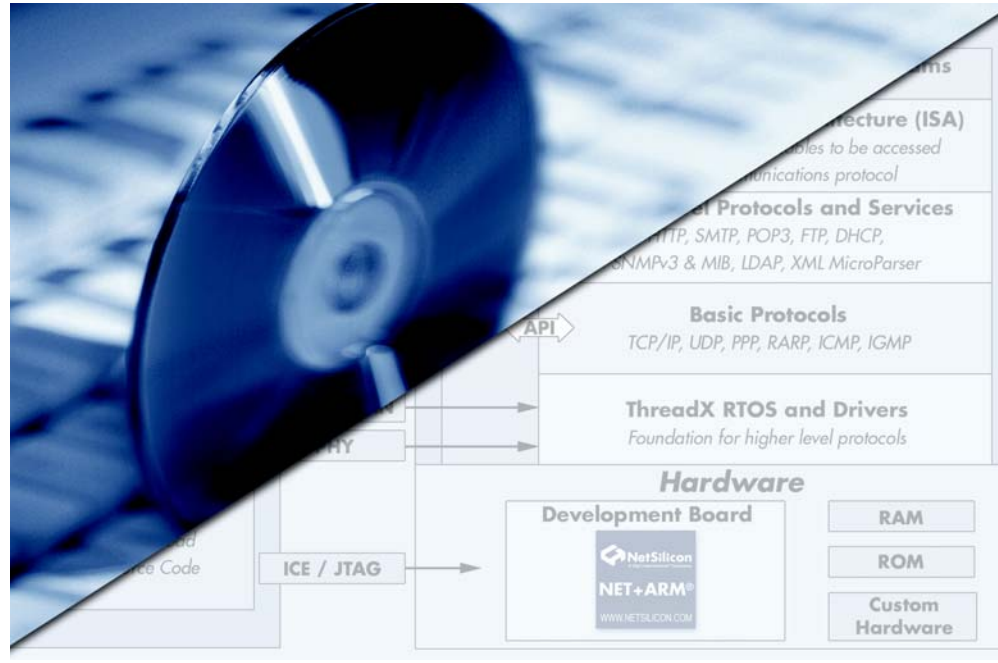
NetSilicon
A Digi International® Company

NET+OS with Green Hills BSP Porting Guide



NET+OS 5.0
8833231E

NET+OS with Green Hills BSP Porting Guide

Operating system/version: NET+OS 5.0
Part number/version: 8833231E
Release date: July 2002
www.netsilicon.com

Contents

Using This Guide

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

About this guide

This guide provides general information for porting the NET+Works for NET+OS Board Support Package (BSP) with Green Hills software to new hardware platform based on the NetSilicon development board. NET+OS, a network software suite optimized for the NET+ARM chip, is part of the NET+Works integrated product family.

Who should read this guide

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

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

- Be familiar with installing and configuring network software and development board systems.
- Have sufficient system or user privileges to do these tasks.
- Have access to a computer system that meets NET+OS hardware and software requirements.

What's in this guide

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

To read about	See
Porting the BSP to a new hardware platform	Chapter 1, "Porting the BSP to Application Hardware"
Hardware dependencies for porting NET+OS to your application hardware	Chapter 2, "Hardware Dependencies"
The board initialization process	Chapter 3, "Initialization Process"
The modes in which NET+OS operates, and how interrupts are handled	Chapter 4, "Processor Modes and Exceptions"
Device driver functions and device definitions	Chapter 5, "Device Drivers"
The linker files provided for sample projects, including the Green Hills Software and NET+OS sections of the linker files	Chapter 6, "Linker Files"
Adding new flash ROM parts to the existing table of supported flash ROM parts	Chapter 7, "Updating Flash Support"
Building a compressed ROM image	Chapter 8, "ROM Image Compression and Decompression"

Conventions used in this guide

This table describes the typographic conventions used in this guide:

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

Related documentation

- *NET+OS Getting Started Guide* explains how to install NET+OS with Green Hills or with GNU tools, and how to build your first application.
- *NET+OS User's Guide* describes how to use NET+OS to develop programs for your application and hardware.
- *NET+OS Application Software Reference Guide* describes the NET+OS software application programming interfaces (APIs).
- *NET+OS BSP Software Reference Guide* describes the board support package APIs.
- *NET+OS Kernel User's Guide* describes the real-time NET+OS kernel services.
- Review the documentation CD-ROM that came with your development kit for information on third-party products and other components.
- Refer to the NET+Works hardware documentation for information appropriate to the chip you are using.

