The last sector of serial flash is to be used for NVRAM. |

TCP/IP stack

The TCP/IP stack, which is started as part of the BSP initialization process, is the software module that handles networking functionality. These functions and constants are used for configuring the TCP/IP stack:

| Function or constant | Description |
|--|--|
| <code>BSP_LOW_INTERRUPT_LATENCY</code> | <p>This constant in <code>bsp.h</code> determines how the TCP/IP stack implements its critical section:</p> <ul style="list-style-type: none">■ To use a semaphore for the TCP/IP critical section, set <code>BSP_LOW_INTERRUPT_LATENCY</code> to <code>TRUE</code>.■ To disable processor interrupts to implement the TCP/IP critical section, set <code>BSP_LOW_INTERRUPT_LATENCY</code> to <code>FALSE</code>. |
| <code>BSP_ENABLE_FAST_IP</code> | <p>This constant in <code>bsp.h</code> enables Fast IP:</p> <ul style="list-style-type: none">■ To enable Fast IP, set <code>BSP_ENABLE_FAST_IP</code> to <code>TRUE</code>.■ To disable Fast IP, set <code>BSP_ENABLE_FAST_IP</code> to <code>FALSE</code>. <p>Fast IP is not supported for low interrupt latency.</p> |

| Function or constant | Description |
|------------------------------------|---|
| BSP_WAIT_FOR_IP_CONFIG | <p>This constant in <code>bsp.h</code> determines whether the BSP waits for the stack to be configured before starting the application by calling the <code>applicationStart()</code> function. Previous versions of NET+OS always waited for the stack to be configured.</p> <p>Your application should not use any network resources until the stack has been configured by setting an IP address on at least one interface. You can use the <code>customizeAceGetInterfaceAddrInfo()</code> function to determine whether an IP address has been assigned to an interface.</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>. <p>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>.</p> |
| BSP_ENABLE_ADDR_CONFLICT_DETECTION | <p>This constant in <code>bsp.h</code> enables IP address conflict detection, during initial IP address configuration.</p> <p>If <code>BSP_ENABLE_ADDR_CONFLICT_DETECTION</code> is defined to <code>TRUE</code>, the ACE subsystem sends ARP probes to detect IP address conflict for BOOTP, RARP, Ping ARP, and static IP address protocols. IP address conflict detection also must be enabled on a network device. You can retrieve the device configuration for IP address conflict detection with the <code>NAGetAddrConflictData</code> function.</p> |
| NAIpSetKaInterval | <p>This function (<code>naip_global.c</code> overrides the default value for the TCP keepalive interval, which by default is 2 hours (7200 seconds).</p> <p>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 socket by the <code>IP_TTL</code> socket option.</p> |
| NAIpSetTcpMsl | <p>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>.</p> |
| APP_NET_HEAP_SIZE | <p>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.</p> |

File system

The BSP can be configured to interface the C library file I/O functions to the file systems. NET+OS 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 | Set this constant in <code>bsp.h</code> to <code>TRUE</code> to include the native file system in the C library and create a RAM and flash volume as part of the BSP initialization process. |
| BSP_NATIVE_FS_MAX_INODE_BLOCK_LIMIT | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the percentage of the maximum number of inode blocks that can be allocated to store inodes for a volume. This constant allows specifying the upper limit of the number of blocks reserved to store inodes. Valid values are from 1 to 100. For more information, see the <code>NAFSinit_volume_cb</code> native file system API. |
| BSP_NATIVE_FS_MAX_OPEN_DIRS | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the maximum number of open directories that the file system will track. A directory is considered open if the directory has open files. Valid values are from 1 to 64. For more information, see the <code>NAFSinit_volume_cb</code> native file system API. |
| BSP_NATIVE_FS_MAX_OPEN_FILES_PER_DIR | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the maximum number of open files per directory that the file system will track. Valid values are from 1 to 64. For more information, see the <code>NAFSinit_volume_cb</code> native file system API in the online help. |
| BSP_NATIVE_FS_BLOCK_SIZE | When the BSP creates a native file system volume, this constant in <code>bsp.h</code> specifies the block size used for the volume. Valid values are: <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> |

| Constant | Description |
|---|--|
| BSP_NATIVE_FS_RAMO_VOLUME_SIZE | When the BSP creates the native file system RAM volume, this constant specifies the size of the RAM volume in bytes. |
| BSP_NATIVE_FS_FLASHO_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 the following:</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 no sector transfer algorithm is specified or if multiple sector transfer algorithms are specified, 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 NAFSinit_volume_cb native file system API function.</p> |
| BSP_NATIVE_FS_FLASHO_COMPACTING_THRESHOLD | <p>If the BSP_NATIVE_FS_FLASHO_OPTIONS constant includes NAFS_BACKGROUND_COMPACTING, 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 NAFSinit_volume_cb native file system API in the online help.</p> |
| BSP_INCLUDE_FAT_FILESYSTEM_FOR_CLIBRARY | Set this constant to TRUE to include the FAT file system in C library. The FAT file system is supported only on the NS9360 and NS9750 platforms. |

Linker Files

C H A P T E R 4

This chapter describes the linker files that are provided for sample projects and the corresponding memory map.

Overview

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

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

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

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

Linker files provided for sample projects

Linker files are provided in `src/bsp/platforms/my_platform` and are used to link the sample applications. Most projects use the `image.ldr` and `rom.ldr` linker files to create applications that execute from RAM or ROM, respectively. These linker scripts are generated when applications are built by the BSP Makefile.

The source files for the linker scripts are stored in the `C:/NETOS63_GNU/bsp/init/arm9` and `C:/NETOS63_GNU/bsp/init/arm7` directories. When the BSP is built, these files, along with `customize.ldr`, are used to generate the linker files in the `platforms` directory which is then used by the application Makefile.

These linker files are provided:

| Linker file | Description |
|----------------------------|---|
| <code>customize.ldr</code> | The customization file for linker scripts |
| <code>image.ldr</code> | Generates an executable file for debugging and for the <code>image.bin</code> image |
| <code>rom.ldr</code> | Generates an executable file for the <code>rom.bin</code> image |

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

Basic GNU Tools section of the linker files

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

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

NET+OS section of the linker files

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

| Section | Description |
|------------------|--|
| gnu_initdata | Stores jumper and button settings read at startup. |
| heap | Heap; grows upward. |
| gnu_ncc_initdata | Stores NET+OS settings determined at startup. |
| stack | System stack; grows downward. |
| netosstack | Stack for each processing mode; grows downward. |
| free_mem | Used for the kernel to create the timer thread and root thread. Do not use this section for any other purpose. |
| ttb | Stores the mmuTTB table at the end of RAM. Initially is set up by the bootloader. Note that if you change <code>ttb_size</code> , you also must change <code>SECOND_LEVEL_TABLE_SIZE</code> in the mmu utility <i>Do not overwrite this table.</i> |

The ThreadX library is hard-coded to call `tx_application_define` (void *first_unused_memory) after the kernel has been loaded and just before the kernel scheduler starts. The `free_mem` address is passed to this function, which creates the root thread that is responsible for starting NET+OS and the IP stack. Do not pass any other address to create the root thread. The `first_unused_memory` argument points to a global variable that the kernel sets up.

Address mapping (ARM9 only)

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

The NS9360_a development board currently has:

- 16 MB SDRAM on CS4, mapped at 0X00000000
- 2 MB flash on CS1, mapped at 0X50000000

In NET+OS, the `netos_63_gnu/src/bsp/my_platform/customizeCache.c` file contains the table used to set up the MMU translation tables. A sample of the table is shown next. In this table:

- The starting and ending virtual addresses are the address software can read.
- The cache mode defines whether the memory region is buffered, write through, or write back.
- The user access defines the access permissions for the region. If an access violation occurs, an exception is raised by the CPU. The user access allows the MMU to set up address ranges that are invalid for the software to write to, which is a useful in debugging rogue pointers.

| Starting virtual address | Ending virtual address | Page size | Cache mode | User access | Physical address |
|--------------------------|------------------------|-----------|----------------|-------------|-----------------------|
| 0x00000000 | 0x00FFFFFF | SIZE_1M | MMU_WRITE_BACK | RW | 0x00000000-0x00FFFFFF |
| 0xA0000000 | 0xA0FFFFFF | SIZE_1M | MMU_BUFFERED | RW | 0xA0000000-0xA0FFFFFF |
| 0xC0000000 | 0xC0FFFFFF | SIZE_1M | MMU_BUFFERED | RW | 0x000000-0x00FFFFFF |

This is what the rows in the table specify:

- **Top row.** Specifies that the virtual address `0x00000000-0x00FFFFFF` is write back, is readable and writable and maps to the physical memory at `0x00000000-0x00FFFFFF`.
- **Second row.** Specifies that the address `0xA0000000` to `0xA0FFFFFF` is set up to be a buffered region of memory with read/write access; this is the section of memory used for PCI I/O. The possible values for the cache mode are shown in the next table.
- **Third row.** Shows that the address range `0xC0000000-0xC0FFFFFF` is mapped to the physical address range `0x00000000-0x00FFFFFF` and is buffered but not cached. The `0xC0000000` is the address range that the software can use to access memory as non-cached; all reads and writes go directly to main memory.

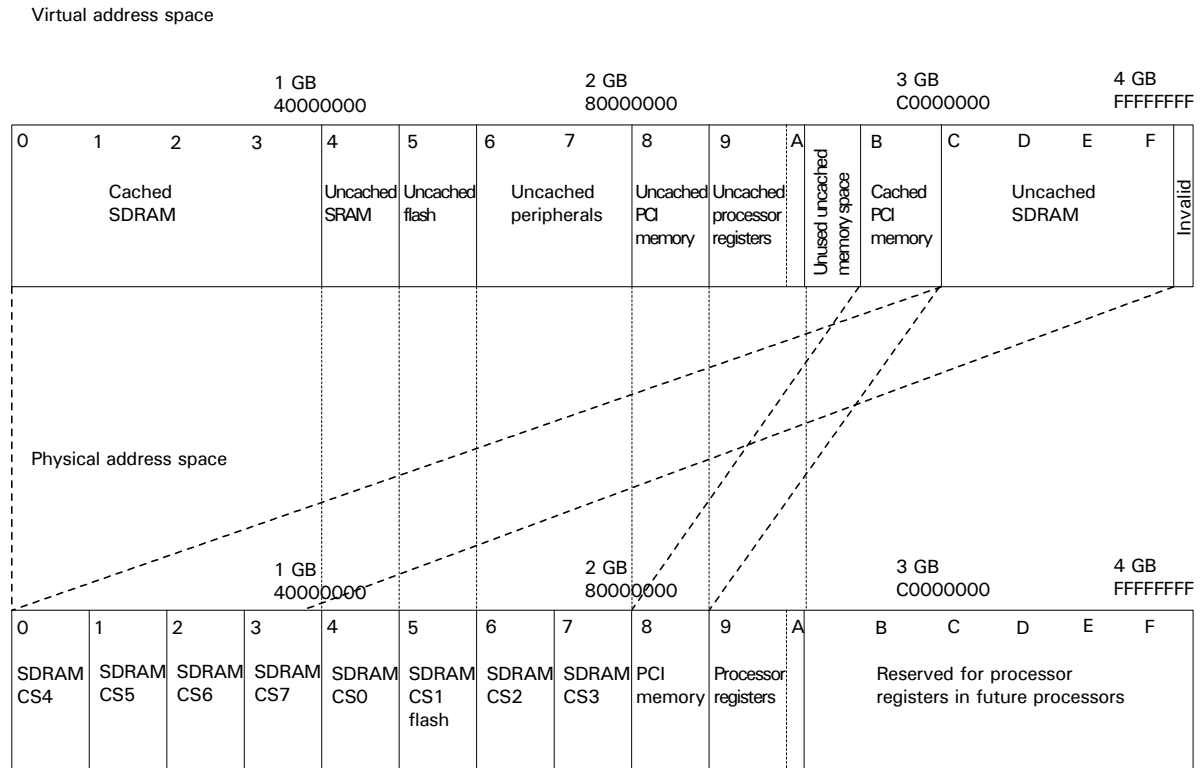
For more information, see the online help.