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 the contact information listed in this table:

For	Contact information
Technical support	Telephone: 1 800 243-2333/ 1 781 647-1234 Fax: 1 781 893-1388 Email: tech_support@netsilicon.com
Documentation	techpubs@netsilicon.com
NetSilicon home page	www.netsilicon.com
Online problem reporting	www.netsilicon.com/EmbWeb/Support/forms/bugreport.asp An engineer will analyze the information you provide and call you about the problem.

Porting the BSP to Application Hardware

CHAPTER 1

This chapter describes how to port the Board Support Package (BSP) to your application software. Read this chapter for the following information:

- Understanding the BSP.
- Basic steps in porting the BSP to a new hardware platform.

Understanding the BSP

The BSP consists of the hardware-dependent parts of the real-time operating system (RTOS). The NET+OS BSP that is provided with the NET+Works development kit is designed to work on NET+Works development boards. Most user applications require custom hardware, however, so you need to modify the NET+OS BSP to work correctly on your application hardware.

BSP system requirements

Your development board must meet these requirements to allow porting:

- Flash at CS0
- RAM (32-bit wide) at CS1
- NVRAM at CS3

Basic steps for porting the BSP to application hardware

Table 1 lists the basic steps for porting the BSP to your application hardware. The remainder of the chapter discusses each step in detail.

Step	Action
1	Follow the NET+Works hardware design guidelines
2	Purchase Ethernet MAC addresses from the IEEE
3	Create the debugger initialization files
4	Copy the BSP directories
5	Update the new BSP for the application hardware
6	Debug the new BSP from RAM
7	Debug the initialization code
8	Debug the new BSP from ROM

Table 1: Steps for porting BSP to application hardware

Step	Action
9	Update the power-on self-test (POST) routines
10	Add target-specific device drivers

Table 1: Steps for porting BSP to application hardware

Step 1: Follow the NET+Works hardware design guidelines

When designing your application hardware, follow the NET+Works reference design and hardware design guidelines as closely as possible. Use the same parts as used on the NET+Works development board, especially memory peripherals and Ethernet PHY chips. This will minimize the differences between your application hardware and the development board, reducing the amount of work needed to make the NET+OS BSP work on your application hardware.

In addition, be sure your hardware supports these features:

- JTAG port, which allows you to use an in-circuit emulator (ICE) to debug the hardware and software. This is essential when bringing up a new board.
- Extra serial port that can be used 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 out of 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 it is powered up. The CPU is left in a state in which it cannot be accessed, even with the ICE through the JTAG debugger port. The only way to recover from these situations is to disable flash and use the ICE to download a program into RAM that programs correct firmware into flash.

Step 2: Purchase Ethernet MAC addresses from the IEEE

Each device on a network must have a unique Ethernet MAC address. Your company is responsible for purchasing its own block of addresses from the IEEE.

After you purchase a block of addresses, you must assign a unique number to each board. The addresses are stored in NVRAM or flash ROM.

NetSilicon provides an Ethernet MAC address with each development board, but you need a unique address for your own boards.

Step 3: Create the debugger initialization files

When you use the ICE debugger, you must initialize hardware registers on the board that would normally be set up by the BSP ROM startup code:

- `net4032.dbs.` Command file for the Green Hills Software MULTI debugger, which initializes SDRAM-based NET+Works development boards.
- `net4032bedo.dbs.` File for initializing EDO-based boards.

These files contain debugger commands that are hardware-dependent and write appropriate values to NET+ARM hardware registers. You must create your own initialization command files to configure your application hardware. Be sure to configure the registers — the System Control register, the System Status register, and the Memory Module registers — in the NET+Works development board `.dbs` files. See the NET+Works hardware documentation for the chip you are using.