The ARM processor has a 16-word write buffer that performs burst writes to memory to increase efficiency. NET+OS sets up the non-cached regions as bufferable; this does not cause any coherency problems because writes are always performed through the write buffer. So if you are using the DMA, by the time the F (full) bit is set the data written before it would have been flushed from the write buffer. The cache flush routines are described in the MMU section of the online help.

| Cache mode | Description |
|-------------------|--|
| MMU_NONBUFFERED | Disable all caching and buffering. |
| MMU_BUFFERED | Disable caching, but allow writes to be buffered. |
| MMU_WRITE_THROUGH | Cache reads, but do not cache writes. Allow writes to be buffered. |
| MMU_WRITE_BACK | Cache both reads and writes, and allow writes to be buffered. |

NET+OS memory map (ARM9 only)

The NET+OS memory map for the ARM9 based development boards is shown next. Note that the NS9360 does not have 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 gigabyte 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 GB-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 4GB 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.

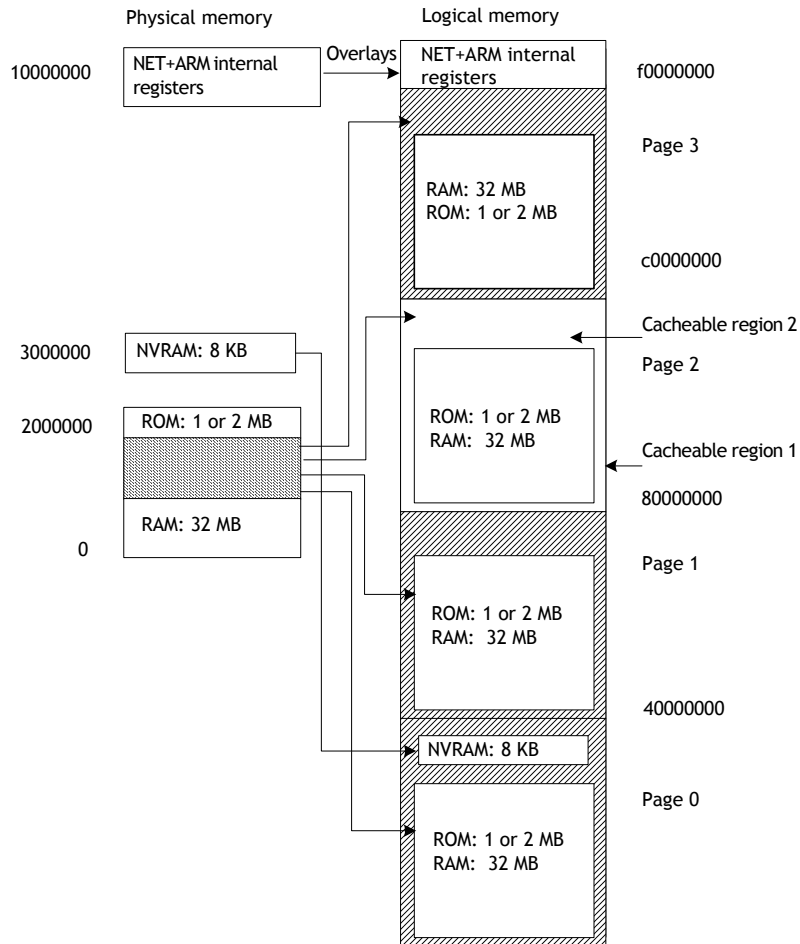
All applications use the 0GB-1GB range of addresses, which is set up as write-back cache; NET+OS drivers typically use the 3GB-4GB to store DMA buffer descriptors that should not be cached. You usually need to access the uncached region only if you are writing drivers that use DMA; typical applications never need to use this region.

Memory aliasing in NET+OS (ARM7 only)

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. Code is executed from this area of the address map to improve performance. NET+OS uses uncached areas for general data storage.


The next figure shows the NET+OS memory map with cache enabled. In the figure:

- 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 CS2) at addresses 0x0 through 0x1ffffff.
- Either 1 or 2 MB of flash ROM on CS0 begin 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 be accessed at |
|------------------|-----------|-------------------------------------|
| 0x100 | RAM | 0x4000100, 0x8000100, and 0xC000100 |
| 0x20000100 | Flash ROM | 0x6000100, 0xA000100, and 0xE000100 |
| 0x3000100 | NVRAM | Only at 0x3000100 |



Adding Flash



C H A P T E R 5

This chapter describes how to update flash memory.

Overview

NET+OS includes application program interface (API) functions for reading, writing, and erasing flash memory. The internals of the flash memory API rely on `flash_id_table` in the `naflash.c` file (located in `C:/NETOS62_GNU/src/flash`) to define the known flash parts. The flash API is guaranteed to function only with parts that are defined in the `flash_id_table`. If the part is not recognized, you need to update the `flash_id_table`.

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

NET+OS 6.3 supports these flash ROM parts:

| Manufacturer | Part number |
|--------------|-------------|
| AMD | AM29F800B |
| AMD | AM29DL323DB |
| AMD | AM29LV16 |
| Atmel | AT29C040A |
| Atmel | AT49BV8011 |
| Atmel | AT49BV8011T |
| Atmel | AT49BV1614A |
| Fujits | 29LV800BA |
| Macronix | MX28F4000 |
| Sharp | H28F800SG |
| SST | 28SF040 |
| SST | 9VF800 |
| STM | M29W800AB |
| STM | M29W160DB |
| STM | M29W320DB |

Flash table data structure

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

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

typedef struct
{
    WORD16 mcode;
    WORD16 mcode_addr;
    WORD16 dcode;
    WORD16 dcode_addr;
    WORD16 total_sector_number;
    WORD32 sector_size;
    WORD16 prog_size;
    WORD16 access_time;
    flash_cmd_t *id_enter_cmd;
    WORD16 id_enter_len;
    flash_cmd_t *id_exit_cmd;
    WORD16 id_exit_len;
    flash_cmd_t *erase_cmd;
    WORD16 erase_len;
    flash_cmd_t *write_cmd;
    WORD16 write_len;
    flash_cmd_t *sector_erase_cmd;
    WORD 32 *sector_size_array;
} flash_id_table_t;
```

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

| Data type | Description |
|-----------|----------------|
| WORD8 | Unsigned byte |
| WORD16 | Unsigned short |
| WORD32 | Unsigned long |

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

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

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

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

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

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

```
#define STM_29W800AB 0x0D
```

- 3 In the `naflash.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.
- 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 To rebuild the driver, enter `make PLATFORM` in the `C:/netos62_gnu/src/flash` directory to rebuild the flash library in the top-level directory.

Rebuilding this library rebuilds the flash driver.

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

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