Note: Initialization must recognize the difference between running normally from flash ROM and running with the debugger from RAM. The BSP makes the distinction by clearing (setting to 0) the CS0 Base Address register, which normally is initialized to 1.

Step 4: Copy the BSP directories

The `src/bsp` directory and its subdirectories contain the code that implements the power-on initialization code, the basic BSP code, and the device drivers. Copy the BSP directories and modify them as follows:

1. Create your own copies of the BSP directories from the source directory.
2. Modify the build files for the `src/bsp` libraries to use different output file names.
3. Test the build files by building the new BSP library.
4. Use the NET+Works utility to create a new application.
5. Modify the build file for the application to link it with the new BSP library.
6. Verify that the application runs correctly on the development board.

Step 5: Update the new BSP for the application hardware

After you modify the BSP directories, you need to update the BSP for your particular application hardware.

1. See Chapter 2, "Hardware Dependencies," for a list of the BSP hardware dependencies and the files associated with them. Review this information and find all areas in the code where differences in the application hardware affect the BSP.
2. Recode these sections of the BSP to support your application hardware.
3. Edit the `bspconf.h` and `devices.c` files, and remove the entries for the devices that are not supported by your application hardware.

Step 6: Debug the new BSP from RAM

To debug code from RAM, you need to use the ICE and download the code through the debugger into the RAM on your board.

Preliminary tasks

1. Rebuild the application you created in "Step 4: Copy the BSP directories" with the new modifications to the BSP.
2. Build the application with the power-on self-test (POST) disabled.
3. Use the functions and routines in Chapter 3, "Initialization Process," to update (initialize) the BSP. Use the debugger initialization files created in "Step 3: Create the debugger initialization files" on page 4 to configure your application hardware.

Debugging the BSP from RAM

Once the BSP is updated, use these debug steps to check that the initialization code and process was followed:

1. Load the application with the MULTI debugger.
2. Set up the debugger to view assembler instructions, then step through the debug image one instruction at a time. This step leaves the program counter (PC) at the beginning of the startup code.
3. 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 appropriately.
   - All interrupts are masked off.
   - The PLL registers are properly programmed for the crystal on your application hardware.
4. Verify that you can read and write RAM on your application board.
5. Check the Control Status register (`0xffb0 0000`) to verify that the board has come up in Big Endian mode.

6 Check the System Status register (`0xffb0 0004`) to verify that the `GEN_ID` field reflects the bootstrap configuration values that the application hardware is trying to set.

7 Step through the startup sequence and verify the initialization process.

Be aware: Be aware that the ICE may lock up when writing to the Chip Select registers, even when the chip select is programmed correctly. If this happens, try setting a breakpoint a few instructions past the instruction that writes to the chip select. Then, let the program run to the breakpoint rather than trying to step into the instruction that writes to the chip select.

Step 7: Debug the initialization code

Debug the initialization code in stages, in the same order as presented in this section:

- `init.s` file
- `ncc_init` routine
- `NABoardInit` routine
- Ethernet driver startup

Debug the init.s file

The `init.s` file performs initialization functions. Step through the code that performs software reset, programs PORTC6 to fix the SDRAM-based board's reset problem, initializes the clock, and programs CS0. Because your debugger command file has already set up CS1, the code in `init.s` should leave it unchanged.

Debug the ncc_init routine

The `ncc_init` routine performs most of the hardware initialization and is most likely to be affected by hardware changes.

The `settings.c` file has a table containing the chip select settings for flash (CS0), RAM (CS1), and NVRAM (CS3) for all chips supported by NET+OS. You can modify this table to customize the BSP rather than hardcoding initialization parameters in `ncc_init`. See the `settings.h` file for the definition of the `NetarmInitData` data members. The `ncc_init` routine reads the chip type (`REVID`) and uses it as an index to the table to get the parameters for initialization.