```

sizeof(STM_M29W800AB_flash_erase_cmd) / sizeof(flash_cmd_t),
(flash_cmd_t *)STM_M29W800AB_flash_write_cmd,
sizeof(STM_M29W800AB_flash_write_cmd) / sizeof(flash_cmd_t),
(flash_cmd_t *)STM_M29W800AB_flash_block_erase_cmd,
(WORD32*) STM_M29W800AB_flash_block_size_array
}

```

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

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

Supporting larger flash

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

► To support larger flash configurations:

- 1 Increase these three constants in `flash.h`:
 - `MAX_SECTORS` - The maximum number of flash sectors supported
 - `MAX_SECTOR_SIZE` - The maximum sector size supported
 - `MAX_FLASH_BANKS` - The maximum number of flash banks supported
- 2 To rebuild the flash library in the top-level directory, enter `make`.



Device Drivers



C H A P T E R 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> | <p>Pointer to the device's <code>open</code> routine for the channel. This routine is called whenever an application opens the channel to indicate that a new session is starting.</p> <p>The <i>flags</i> field indicates whether the channel:</p> <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 <code>close</code> routine for the channel. This routine is called at the end of every session. |
| <i>deviceRead</i> | Pointer to the driver's <code>read</code> routine for the channel. |
| <i>deviceWrite</i> | Pointer to the driver's <code>write</code> routine for the channel. |
| <i>deviceIoctl</i> | Pointer to the driver's I/O control routine for the channel. |
| <i>flags</i> | <p>Bit field that indicates which bits are valid in the <i>flags</i> field of an <code>open</code> call to the device.</p> <p>A bit set in this field indicates that the bit also can be set in the driver's <code>open</code> routine.</p> |

Device driver functions

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

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

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

deviceEnter

First-level initialization function for a device table.

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

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

Format

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

Arguments

| Argument | Description |
|----------------|---|
| <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 routine for the device channel.

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

Format

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

Arguments

| Argument | Description |
|----------------|---|
| <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

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

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

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

Format

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

Arguments

| Argument | Description |
|----------------|---|
| <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."

deviceRead

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

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

- 1 Sets `bytesRead` to 0.
- 2 Checks that the arguments are correct and the channel is open.
- 3 Checks for a pending error on the device.
If an error condition is detected, the driver returns an error without transferring any data.
- 4 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 Either sets the channel semaphore or returns `EBUSY` if the semaphore already is set.
- 5 Opens a transmit buffer and fills it with data from the caller's buffer.
- 6 Starts the transmit operation for the transmit buffer.
- 7 *This step applies to blocking mode only.* If an error condition is detected, aborts the transmission and returns an appropriate completion code.
- 8 If there is more data in the caller's buffer, repeats steps 5 through 7 until there is no more data.
- 9 Updates *bytesWritten* to indicate the number of bytes transmitted.
- 10 Releases the channel semaphore.
- 11 Returns a completion code.

Format

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

Arguments

| Argument | Description |
|---------------------|---|
| <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) routines map to the DDI application layer calls as shown in this table:

| DDI routine | DDI application layer call |
|--------------------------|----------------------------|
| <code>deviceOpen</code> | <code>open</code> |
| <code>deviceClose</code> | <code>close</code> |
| <code>deviceIoctl</code> | <code>ioctl</code> |
| <code>deviceRead</code> | <code>read</code> |
| <code>deviceWrite</code> | <code>write</code> |

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

The application return values fall into one of two categories:

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

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

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

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

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

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

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

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

Modifications to Cygwin's standard C library and startup file

The standard C library has been rebuilt to support the NET+OS DDI. A customized version of the startup files and C libraries is in the `C:/NETOS63_GNU/lib/32b/gnu` directory. All the sample applications link to these files instead of to the standard GNU versions.

To use the NET+OS device drivers and the ThreadX kernel, you must make your applications link to these files. For an example, see either of the `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:/NETOS63_GNU/gnusrc` directory.

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

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

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

► To modify the `libc.a` and `crt0.o` files:

- 1 Copy `cygwin/usr/arm-elf/lib/be/libc.a` to the `C:/NETOS61_GNU/gnusrc` directory.
- 2 To open a GNU X-Tools shell, enter this command:
`xtools arm-elf`
- 3 To produce the new `libc.a` and a new `crt0.o` file to support NET+OS I/O devices, change to the `C:/NETOS61_GNU/gnusrc` directory and enter:
`make all`
- 4 Copy `gnusrc/libc.a` and a new `crt0.o` to the `C:/NETOS61_GNU/lib/32b/gnu` directory.

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

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

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.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 | All |
| Serial | Serial | All |
| NVRAM | Non- volatile RAM | All |
| System clock | System clock interface routines | All |
| Timer | Timer | All |
| HDLC | High level data link control | All |
| MMU | Memory Management Unit | All |
| gpio | General purpose I/O | All |
| Parallel | Parallel driver | All |
| I2c | Inter-IC | All |
| LCD | LCD routines | All |
| USB device | USB device | All |
| USB host | USB Host | All |
| PWM | Pulse Width Modulator | NS9360 |
| RTC | Real Time Clock | NS9360 |
| PCI | PCI Bus | NS9750 |
| Ethernet | Ethernet | All |

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.

Hardware Dependencies for ARM7-based Platforms

C H A P T E R 7

This chapter discusses the NET+OS hardware dependencies for platforms that use the NS7520 and NET-50 processors.

Overview

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

- DMA channels
- Ethernet PHY
- ENI controller
- Serial ports
- Software watchdog
- Endianness
- System clock and timers
- Interrupts

The rest of the sections in this chapter describe these 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 <code>bsp/devices/ethernet</code> directory. |
| 2 | Ethernet driver | Moves data from memory to the Ethernet transmitter. |
| 3 through 6 | Parallel ports (NET+50) | For the NET+50 only. Moves data between the parallel port and memory. |
| | External peripherals (NS7250) | For the NS7250 only. Only two channels-either 3 and 5 or 4 and 6-can be configured at one time. |
| 7 and 8 | HDLC/serial/SPI driver | Receives data |
| 9 and 10 | HDLC/serial/SPI driver | Transmits data. |
| 11 through 13 | | Moves data from memory to memory (NS7520 only) |

Ethernet PHY

NET+OS supports PHYs that use the MII interface. The PHY driver, which is implemented in the `mi.c` file, supports these PHYs:

| PHY | Manufacturer |
|---|---------------------|
| FastCat (also known as the 3-volt enable PHY) | Lucent Technologies |
| LXT970 | Level One |
| LXT971A and LXT972A | Intel |
| AM79C874 and AM79C875 | AMD |

You can modify the `mi.c` file to support additional PHYs.

ENI controller

The BSP configures the ENI controller for IEEE 1284 host port mode, which supports four parallel ports.

Serial ports

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

To use the serial peripheral interface (SPI) controller, disable the serial driver, using either of these methods:

- **Recommended method.** Undefine `BSP_INCLUDE_SERIAL_DRIVER1` and `BSP_INCLUDE_SERIAL_DRIVER2` in the `bsp.h` file.
- **Alternate method.** Remove the serial driver entries from the device driver table in the `devices.c` file.

You do not need to disable the serial driver to use the HDLC driver.

Software watchdog

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

The BSP system clock depends on whether you are using an external crystal or an external oscillator. The external `PLLST*` signal indicates the choice. The frequency of the selected source affects the BSP timing.

The PLL setting for the NS7520 is determined by the pull-up and pull-down resistors tied to pins on the NS7520.

The rest of this section describes the constants you need to set for the system clock in the `bsp.h` file.

BSP_CLOCK_SOURCE

The value of `BSP_CLOCK_SOURCE` in `bsp.h` determines the clock source to be used. `BSP_CLOCK_SOURCE` indicates the input to the `SYSCLK` signal multiplexer, which has two possible sources:

- TTL clock input applied to the XTAL1 pin
- Crystal oscillator and PLL circuit

Set `BSP_CLOCK_SOURCE` to either of these:

- `SELECT_THE_XTAL1_INPUT`
- `SELECT_THE_CRYSTAL_OSCILLATOR_INPUT`

XTAL1_FREQUENCY

`XTAL1_FREQUENCY` and `XTAL1_FREQUENCY_20UM` indicate the frequency of the TTL clock input to the XTAL1 pin on the NET-50 and NS7520 platforms, respectively. If `BSP_CLOCK_SOURCE` is set to `SELECT_THE_XTAL1_INPUT`, this value determines the frequency of `SYSCLK`.

CRYSTAL_OSCILLATOR_FREQUENCY

This setting indicates the frequency of the crystal oscillator. If `BSP_CLOCK_SOURCE` is set to `SELECT_THE_CRYSTAL_OSCILLATOR_INPUT`, the crystal oscillator is input to the PLL, and in conjunction with `PLL_CONTROL_REGISTER_N_VALUE`, determines the frequency of the internal `SYSCLK` signal.

PLL Control register setting

This setting indicates the N factor used in the divide-by circuits of the NET+ARM clock generation section. The N factor multiplies or divides clock sources. The value is stored in the `PLLCNT` field in the PLL Control register.

For more information, see the hardware reference for the processor you are using.

The range of values is 0 through 15; the suggested values are based on device type and revision:

- For NET+50-based platforms, the value is determined by entries in the `NA_PLL_TABLE` table in `bsp.h`.
- For NS7520-based platforms, the value is determined by hardware bootstrap settings.

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.h` file. This value, which determines the heartbeat rate, should be between 1 and 1000. A value of 100, for example, provides a heartbeat rate of one tick every ten milliseconds.

Timer 2

The BSP uses Timer 2 to support the parallel driver. If this timer is disabled, or if its frequency is changed, the parallel driver code in the `nammpara.c` file is affected. Timer 2 normally is programmed to have a period of 217 microseconds.

If `BSP_SERIAL_FAST_INTERRUPT` is set in `bsp.h`, Timer 2 is used by the serial driver.

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) | <ul style="list-style-type: none">■ HDLC driver channel 1 receive frame interrupt■ Serial/SPI 1 DMA mode receive interrupt |

| Interrupt level | Use |
|-----------------------------------|--|
| 24 (DMA 8) | <ul style="list-style-type: none"> ■ HDLC driver channel 1 receive frame interrupt ■ Serial/SPI 1 DMA mode receive interrupt |
| 23 (DMA 9) | <ul style="list-style-type: none"> ■ HDLC driver channel 2 receive frame interrupt ■ Serial/SPI 2 receive interrupt |
| 22 (DMA 10) | <ul style="list-style-type: none"> ■ 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 |
| 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 |

Memory map

The NET-50 and NS7520 platforms have the same memory map:

- Addresses from 0xf0000000 to 0xffffffff are reserved for devices internal to the NET+ARM.
- RAM on CS1 and CS2 is mapped from address 0x0 to 0x01ffffff.
- ROM on CS0 is mapped from address 0x02000000 to 0x021ffffff.
- 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.

Hardware Dependencies for ARM9-based Platforms

C H A P T E R 8

This chapter discusses NET+OS hardware dependencies for platforms that use the NS9360 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
- Ethernet PHY
- Endianness
- Timers
- Interrupts
- Memory map

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

DMA channels

The NS9750 and NS9360 use three DMA controllers. Two of them exist on the Bbus, and one exists in the Bbus Bridge module. (For detailed information, see the *NS9750 Hardware Reference* and the *NS9360 Hardware Reference*.)

One of the Bbus DMA controllers supports all Bbus peripherals except the USB device, and the other is dedicated to the USB device interface. The AHB DMA has two DMA channels. These channels can be used for memory-to-memory transfers on both the NS9750 and NS9360, and for transfers between memory and an external device on the NS9360. NET+OS does not use these channels. Your application can use the AHB DMA channels.

Ethernet PHY

NET+OS supports PHYs that use the MII interface. The PHY driver for the `ns9750_a` platform, which is implemented in the `miic.c` file, supports the LXT971A PHY by Intel. The PHY driver for the `ns9360_a` platform supports the ICS ICS1893AF and ICS 1893BF PHYs.

The PHY driver also supports these PHYs:

| PHY | Manufacturer |
|---|---------------------|
| FastCat (also known as the 3-volt enable PHY) | Lucent Technologies |
| LXT970 | Level One |
| LXT971A and LXT972A | Intel |
| AM79C874 and AM79C875 | AMD |

To support additional PHYs, you modify your platform's `mii.c` file.

To use the PHY interrupt to monitor the Ethernet link, set `BSP_USE_PHY_INTERRUPT` to `TRUE` in the `bsp.h` file. Do not set `BSP_USE_PHY_INTERRUPT` to `TRUE` if your PHY or platform does not support PHY interrupts. If you do not set `BSP_USE_PHY_INTERRUPT` to `TRUE`, the ThreadX timer is used to monitor the Ethernet link.

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

Endianness

The BSP supports big endian mode only.

General purpose timers

This section describes how the general purpose timers are used.

System timers

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

This table shows how timers 0-3 are used:

| Timer | How used by NET+OS |
|-------|--|
| 0 | <p>As the system heartbeat clock. The kernel uses the system heartbeat clock for timing and pre-emption of tasks.</p> <p>The <code>BSP_TICKS_PER_SECOND</code> constant in the <code>bsp.h</code> file controls the frequency of the system heartbeat clock. This value, which determines the heartbeat rate, should be between 1 and 1000. A value of 100, for example, provides a heartbeat rate of one tick every ten milliseconds.</p> |
| 1 | <p>Used by <code>NAuWait</code> and <code>NAWait</code>, which the flash driver uses to:</p> <ul style="list-style-type: none"> ■ Provide delays needed for programming flash, ■ Provide the reads that are needed to verify that a flash was properly programmed. |
| 2 | <p>To support the statistical profiler that is included with NET+OS. You use the profiler to understand trends of execution.</p> <p>The profiler records the location of an application using two resources - the FIQ interrupt and Timer 2 - that normally are not used.</p> |
| 3 | <p>To support the USB device DMA timeout function. Used by the USB device driver to close out a DMA transfer when the received size matches a multiple of the endpoint packet size. For example, if the packet size is 64, this timer is needed to close out the DMA buffer when the data received is 64, 128, or $n \times 64$.</p> <p>If you do not plan to use a particular feature, you can shut it off, and use the timer in your application. This applies only to the timers that NET+OS uses.</p> |

All other general purpose timers

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

Interrupts

The interrupt priorities are specified in the `bsp.c` file in the `platforms` directory. You can modify the priority of the interrupts by editing the `NAAhbPriorityTab` and `NABbusPriorityTab` tables in `bsp.c`.

The Bbus peripherals - all four serial ports, the USB device, and the 1284 - combine all their interrupts into one Bbus Aggregate interrupt. The Bbus 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 a description of interrupts in NET+OS, see Appendix F, “Processor Modes and Exceptions.”

For information about the interrupt controller, see the *NS9750 Hardware Reference* and the *NS9360 Hardware Reference*.

System clock

The constant `NA_ARM9_INPUT_FREQUENCY` 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 the PLL is used.

Chip selects

NET+OS requires the flash ROM to be connected to `CS1`, and RAM to be connected to `CS4`. The exception to this is if SPI flash is used. In that case, nothing needs to be connected to `CS1`. RAM on `CS4` is mapped to the physical address range from `0x0` to `0x0fffffff`. ROM on `CS4` is mapped to the physical address range from `0x50000000` to `0x507fffff`.

The chip selects are configured by functions you write in your platform's `cs.c` file. Each chip select has a function named `customizeSetupCSX` (`X` is replaced by the chip select number), which the initialization code calls to set up the chip select. The chip selects supplied for the NET+OS development board platforms set up `CS1` and `CS4` for the development boards. You must update these functions for your application hardware.

When a debugger is used, the debugger must configure the RAM chip select before it loads your application. The commands to do this are inside of a script file that the debugger executes whenever it prepares to download an application. The script `C:\Program Files\EPITools\edta22a\targets\ns9xxx\ns9xxx.cmd` sets up CS4 to support the RAM on the NET+OS development board. You must create your own debugger script that sets up the chip selects for your application hardware.

Memory map

The NS9360 and NS9750 have an embedded MMU. The MMU allows physical addresses to be remapped to virtual addresses. NET+OS sets up the address map shown next. The BSP assumes that all processor CSRs are mapped to their physical addresses.

The address map is set up in the `netos/src/bsp/platforms/CustomizeCache.c` file.

Porting NET+OS v6.0 Applications to NET+OS v6.3

C H A P T E R 9

This chapter describes the differences between the APIs in NET+OS 6.0 and NET+OS 6.3

Overview

This chapter describes the differences between the APIs in NET+OS 6.0 and NET+OS 6.3

The NET+OS 6.0 and NET+OS 6.1 releases supported the ARM7 and ARM9 platforms, respectively. NET+OS 6.3 merges the two API sets. In addition, some of the NET+OS 6.0 APIs have been deprecated or changed in the NET+OS 6.3 release.

This chapter lists these APIs and describes the replacements for them.

BSP Makefile

These are the changes to the BSP `Makefile`:

- NET+OS 6.0 built a single BSP library that you needed to delete and rebuild whenever you changed platforms.

NET+OS 6.3 builds separate libraries for each BSP platform.

The `Makefile` `PLATFORM` variable determines which platform is built when the BSP `Makefile` is executed and which BSP library is linked when applications are built.

Using either of these steps, you must set `PLATFORM` to indicate the platform you want to build for:

- Set the `PLATFORM` environment variable.
- Specify the value of `PLATFORM` in the command line to make; for example:
`make PLATFORM=net50bga_a.`

- NET+OS 6.0 `Makefiles` send the compiler and linker output to the console.

By default, NET+OS 6.3 `Makefiles` discard this output.

To see the compiler and linker messages, enter this command:

```
make MODE=verbose
```

Application Makefiles

In NET+OS 6.0, the `Makefiles` in the sample applications attempted to build the application for any platform, even for platforms that did not support the application.

In NET+OS 6.3, sample applications that cannot run on all platforms determine the platform on which they are being built and will terminate if they are being built on an unsupported platform. If you create a new platform, you must modify these `Makefiles` to build the applications under the new platform.

Linker scripts

- In NET+OS 6.0, the `BSP Makefile` generated a set of linker scripts that were stored in the `netos63_gnu/src/linkerscripts` directory.
In NET+OS 6.3, these scripts have been moved to the `platforms` directory. For example, the linker script for the `net50bga_a` platform is stored in the `netos63_gnu/src/bsp/platforms/net50bga_a` directory.
- In NET+OS 6.0, the NET+OS libraries were specified in the linker scripts.
In NET+OS 6.3, the libraries are specified in each application's `Makefile`.

Bootloader files

In NET+OS 6.0, the bootloader `rom.bin` file was stored in `netos63_gnu/src/bsp/bootloader/romimage`.

In NET+OS 6.3, the bootloader `rom.bin` file is stored in the `platforms` directory. For example, the bootloader `rom.bin` file for the `net50bga_a` platform is stored in the `netos63_gnu/src/bsp/platforms/net50bga_a` directory.

Cache API

Significant hardware differences exist between the cache implementations on the NET+50 processor (there is no cache on the NS7520) and the ARM9-based processor.

Because of these differences, the NET+OS 6.0 cache API, which supports cache on the NET+50, is not supported on the ARM9 platforms. When you port your application to an NS9750 or NS9360, you must rewrite your code to use the ARM9 MMU API.

The NET+OS 6.0 cache API is still supported on the NET+50 platforms.

Embedded Networking Interface

The Embedded Networking Interface (ENI) API is no longer supported.

ISR API

These functions in the Interrupt Service Routine (ISR) API have been renamed:

- `NADisableIsr` has been renamed `naInterruptDisable`.
- `NAEnableIsr` has been renamed `naInterruptEnable`.
- `NAInstallIsr` has been renamed `naIsrInstall`.
- `NAUninstallIsr` has been renamed `naIsrUninstall`.

RAM API

These functions have been deprecated and are not supported on the NS9360 and NS9750 processors:

- `nccCopyCSSetup`
- `nccDetermineRamType`

Real Time Clock driver

NET+OS 6.0 had a Real Time Clock (RTC) driver that supported an external RTC chip. This driver was never implemented on any NET+OS development board, and the NET+OS 6.0 RTC driver has been dropped.

NET+OS 6.3 implements a new RTC driver that supports the RTC built into the NS9360 processor.

These functions defined in the NET+OS 6.0 RTC driver are no longer supported:

- `NAinstallRealTimeClockTime`
- `rtcGet`
- `rtcInitialize`
- `rtcSet`

SYSCLK API

These functions in the SYSCLK API have been deprecated:

- `NAgetSysClkFreq`. Use `NAgetCpuClkFreq` or `NAgetBbusClkFreq` instead.
- `NAgetXtalFreq`. Use `NAgetSysOscFreq` instead.

GPIO configuration

NET+OS 6.0 supplied a set of functions that you, the developer, customized to configure the General Purpose I/O (GPIO) pins for your application.

In NET+OS 6.3, you configure GPIO by setting constants in the platform's `gpio.h` file.

These customization hooks are no longer supported.

- `customizeSetupPortA`
- `customizeSetupPortB`
- `customizeSetupPortC`
- `customizeSetupPortD`

- `customizeSetupPortF`
- `customizeSetupPortG`
- `customizeSetupPortH`

SPI API

The NET+OS 6.0 SPI API is deprecated and has been replaced by the NET+OS SPI master driver in NET+OS 6.3.

Write new applications to use the new SPI master driver.

Because the old driver will be discontinued in a future release, Digi strongly recommends that you port old applications to the new driver.

Stack sizes for exception handlers

In NET+OS 6.0, the stack sizes for the exception handlers were set in the `settings.s` file in the `platforms` directory.

In NET+OS 6.3, these values are set in the `init_settings.h` file.

Interrupt priorities

On the NET-50 and NS7520 platforms, interrupt priorities are determined by the `NAInterruptPriority` table in the platform's `bsp.c` file.

If you port your application to the NS9360 or NS9750, be aware that interrupt priorities on these platforms are determined by the `NAAhbPriorityTab` and `NABbusPriorityTab` tables in the platform's `bsp.c` file.

Porting NET+OS v6.1 Applications to NET+OS v6.3

C H A P T E R 1 0

This chapter describes the differences between the APIs in NET+OS 6.1 and NET+OS 6.3.

Overview

The two previous releases of NET+OS, 6.0 and 6.1, supported the ARM7 and ARM9 platforms respectively. NET+OS 6.3 merges the two API sets. Some of the NET+OS 6.1 APIs have been deprecated or changed in the 6.3 release. This chapter lists these APIs and describes the replacements for them.

BSP Makefile

The `BSP Makefile` has changed:

- NET+OS 6.1 built a single BSP library that needed to be deleted and rebuilt whenever you changed platforms.
NET+OS 6.3 builds separate libraries for each BSP platform.
The `Makefile` `PLATFORM` variable determines which platform is built when the `BSP Makefile` is executed, and which BSP library is linked when applications are built. You must set `PLATFORM` to indicate which platform you want to build for, using either of these steps:
 - Set the `PLATFORM` environment variable.
 - Specify the value of `PLATFORM` in the command line to make; for example:
`make PLATFORM=net50bga_a`
- NET+OS 6.1 `Makefiles` send the compiler and linker output to the console. By default, NET+OS 6.3 `Makefiles` discard this output. To see the compiler and linker messages, enter this command:
`make MODE=verbose`

Application Makefiles

In NET+OS 6.1, the `Makefiles` in the sample applications attempted to build the application for any platform, even for platforms that did not support the application.

In NET+OS 6.3, sample applications that cannot run on all platforms determine the platform on which they are being built and will terminate if they are being built on an unsupported platform. If you create a new platform, you must modify these `Makefiles` to build the applications under the new platform.

Linker scripts

- In NET+OS 6.1, the `BSP Makefile` generated a set of linker scripts that were stored in the `netos63_gnu/src/linkerscripts` directory.
- In NET+OS 6.3, these scripts have been moved into the `platforms` directory. For example, the linker scripts for the `ns9750_a` platform is stored in the `netos63_gnu/src/bsp/platforms/ns9750_a` directory.
- In NET+OS 6.1, the NET+OS libraries were specified in the linker scripts. In NET+OS 6.3, libraries are now specified in each application's `Makefile`.

Bootloader files

In NET+OS 6.1, the `rom.bin` bootloader file was stored in `netos63_gnu/src/bsp/bootloader/romimage`.

In NET+OS 6.3, the `rom.bin` bootloader file is stored in the `platforms` directory. For example, the bootloader `rom.bin` file for the `ns9750_a` platform is stored in the `netos63_gnu/src/bsp/platforms/ns9750_a` directory.

Client parallel driver

The client parallel driver has been simplified.

- The 6.1 `PCM_SET_RX_BUFFER` and `PCM_GET_TX_BUFFER ioctl` commands, which the application used to send empty receive buffers to the driver and get empty transmit buffers from the driver, have been dropped. The 6.3 driver does its own buffer management.
- The 6.1 `PCM_SET_TX_CHANNEL` and `PCM_SET_RX_CHANNEL ioctl` commands, which selected between data and command channels, are no longer supported. The underlying hardware does not support this functionality.

- The 6.1 `PCM_SET_SUPPORTED_MODE` and `PCM_GET_SUPPORTED_MODE` `ioctl` commands have been eliminated. The parallel port hardware automatically negotiates the interface mode with the host. The application can use the `PCM_SET_CHANGE_CALLBACK` `ioctl` command to install a callback function that is called with the newly selected interface mode whenever the mode changes.
- The 6.1 `PCM_SET_RX_BUFLN`, `PCM_GET_RX_RING_SIZE`, and `PCM_GET_TX_RING_SIZE` `ioctl` commands have been eliminated. The size and number of receive and transmit buffers are now set in the `1284.h` file in the `platforms` directory.
- The 6.1 `PCM_GET_CHANGE_CALLBACK` `ioctl` command has been dropped.

I2C driver

- The `MCI2cBuildMsg` function has been renamed `NAI2CBuildMsg`.
- The `MC_I2C_MESSAGE_TYPE` data type has been renamed `NA_I2C_MESSAGE_TYPE`.
- The `MC_I2C_BUFFER_STATE` data type has been renamed `NA_I2C_BUFFER_STATE`.
- The `NAI2CInit` and `NAI2COperation` functions have been added to support easier I2C Master operation without the use of I/O function calls.

Interrupt Service Routine (ISR) API

- `MCDisableIsr` has been renamed `naInterruptDisable`.
- `MCEnableIsr` has been renamed `naInterruptEnable`.
- `MCInstallIsr` has been renamed `naIsrInstall`. The `MCInstallIsr` function takes four parameters, but `naIsrInstall` takes only three. The fourth parameter to `MCInstallIsr` is a flag word that uses two bits. One bit determines whether the interrupt request line is high or low active when installing an ISR for an external interrupt. In NET+OS 6.3, you do this by setting the appropriate `BSP_GPIO_MUX_IRQ_X_CONFIG` constant in the platform's `gpio.h` file. The other bit determines whether interrupt is the Fast Interrupt Request (FIQ). In NET+OS 6.3, you do this by calling the `naIsrSetFiq` function. `MCUninstallIsr` has been renamed `naIsrUninstall`.

MMU API

The 6.1 `nonCachedMalloc` and `nonCachedFree` functions have been deprecated. They should not be used. Current applications that use them should be rewritten to use the 6.3 functions.

- Use the `NAVaToUncachedVa` function to translate the Virtual Address (VA) of a cached buffer to its uncached equivalent. The buffer can be dynamically or statically allocated. Use the `NACleanBuffer` function before reading or writing to the uncached VA. Use the `NAInvalidateBuffer` function after writing to the uncached VA.
- Use the `NAVaToPhys` function to get the physical address of a buffer given its VA. The buffer can be dynamically or statically allocated.
- Always use the `NABeforeDMA` and `NAAfterDMA` macros on DMA buffers before and after a DMA transfer.

PLL functions

Several PLL functions have been renamed. This table shows the NET+OS 6.1 names and the new NET+OS 6.3 names:

| This NET+OS 6.1 name | Has been changed to this NET+OS 6.3 name |
|------------------------------------|--|
| <code>MCReadPLLNSW</code> | <code>NAReadPLLNSW</code> |
| <code>MCSetPLLNSW</code> | <code>NASetPLLNSW</code> |
| <code>MCReadPLLNDStatus</code> | <code>NAReadPLLNDStatus</code> |
| <code>MCReadPLLISStatus</code> | <code>NAReadPLLISStatus</code> |
| <code>MCReadPLLBypassStatus</code> | <code>NAReadPLLBypassStatus</code> |
| <code>MCSetSWChange</code> | <code>NASetSWChange</code> |
| <code>MCSetPLLBypassSW</code> | <code>NASetPLLBypassSW</code> |
| <code>MCReadPLLBypassSW</code> | <code>NAReadPLLBypassSW</code> |
| <code>MCReadCPUSpeedGrade</code> | <code>NAReadCPUSpeedGrade</code> |

Real time clock driver

NET+OS 6.1 had a real time clock (RTC) driver that supported an external RTC chip. This driver was never implemented on any NET+OS development board, and the NET+OS 6.1 RTC driver has been dropped.

NET+OS 6.3 implements a new RTC driver that supports the RTC built into the NS9360 chip.

These functions that were defined in the NET+OS 6.1 RTC driver are no longer supported:

- `NAinstallRealTimeClockTime`
- `rtcGet`
- `rtcInitialize`
- `rtcSet`

GPIO configuration

In NET+OS 6.1, the external interrupts were configured to be high active or low active by the `MCInstallIsr` function.

In NET+OS 6.3, this is determined by the value of the external interrupt line's `BSP_GPIO_MUX_IRQ_X_CONFIG` constant in the platform's `gpio.h` file, where `X` indicates which external IRQ line. You can use this configuration setting to select between level sensitive high active, level sensitive low active, rising edge, and falling edge interrupt triggers.

Timer driver

The NET+OS 6.1 timer driver has been replaced. These functions are no longer supported:

- `MCDisableTimer` — **Use** `NATimerStop` **to stop a timer.**
- `MCEnableTimer` — **Use** `NATimerStart` **to start a timer.**
- `MCSetTimerClockSelect` — **Use** `NATimerConfigure` **to select the clock input to a timer.**

- MCSetTimerMode — Use NATimerConfigure to select the timer's mode.
- MCSetTimerInterruptSelect — Use NATimerInterruptEnable to enable a timer's interrupt, and NATimerInterruptDisable to disable a timer's interrupt.
- MCSetTimerUpDownSelect — Use NATimerConfigure to select whether a timer counts up or down.
- MCSetTimerBit — Use NATimerOpen to set the size of a timer.
- MCSetTimerReloadEnable — Use NATimerConfigure to determine whether a timer should automatically reload.
- MCReloadTimerCounter — Use NATimerConfigure to set the reload count.
- MCGetTimerCounter — Use NATimerRead to read the current timer value.
- MCClearTimerInterrupt — Use NATimerInterruptAck to acknowledge a timer interrupt.

SPI API

The NET+OS 6.1 SPI API is deprecated. This API has been replaced by the NET+OS SPI master driver in NET+OS 6.3. New applications should be written to use the new SPI master driver. You should port old applications to the new driver because it will be discontinued in a future release.

Network heap caching

The NET+OS 6.1 `BSP_CACHE_NETWORK_HEAP` configuration constant has been dropped. The network heap is always cached in NET+OS 6.3.

USB host API

The NET+OS 6.1 USB host API and USB host header files have been changed. USB host applications written under NET+OS 6.1 must be ported to use the NET+OS 6.3 USB host API. All the existing USB host device class drivers use the NET+OS 6.3 USB host API.

The `usbHost.h` USB host header file has been renamed to `usbHostApi.h`. Within the file, some of data structures have been changed. Therefore, USB host-related compiler errors require referring to the specific data structures in this file.

These USB host API functions have been replaced:

- `usbHostInit` — Use `usb_host_init` to initialize the USB host.
- `usbRegister` — Use `usb_register` to register a device class driver.
- `usbDeregister` — Use `usb_deregister` to de-register a device class driver.
- `usbBulkOut` and `usbBulkIn` — Use `usb_bulk_transfer` to perform bulk data transfers.
- `usbGetString` — Use `usb_get_string` to retrieve USB device string data.
- `usbGetDeviceDescriptor` — Use `usb_get_device_descriptor` to retrieve USB device descriptor data.
- `usbGetStatus` — Use `usb_get_status` to retrieve USB device status data.
- `usbClearFeature` — Use `usb_clear_feature` to send a clear feature command to the USB device.
- `usbSetFeature` — Use `usb_set_feature` to send a set feature command to the USB device.
- `usbClearEndpointFeature` — Use `usb_clear_endpoint_feature` to send a clear endpoint feature to the USB device.
- `usbSetConfiguration` — Use `usb_set_configuration` to enable device configuration in the USB device.
- `usbSetInterface` — Use `usb_set_interface` to select an interface in the USB device.
- `usbGetConfiguration` — Use `usb_get_configuration` to retrieve the device configuration from the USB device.
- `usbRequestIrq` — Use `usb_request_interrupt_transfer` to request interrupt transfers from the USB device.

The NET+OS 6.1 USB host API functions that are device class requests have been moved to the respective device class drivers in the `netosxxx\src\usb_host_drivers` directory. *netosxxx* is your NET+OS installation directory.

These USB hub-related API functions have been replaced.

- `usbHubInit` — Use `usb_hub_init` to initialize the USB hub driver.
- `usbGetHubDescriptor` — This function is in the NET+OS 6.1 USB host API and is replaced by `usb_hub_get_hub_descriptor` in `usbHub.c` in `netosxxx\src\usb_host_drivers\hub`. Use `usb_hub_get_hub_descriptor` to retrieve the USB Hub device descriptor data.
- `usbClearPortFeature` — This function is in the NET+OS 6.1 USB host API and is replaced by `usb_hub_clear_port_feature` in `usbHub.c` in `netosxxx\src\usb_host_drivers\hub`. Use `usb_hub_clear_port_feature` to send a clear port feature command to the USB hub device.
- `usbSetPortFeature` — This function is in the NET+OS 6.1 USB host API and is replaced by `usb_hub_set_port_feature` in `usbHub.c` in `netosxxx\src\usb_host_drivers\hub`. Use `usb_hub_set_port_feature` to send a set port feature command to the USB hub device.
- `usbGetHubStatus` — This function is in the NET+OS 6.1 USB host API and is replaced by `usb_hub_get_hub_status` in `usbHub.c` in `netosxxx\src\usb_host_drivers\hub`. Use `usb_hub_get_hub_status` to retrieve send a clear port feature command to the USB hub device.
- `usbGetPortStatus` — This function is in the NET+OS 6.1 USB host API and is replaced by `usb_hub_get_port_status` in `usbHub.c` in `netosxxx\src\usb_host_drivers\hub`. Use `usb_hub_get_port_status` to retrieve the port status of the USB Hub device.

These USB keyboard related API functions have been replaced. `netosxxx` is your NET+OS installation directory.

- `usbKeyboardIni` - Use `usb_keyboard_init` to initialize the USB Keyboard driver.
- `usbSetReport` - This function is in the NET+OS 6.1 USB host API and is replaced by `usb_keyboard_set_report` in `usbKeyboard.c` in `netosxxx\src\usb_host_drivers\keyboard`. Use `usb_keyboard_set_report` to send a set report command to the USB device.
- `usbGetReport` - This function is in the NET+OS 6.1 USB host library and is replaced by `usb_keyboard_get_report` in `usbKeyboard.c` in `netosxxx\src\usb_host_drivers\keyboard`. Use `usb_keyboard_get_report` to send a get report command to the USB device.

- `usbSetIdle` - This function is in the NET+OS 6.1 USB host API and is replaced by `usb_keyboard_set_idle` in `usbKeyboard.c` in `netosxxx\src\usb_host_drivers\keyboard`. Use `usb_keyboard_set_idle` to send a set idle command to the USB device.
- `usbSetProtocol` - This function is in the NET+OS 6.1 USB host API and is replaced by `usb_keyboard_set_protocol` in `usbKeyboard.c` in `netosxxx\src\usb_host_drivers\keyboard`. Use `usb_keyboard_set_protocol` to send a set protocol command to the USB device.
- `usbGetProtocol` - This function is in the NET+OS 6.1 USB host API and is replaced by `usb_keyboard_get_protocol` in `usbKeyboard.c` in `netosxxx\src\usb_host_drivers\keyboard`. Use `usb_keyboard_get_protocol` to send a get protocol command to the USB device.

These USB mouse-related API functions have been replaced. `netosxxx` is your NET+OS installation directory.

- `usbMouseInit` - Use `usb_mouse_init` to initialize the USB Mouse driver.
- `usbGetHidDescriptor` - This function is in the NET+OS 6.1 USB host API and is replaced by `usb_mouse_get_hid_descriptor` in `usbMouse.c` in `netosxxx\src\usb_host_drivers\mouse`.
Use `usb_mouse_get_hid_descriptor` to request the descriptor for an HID (Human Interface Device).

This USB printer related API function has been replaced:

- `usbPrinterInit` - Use `usb_printer_init` to initialize the USB printer driver.

Appendix A: Library Makefile System

Overview

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

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

```
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 convenient to set the shell variable and then just enter `make`. To set the `PLATFORM` shell variable, enter:

```
export PLATFORM=my_platform
```

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

```
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 is a list of the supported platforms:

- `ns9750_a`
- `ns9360_a`
- `connectme`
- `connectem`
- `connectsp`
- `ns7520_a`
- `net50bga_a`
- `connectwime`
- `connectwiem`
- `net50_d`

Two optional variables are:

- `DEBUG=on` — Turns on the debugging information.

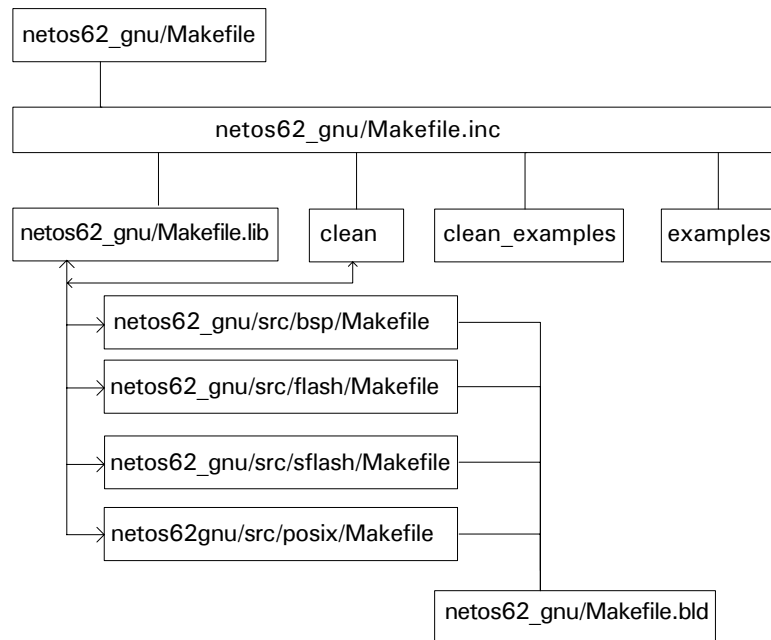
By default, this variable is set to `off`, which turns on the `NETOS_DEBUG` flag; the `NETOS_DEBUG` flag then turns on the `NA_ASSERT` macro.

- `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 diagram shows how the system is organized; the list after the diagram describes the directories.



- netos63_gnu/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.
- netos63_gnu/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 netos63_gnu
```
- 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:

```
netos63_gnu/src/bsp directory
```
- 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 *PROCESSOR*, *ENDIAN* and *TOOLSET* variables. This is the library directory structure:

```
netos63_gnu/lib/arm7/32b/gnu/
```

```
netos63_gnu/lib/arm9/32b/gnu/
```

Except for the BSP library, *libbsp.a*, all libraries are found in 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:

```
netos63_gnu/lib/arm7/32b/gnu/bsp/connectme/
```

```
netos63_gnu/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.

For information about the application build files, see Appendix B, “Application Makefile.”

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

| Variable | Description |
|------------------|---|
| <i>OBJS</i> | Location of object files |
| <i>SRCDIR</i> | List of source directories |
| <i>LOCAL_LIB</i> | Name of the library |
| <i>INCDIRS</i> | List of include paths with -I prefix |
| <i>C_OBJ</i> | List of C object files |
| <i>S_OBJ</i> | List of assembly object files |
| <i>CC_OBJ</i> | List of C++ object files (*.cc and *.cxx) |

| Variable | Description |
|-------------------|--|
| <i>MY_CFLAGS</i> | List of C flags (such as -Werror, -Os). If you want to turn on debug information, set the -g flag here |
| <i>MY_DEFINES</i> | List of defines with the -D prefix |

Adding a new libraries to the system

- To add new libraries:
 - 1 Copy a child Makefile from netos63_gnu/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 netos63_gnu/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:

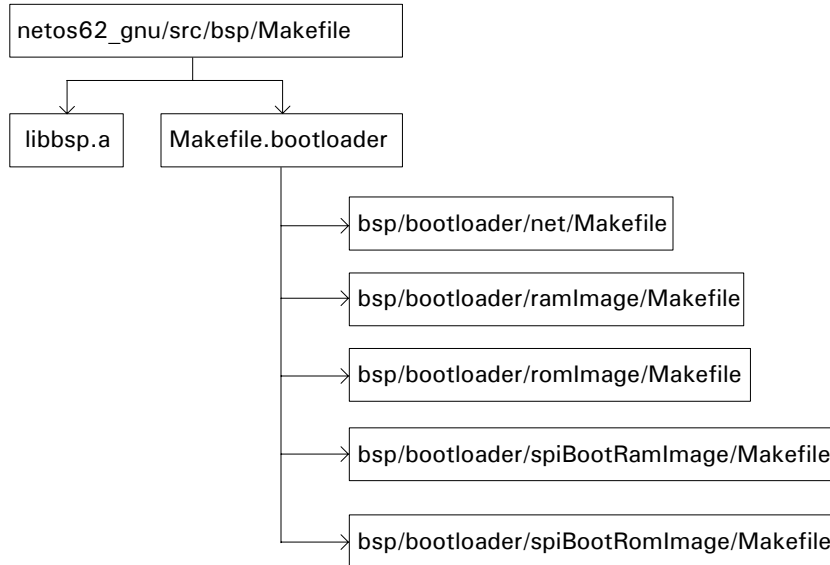
```
make PLATFORM=my_platform clean
```

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

Bootloader Makefile



Because bootloader Makefiles are called from netos63_gnu/src/bsp/Makefile, you always build the bootloader as part of the BSP. The next diagram shows how the bootloader Makefile system is organized. The list after the diagram describes the directories.



- `netos63_gnu/src/bsp/bootloader/net/Makefile` **generates** `libnet.lib` **directory under** `bootloader/libs/gnu/my_platform/`
- `netos63_gnu/src/bsp/bootloader/ramImage/Makefile` **generates** `blram.s`, `bin` **to** `s` **converted file, which is linked in to** `rom.bin`
- `netos63_gnu/src/bsp/bootloader/romImage/Makefile` **generates** `rom.bin` **under** `netos63_gnu/src/bsp/platforms/my_platform/`
- `netos63_gnu/src/bsp/bootloader/spiBootRamImage/Makefile` **generates** `spi_blram.s`, `bin` **to a converted file, which is linked in to** `spi_rom.bin` **(supported only on ARM9)**
- `netos63_gnu/src/bsp/bootloader/romImage/Makefile` **generates** `spi_rom.bin` **under** `netos63_gnu/src/bsp/platforms/my_platform/` **(supported only on ARM9)**

Example: using the Makefile

This example shows how to build NET+OS libraries and the `naftpapp` sample application:

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

```
cd c:/netos63_gnu
```

- 2 Enter these commands:

```
export PLATFORM=my_platform
export MODE=verbose
make
cd src/examples/naftpapp/32
make
```

The results of the build, which are located in the `32b` directory, include these files:

- `image.elf` - Debug image
- `image.bin` - Executable image
- `image.map` - Map file for image
- `image.sym` - Symbol table for image

If the ROM image also is built, these files are created in the `32b` directory:

- `rom.elf` - ROM debug image
- `rom.bin` - ROM executable image
- `rom.map` - ROM map file for image
- `rom.sym` - Symbol table for ROM image

Appendix B: Application Makefile

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 |
| <code>rom.elf</code> | ROM-based application image in ELF format. Contains debug symbols; you use it to debug your ROM-based application |

| File | Description |
|---------|--|
| 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 |

Sections of the Makefile

The application `Makefile` includes these sections:

| 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. Applications that have critical boot time requirements might want to use the application ROM image (or modify the decompression scheme in the bootloader). <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), eliminating the need to decompress the application image from ROM to RAM and reducing 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> |
| 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> |
| APP_ASM_FLAGS | <p>Assembler directives and defines passed through to the GCC assembler and used during assembly of the <code>APP_ASM_FILES</code> application files.</p> |

| Section | Description |
|-------------|--|
| 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. |
| 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 the linkerScripts/Makefile.app.inc. Defines the NS9360 flag and uses the armv5te architecture type, needed for the arm926ejs. |
| Makefile.appbuild.ns9750 | This Makefile is included in the linkerScripts/Makefile.app.inc. Defines the NS9750 flag and uses the armv5te architecture type, needed for the arm926ejs. |
| Makefile.appbuild.original | This Makefile is for linkerScripts/Makefile.appbuild.\$(BUILD_METHOD) for all other processor types (which defaults to the ARM7 family). This is where the arm7tdmi processor type is selected. |

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.

To build your application:

- 1 Change to this directory:

```
cd netos63_gnu/source/examples/sample_application/32b
```
- 2 Do one:
 - *Either* enter this command:

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

```
export PLATFORM=my_platform
make
```

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 it 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.

Appendix C: Customizing the SPI Bootloader

Overview

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

You enable SPI-EEPROM boot logic by strapping off the `boot_cfg` pins to the boot from the SDRAM setting in the Miscellaneous System Configuration and Status register. 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 works by interfacing 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 the “SPI Bootloader Overview” in the online help.

The `SPI bootloader` is copied from ROM to RAM at powerup through the `SPI-boot_logic` hardware. The image can be compressed to save space in serial flash. In normal operation, the RAM image verifies that the application image stored in serial 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.

For the NS9750 and NS9360 processors, SPI serial flash (or SPI-EEPROM) must be connected to serial port B because the boot logic does not communicate with any other serial port.

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

The `SPI bootloader` utility consists of two application images:

- **ROM image.** A small application that is copied from SPI flash to RAM by hardware and executed in RAM
- **RAM image.** Your large application, which runs from RAM

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

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

SPI bootloader application images

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

ROM image

The ROM image is located in the first (and possibly the second) sector of SPI serial flash (or SPI EEPROM). The processor automatically copies the ROM image to RAM after a reset and immediately starts to execute the SPI bootloader ROM image. The `SPI bootloader` uses the BSP initialization code to configure the hardware.

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

You build the ROM image with the `SPI bootloader` utility, which is located in `/bin`.

RAM image

The RAM image is stored as an application image in SPI serial flash (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 `SPI bootloader` (about 128 KB), the compressed application image downloaded from the network, and the decompressed version of the application image.
The maximum sizes of both the compressed and decompressed versions of the application image are set in the linker script customization file.
- The application image must be built with the `boothdr` utility, which is located in `/bin`.

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

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

The RAM image uses these steps to perform the recovery:

- 1 Initializes the Ethernet driver.
- 2 Initializes the UDP stack.
- 3 Downloads the application image from a network server to RAM.
- 4 Validates the downloaded application image by performing a `CRC32` checksum.
- 5 Stores the image into flash.
- 6 Resets the unit, which restarts the process.

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

Application image structure

An application image consists of:

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

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

Application image header

The application image header has two sections of variable length. The first part contains data that the `SPI 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;
} b1ImageHeaderType;
```

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 boothHdr to identify this header as a valid image header. |
| <i>version</i> | Set to 0 for this version 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. |

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

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

boothdr utility

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

- 1 Inserting a header at the beginning of the image.
The data to place inside the header is read from a configuration file.
- 2 Inserting a customer header.
You specify this action at the command line by providing the name of a file that contains the custom header.
- 3 Calculating a CRC32 checksum for the entire image, including the header, and placing it at the end of the file.

The `boothdr` utility takes this command line:

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

Arguments

| Argument | Description |
|--------------------|-------------------------------------|
| <i>config-file</i> | The name of the configuration file |
| <i>input-file</i> | The name of the bin file to convert |

| Argument | Description |
|---------------------------|--|
| <i>output-file</i> | The name of the file to create |
| <i>custom-header-file</i> | The name of a file that contains your custom header as binary data |

spibootldr utility

The `SPI bootldr` utility inserts a SPI boot header at the beginning of the ROM image. The `SPI boot` header is needed because the memory controller exits the reset state in non-operational mode, requiring the SPI-EEPROM boot logic to configure the memory controller as well as the external SDRAM before any memory access. The information required to configure the memory controller and the external SDRAM must be stored in a configuration header in the SPI serial flash (or SPI-EEPROM) in a contiguous block that starts at address 0. Each entry in the header, with the exception of the pad entry, must be 4 bytes long.

The size of the configuration header varies from 128 bytes to 130 bytes because of the variable length nature of the SPI serial flash (or SPI-EEPROM) `read` command.

The `spibootldr` utility takes this command line:

```
spibootldr config_file input_file output_file
```

These are the arguments for the `spibootldr` utility:

| Argument | Description |
|--------------------|--|
| <i>config_file</i> | The name of the SDRAM configuration file. NET+OS 6.3 uses <code>bsp/platforms/my_platform/init_settings.h</code> . |
| <i>input_file</i> | The name of the bin file to convert. |
| <i>output_file</i> | The name of the file to create. |

For more information about the SPI boot header, see the “SPI Bootloader Overview” in the online help.

For information about SPI-EEPROM boot logic, see the hardware documentation for your processor.

Generating an image

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

- 1 The Makefile is compiled and linked.
The application is linked for its execution address in RAM (`image.bin`) or ROM (`rom.bin`), but is linked as a ROM application. Normally, this image is set up for debugging.
- 2 The compression program that ships with NET+OS compresses the image.
- 3 The `bootldr` creates an application image that the `bootloader` supports.

Configuration file

The configuration file contains configuration information in the form of several keyword/value pairs. The default configuration file, `bootldr.dat`, is stored in the `bsp/platforms/my_platform` directory.

This table describes the keyword/value pairs:

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

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

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

```
WriteToFlash      Yes
Compressed        Yes
ExecuteFromRom    No
FlashOffset       0x20000
RamAddress        0x4000
MaxFileSize       0xD0000
```

General bootloader limitations

Be aware of these general limitations about the `bootloader`:

- The `bootloader`'s DHCP/BOOTP client is limited. The client supports options for getting the IP address, subnet mask, gateway address, boot image 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.

Customizing the SPI bootloader utility

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

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

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

Customization hooks

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

| Function | Description |
|----------------------------------|--|
| <code>NABlReportError</code> | Called whenever an error occurs |
| <code>getMacAddress</code> | Gets the Ethernet MAC address that 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 |

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.

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

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

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

getMacAddress

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

Format

```
char *getMacAddress;
```

Arguments

None

Return values

Returns the Ethernet MAC address as an array of characters

Implementation

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

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

isImageValid

Determines whether a downloaded image is valid.

Format

```
int isImageValid (bImageInfoType *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, this routine first reads the image in serial flash into RAM.

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

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

shouldDownloadImage

Determines whether to download an application image from the network.

Format

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

Arguments

None

Return values

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

Implementation

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

```
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 *)&d1Buffer[0], sizeof
            (blImageHeaderType), 0)
            != BL_SUCCESS)
            NAB1ReportError(SIMPLE_SPI_EEPROM_READ_FAIL);
        fmemcpy(&imageInfo, &d1Buffer[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. Some possibilities are:

- 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

Appendix D: Customizing the ROM Bootloader

Overview

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

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

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

The `bootloader` utility consists of two application images:

- **ROM image.** A small application that runs from ROM
- **RAM image.** Your large application, which runs from RAM.
The RAM image verifies that the application image stored in flash is correct, decompresses it to RAM, and executes it.

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

Bootloader application images

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

ROM image

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

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

RAM image

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

The RAM image has these requirements:

- Sufficient RAM must be available to hold the RAM image portion of the `bootloader` (about 128 KB), the compressed application image downloaded from the network, and the decompressed version of the application image. The maximum sizes of both the compressed and decompressed versions of the application image are set in the linker script customization file.
- The application image must be built with the `boothdr` utility, which is located in `/bin`.

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

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

The RAM image uses these steps to perform the recovery:

- 1 Initializes the Ethernet driver.
- 2 Initializes the UDP stack.
- 3 Downloads the application image from a network server to RAM.
- 4 Validates the downloaded application image by performing a `CRC32` checksum.
- 5 Stores the image into flash.
- 6 Resets the unit, which restarts the process.

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

Application image structure

An application image consists of:

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

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

Application image header

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

This data structure defines the application image header:

```
typedef struct
{
    WORD32 headerSize;
    WORD32 naHeaderSize;
    char signature[8];
    WORD32 version;
    WORD32 flags;
    WORD32 flashAddress;
    WORD32 ramAddress;
    WORD32 size;
} 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 <i>bootHdr</i> to identify this header as a valid image header. |
| <i>version</i> | Set to 0 for this version of the image header. |
| <i>flags</i> | A bit field of flags. See the next table for details about bit values. |
| <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. |

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

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

boothdr utility

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

- 1 Inserting a header at the beginning of the image.
The data to place inside the header is read from a configuration file.
- 2 Inserting a customer header.
You specify this action at the command line by providing the name of a file that contains the custom header.
- 3 Calculating a CRC32 checksum for the entire image, including the header, and placing it at the end of the file.

The `boothdr` utility takes this command line:

Format

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

Arguments

| Argument | Description |
|---------------------------|--|
| <i>config-file</i> | The name of the configuration file |
| <i>input-file</i> | The name of the <code>bin</code> file to convert |
| <i>output-file</i> | The name of the file to create |
| <i>custom-header-file</i> | The name of a file that contains your custom header as binary data |

Generating an image

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

- 1 The `Makefile` compiles and links the image.
The application is linked for its execution address in RAM (`image.bin`) or ROM (`rom.bin`), but is linked as a ROM application. Normally, this image is set up for debugging.

- 2 The compression program that ships with NET+OS compresses the image.
- 3 The `bootldr` creates an application image that the `bootloader` supports.

Configuration file

The configuration file contains configuration information in the form of several keyword/value pairs. The default configuration file, `bootldr.dat`, is stored in the `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_LZSS_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> . |

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

| | |
|----------------|---------|
| WriteToFlash | Yes |
| Compressed | Yes |
| ExecuteFromRom | No |
| FlashOffset | 0x20000 |
| RamAddress | 0x4000 |
| MaxFileSize | 0xD0000 |

General bootloader limitations

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

- The `bootloader`'s DHCP/BOOTP client is limited. The client supports options for getting the IP address, subnet mask, gateway address, boot image 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, must have the same IP address).*

Overview of customizing

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

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

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

Customization hooks

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

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

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.

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

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

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

getMacAddress

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

Format

```
char *getMacAddress,(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)
```

Arguments

| Value | Description |
|------------------|-----------------------------|
| <i>imageInfo</i> | Pointer to the image header |

Return values

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

Implementation

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

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

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

shouldDownloadImage

Determines whether to download an application image from the network.

Format

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

Arguments

None

Return values

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

Implementation

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

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

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

getDefaultFilename

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

How the server uses the information depends on the implementation. If no 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. Some possibilities are:

- 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

Appendix E: Customizing the Address Configuration Executive

Overview

The Address Configuration Executive (ACE) controls the process of acquiring an IP address and other IP configuration settings, and configures the IP stack.

ACE provides built-in support for using static IP addresses and for these protocols:

- DHCP
- BOOTP
- Ping ARP
- RARP
- Auto IP

ACE invokes a set of callback functions at various points in the process of acquiring an address. The ACE API consists of a series of functions that the callbacks can use to get more information.

This appendix describes how to program changes for ACE. This document is not intended to provide knowledge of the Address Resolution Protocols; rather, it describes how some of these protocols can be managed in the ACE configuration.

Configuring ACE

To add or remove a protocol from ACE, you must change the NVRAM parameters (ACE Configuration).

These functions store the protocol -specific configuration to NVRAM for the interface identified in the call. `customizeAceSetInterfaceConfig()` writes all the protocol-specific ACE configurations, or the ACE configuration for an interface and `customizeAceSetConfig()` writes the entire ACE configuration into NVRAM.)

These APIs are available to applications for this purpose:

- `customizeAceSetConfig()`
- `customizeAceSetInterfaceConfig()`
- `customizeAceSetStaticConfig()`
- `customizeAceSetRarpConfig()`

- `customizeAceSetDhcpConfig()`
- `customizeAceSetBootpConfig()`
- `customizeAceSetAutoipConfig()`

After you set up the configuration information you want and save it to NVRAM, you can restart ACE. At that point, ACE reads the configuration parameters from NVRAM.

Setting the static IP configuration

To change the static IP configuration, you set the values in a structure of type `configAceStaticInfo`.

When the members `isConfigValid` and `isEnabled` are set to `TRUE`, the configuration can be processed by ACE (that is, it turns static IP on).

These are the arguments for static IP configuration:

- `auto_assign`:
 - When set to true, causes this configuration to take precedence over other protocols and runs with the startup delay 0.
 - When set to false, static IP is invoked after its startup delay, like any other protocol.
- `ip_address`, `subnet_mask` and `gateway` — Required parameters that are not described in this document.
- `name_server_address` — Can be specified. This is an IP address expressed as a 32-bit value.

startInfo structure

This member of the `configAceStaticInfo` structure contains these required parameters:

- `protocol` — A number defined in `ace_params.h` identifying that identifies the protocol (`ACE_PROT_STATIC` in this case).
- `priority` — A non-negative number. Priority granted to the protocol is inversely proportional to this number (0 is highest priority). Priority applies when several protocols acquire the IP address at the same time.

- `delay_before_start` — Number of seconds to delay starting this protocol.
- `shutdown_type` — One of three choices:
 - `ACE_ALWAYS_SHUTDOWN`
 - `ACE_CONT_IF_GOT_ADDRESS`
 - `ACE_NEVER_SHUTDOWN`

For static configuration shutdown, the type must be `ACE_ALWAYS_SHUTDOWN`.

Setting DHCP configuration

Note that in this section, all IP address parameters are 32-bit words in network byte order.

To change the DHCP configuration, set the values in a structure of type `configAceDhcpInfo`. When the members `isConfigValid` and `isEnabled` are set to `TRUE`, the configuration can be processed by ACE (that is, turns on DHCP).

These are the parameters in the DHCP configuration:

- `suggested_ip_address` — Optionally provided.
- `server_ip_address` — For `ACE_RESTART_DHCP_REUSE`.
- `gateway` — Default gateway address.
- `suggested_lease_time` — `ime_t` structure.
- `number_of_retries` — Must be 4.
- `lease_start_time` — Time recorded at start of lease.
- `dhcp_restart_type` — DHCP restart type; only `ACE_RESTART_DHCP_DISCOVER` is supported.
- `need_bcast_response` — Sets broadcast flag in DHCP message.
- `do_init_delay` — Enables initial random delay before sending Discover message.
- `arp_reply_timeout` — Reply timeout for ARP probe.
- `desired_params` — Array of DHCP options to send to the DHCP server.
- `num_desired_params` — Number of valid DHCP options)
- `startInfo` - Same structure as above. (See “Setting the static IP configuration,” earlier in this chapter.)
 - `protocol` — `ACE_PROT_DHCP`.
 - `SHUTDOWN_TYPE` — Must be either `ACE_CONT_IF_GOT_ADDRESS` or `ACE_NEVER_SHUTDOWN` for DHCP to renew the lease.

AUTOIP configuration

To change the AutoIP configuration, you set the values in a structure of type `configAceAutoipInfo()`.

When the members `isConfigValid` and `isEnabled` are set to `TRUE`, this configuration can be processed by ACE (that is, turns AUTOIP on).

These are the parameters in the AUTOIP Configuration:

- `autoip_local_addr` — IP address that AutoIP initially uses when trying to configure an address.
- `startInfo` — Same structure as above.
See “Setting the static IP configuration,” earlier in this chapter.
- `protocol` - `ACE_PROT_AUTOIP`.
- `shutdown_type` — **Must be either** `ACE_CONT_IF_GOT_ADDRESS` **or** `ACE_NEVER_SHUTDOWN`.

Stopping ACE

Stopping the service is necessary to make changes in the protocols which ACE uses to manage address events.

To stop ACE, call `aceStop`. The only parameter passed in this call is the interface name (for example, `eth0`).

Appendix F: Processor Modes and Exceptions

Overview

This appendix 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:

- 26 interrupt signals support AHB devices that are internal to the NS9750 and NS9360.
- 1 interrupt signal supports 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.
- 4 interrupt signals support external devices.
- 1 interrupt signal is not used and is considered reserved.
ARM7-based processors have different 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 then 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; that is, NET+OS 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.

The priority for each AHB source interrupt is specified 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 |
| 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 |

| AHB interrupt source | Software directive |
|----------------------|-------------------------|
| 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 |
| 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 |

| AHB Interrupt source | Software directive |
|----------------------|------------------------------|
| 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 |

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 then 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 have been processed, the NET+OS interrupt driver returns control to the application.

This table lists the supported interrupt sources in the Bbus and the associated software directives. The priority for each Bbus interrupt source is specified in the `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 |

| Bbus interrupt source | Software directive |
|-----------------------|--------------------|
| Serial 4 receive | SER4RX_INTERRUPT |
| Serial 4 transmit | SER4TX_INTERRUPT |
| Serial 3 transmit | SER3TX_INTERRUPT |
| Serial 2 transmit | SER2TX_INTERRUPT |
| Serial 1 transmit | SER2TX_INTERRUPT |
| USB | USB_INTERRUPT |
| Bbus DMA | BBUS_DMA_INTERRUPT |

System interrupts: ARM7-based platforms

The priority for interrupts is set by 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 then returns.
- 5 At this point, the interrupt driver checks for more pending interrupts. If any interrupts are found, their ISRs are called as well.
- 6 When all pending interrupts have been processed, the NET+OS interrupt driver returns control to the application.

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, it can interrupt an IRQ at any time.

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

Use the `naIsrSetFiq` function 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, the watchdog timer and the two general-purpose timers can be configured 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`.

Appendix G: Memory Usage in Networked Applications

Overview

Memory requirements and usage are two aspects of TCP/IP networking that often are misunderstood. For clarification, the term *network heap* refers to the memory used exclusively for the NET+OS TCP/IP stack. All NET+OS networking applications require a segment of network heap. This space is the initial block of memory allocated from the C library heap. It is used for initial static and dynamic allocation of TCP/IP memory needs and managed independently from the C library heap.

Two standard approaches to memory management used in TCP/IP stacks are byte pools and block pools, each with its own benefits and consequences.

Block pools

Block pools (allocation of fixed sized buffers) are beneficial because no search is needed to allocate a block. If one exists, it is merely allocated. The drawback, however, is the wasted space that is not used because of the fixed-size blocks.

For example, suppose all requests for blocks greater than 64 bytes but less than 128 bytes always receive a 128 byte buffer. Requests for 64 bytes would result in 50% wasted space.

Byte pools

Alternatively, a byte pool, which is a large block of bytes, is managed by a linked list to available blocks within the pool, and separated (or fragmented) by already allocated blocks. Traversing this list to find the best fit can become time consuming, and in the worst case, cause allocation failures when fragmentation is excessive. On the other hand, allocation utilization is 100% because the caller receives exactly what was requested. For example, when an application requests 64 bytes, the manager traverses its linked list until a 64 byte block is located.

These examples illustrate a time versus memory trade-off:

- A block pool is faster but wastes space.
- A byte pool takes longer to search and find the best fit block, but enables better use of the block.

The NET+OS network heap is, by default, a byte pool. A portion of the heap can be converted to a block pool.

Network heap application tuning

The NET+OS TCP/IP network heap is defined by parameters in `appconf.h`. The total memory (in bytes) allocated for the heap is defined by `APP_NET_HEAP_SIZE`, and as mentioned above, the heap is, by default, a byte pool.

To allocate portions of the network heap as a block pool, you can add these definitions to `appconf.h` and adjust them as needed:

```
#define APP_TCPIP_16BYTE_BLOCK_COUNT 60
#define APP_TCPIP_32BYTE_BLOCK_COUNT 60
#define APP_TCPIP_64BYTE_BLOCK_COUNT 100
#define APP_TCPIP_128BYTE_BLOCK_COUNT 20
#define APP_TCPIP_256BYTE_BLOCK_COUNT 10
#define APP_TCPIP_540BYTE_BLOCK_COUNT 200
#define APP_TCPIP_1836BYTE_BLOCK_COUNT 200
```

The network heap can be split into a byte pool and seven block pools of size 16, 32, 64, 128, 256, 540, and 1836. These block sizes were chosen based on needs of the TCP/IP stack. When a block pool runs out, a block from the next highest pool is used. When the pools run out, or when the size exceeds the largest block, memory is taken from the byte pool.

Memory usage in TCP connections

The total memory required, M_{Total} , of an active TCP/IP connection can be calculated as:

$$M_{\text{Total}} = M_{\text{Static}} + M_{\text{Recv}} + M_{\text{Transmit}}$$

where

- M_{Static} is a fixed constant that is required for socket data structures and state.
- M_{Recv} is the reserved buffer required for receiving data.
- M_{Transmit} is the buffer needed to store transmit data that might be needed for retransmission.

Other dynamic memory needs for TCP/IP stated timers and configuration are ignored at this time.

The value of M_{Recv} and $M_{Transmit}$ can be computed directly from the socket options for `SO_RCVBUF` and `SO_SNDBUF`, respectively. So as the TCP window size grows, $M_{Static} \ll M_{Recv}, M_{Transmit}$, and the size of M_{Total} can easily be approximated by:

$$M_{\text{Total}} \sim M_{\text{Recv}} + M_{\text{Transmit}}$$

For example, on a high throughput connection, where the TCP window is set to the maximum on both send and receive (64K), a connection will require a total of 128K bytes. Additionally, if this service requires the ability to service eight simultaneous connections, this service alone will require 1MByte of network heap, not including the heap needed for ARP, the passive listener, spare Ethernet buffers, or any other socket memory requirement.

When you design client-server systems, it is critical to consider and test for the worst-case usage models.

Another source memory usage, but more subtle, is the cost of maintaining a closed TCP/IP connection. When a client-server calls `closesocket`, it does not necessarily mean the memory associated with the connection is immediately freed up, and it's crucial which side closes first.

This aspect of TCP/IP is extremely sensitive to which party in the client-server pair closes first.

Active close of a TCP connection

An active close occurs when a TCP client-server first calls `closesocket`, which causes the unit to send a FIN segment. The unit's TCP connection state enters the `FIN_WAIT_1` state after sending the FIN and then enters the `FIN_WAIT_2` state after receiving the ACK to the sent FIN.

The unit's TCP connection state remains in the `FIN_WAIT_2` state until it receives a FIN segment from its peer half-opened connection. There is no TCP/IP timer to terminate from the `FIN_WAIT_2` state, and it's possible for connections to remain half-opened indefinitely, if, for example the peer has crashed, network connectivity is lost, or the client-server protocol is poorly designed.

To protect against sockets remaining in the `FIN_WAIT_2` state, the socket option `SO_KEEPALIVE` is recommended. This option actively probes the peer for disconnections or crashes and terminates the half-opened connections if the keep-alive timeout interval is exceeded. The keep-alive timeout interval is globally set; you can change the interval with the `NAIpSetKaInterval` API call.

Time wait state of a TCP connection

The TCP connection state transitions to the `TIME_WAIT` state (from the `FIN_WAIT_1` or `FIN_WAIT_2` states) after acknowledging the FIN from the peer. However, the TCP connection remains in `TIME_WAIT` state for $2 * \text{TCP_MSL}$ seconds. Note the default TCP MSL is 120 seconds, and therefore, the default `TIME_WAIT` interval is four minutes.

You can change the global per-system TCP MSL value using the `NAIpSetTcpMsl` API call. The value of TCP MSL can be set between 15 and 120 seconds, reducing the time memory and available sockets are tied up after the connection is closed.

The `TIME-WAIT` state is $2 * \text{MSL}$ and can be reduced using `NAIpSetTcpMsl`.

Using a connection reset instead of an orderly close

Another way to keep memory and sockets from lingering after connections are closed is to use the connection reset mechanism instead of an orderly close.

This example uses the connection reset mechanism:

```
struct linger op;
op.l_linger = 0;
op.l_onoff = 1;
setsockopt(fd, SOL_SOCKET, SO_LINGER, (char*)&op, sizeof op);
closesocket(fd);
```

In this example, instead of sending a FIN segment at the `closesocket()` call, a RST is sent instead. The drawback of this mechanism is that any remaining data in the send queue will be discarded.

Maximum number of sockets

The maximum number of active sockets is fixed at 128 and cannot be changed.

Additionally, the socket descriptor 0 cannot be used, so the maximum number of open sockets is limited to 127 (`MAX_SOCKETS - 1`).

[illegible]

Index-1

- deviceIoctl 99
- deviceOpen 95
- deviceRead 97
- deviceWrite 98
- deviceClose routine 96
- deviceEnter routine 93
- deviceInfo structure 90
- deviceInfo structures 90
- deviceInit routine 94
- deviceIoctl routine 99
- deviceOpen routine 95
- deviceRead routine 97
- devices.c file 90
- deviceTable array 90
- deviceWrite routine 98
- DHCP/BOOTP client 159, 176
- downloadImage routine 160, 167, 183

F

- FastCat PHY 107, 115

G

- generating
 - an image 158, 174
- getDefaultFilename routine 160, 166, 182
- getMacAddress routine 160, 162, 179

H

- hard-coding the MAC address 162, 179
- hooks, customization 160, 176, 177

I

- image, generating 158, 174
- Intel PHY 107, 114, 115
- ioctl function 90
- isImageValid routine 160, 163, 180

K

- keyword/value pairs in configuration file 159, 176

L

- Level One PHY 107, 115
- libc.a file and library 102
- limitations of the bootloader utility 159, 176
- Lucent Technologies PHY 107, 115
- LXT970 PHY 107, 115
- LXT971A PHY 114
- LXT971A PHY and LXT972A PHY 107, 115

M

- MAC address 160, 177
 - and hard-coding 162, 179
- mii.c file 107, 115
- modifying Cygwin's standard C library and startup file 101

N

- NABReportError routine 161
- NABReportError routine 160, 177
- NET+OS device driver interface (DDI) 100

O

open function 90

R

RAM image and bootloader utility 153,
171

rammain.c file 176

read function 90

reportError routine 178

return values for NET+OS DDI
routines 100

ROM image and bootloader utility 153,
170

S

setup functions 100

shouldDownloadImage routine 160, 164,
181

smicng.exe 6

spi_blmain.c file 160

spiboothdr.exe 6

T

TFTP client and bootloader utility 159,
176

ThreadX kernel and NET+OS device
drivers 102

U

User Datagram Protocol (UDP) stack and
bootloader utility 159, 176

W

write function 90