Verify these system settings:

- The PLL register is set as needed for your system crystal. See the `settings.c` file for register settings based on chip type.
- GPIO registers are programmed for your hardware requirements. Use the registers shown in the table:

Port	Register
A	ffb0 0020
B	ffb0 0024
C	ffb0 0028

  The NET+OS BSP programs the I/O lines to support two serial ports and the LEDs on the development board. PORTC6 is programmed to fix the SDRAM-based board's reset problem.
- The CS1 and CS2 settings are unchanged. These settings should have already been programmed by the debugger command file. Check the Chip Select Option registers and the Chip Select Base Address register. (See the NET+Works hardware documentation for the chip you are using.)
- CS3 is set up to support the NVRAM on your hardware.

Debug the NABoardInit routine

1. Step through the initialization code in `narmbrd.c` to verify that the NVRAM APIs are initialized to support the NVRAM on your application hardware.
2. Verify that an appropriate Ethernet MAC address is generated from the block of addresses you acquired from the IEEE.

3 Call the `NAInitEthAddress` routine to verify that the MAC address is passed to the Ethernet driver.

Debug the Ethernet driver startup

1 Put a breakpoint on the `eth_chip_reset` routine (in `narmenet.c`) and let the program run until you hit the breakpoint.

2 Perform the steps that apply to the PHY you are using (MII or ENDEC):

PHY	Steps
MII	1 Step into the `mii_negotiate` routine (in the `narmnet.c` file) and then into `mii_identify_phy`. Verify that `mii_negotiate` completes successfully (returns 0) and `mii_identify_phy` identifies the PHY on your application hardware. 2 Verify that `mii_check_speed` determines whether you are connected to a 100 Base-T network.
ENDEC	Change `#define MII_PHY(1)` in the `narmenet.c` file to `#define MII_PHY(0)`.

Step 8: Debug the new BSP from ROM

When you debug code from ROM, you use the debugger only to step through the application code fetched from flash.

After you debug the BSP from RAM, build a ROM image (`rom.bin`) of the application and program it into flash. The ROM startup is different than the RAM startup because the initialization code needs to determine the type of RAM on the board, size it, and set up the chip selects to support it. If you know the board's RAM is a fixed type and amount, you can hardcode it.

Be aware: The Green Hills Software components included with NET+OS do not support a hardware breakpoint for the JEENI. The ROM debugging instructions in this section apply *only* to the Raven ICE.

Start the debugger

1. After you build `rom.bin` and load it into flash on your development board, open the `rom.bld` file in the MULTI debugger Build window.
2. Connect the Raven to the board, enable flash, and turn on the power. To enable flash, locate SW4, enable FBANK1, FBANK2, or both, depending on your hardware configuration.

For Green Hills version 3.5 only

In Green Hills version 3.5, Target windows are disabled by default. To enable the Target window and continue the start-debugger procedure, follow these steps:

3. Select **CONFIG**.
4. Select **debugger** in the Options window, then select **More Debugger Options**.
5. Select **Show target windows when connecting to debug server**. (This is the 10th option.)
6. Select **OK**.
7. Select **OK** in the Options window.

Continue with the remaining steps

8. Use the following command to connect the MULTI debugger:

```
ocdserv rlpt1 arm7t -big -rom
```

9. In the Target window, type the following commands at the `ocd>` prompt:

```
ocd> halt
ocd> debug 16
ocd> load
ocd> debug 0
ocd> reg r15 0x0                    (where PC = 0x0)
ocd> reg cpsr 0xd3                  (disable interrupt)
```

10. Open the MULTI debugger window. Set the window to view the assembly instructions by selecting **View → Display Mode → Assembly Only**.

    At this point, you should be able to see the contents of ROM.

Set a hardware breakpoint

1. Open the Breakpoint window. Select **View → Breakpoints** or the Breakpoints icon (a magnifying glass).
2. Click **Hardware**, then **Execute**.
3. Enter one of the following and click **Set**:
   - In the Expression window, enter `Reset_Handler_ROM`.
   - In the Address window, enter the code address.
4. Click **Go** in the Debugger window. The Raven starts loading.

   Wait until the Raven hits the hardware breakpoint.

Debug the init.s file

1. Step through the initialization code in the `init.s` file up to the point where RAM is set up.
2. Verify that the code identifies the type of RAM connected to CS1, and sets up CS1 to support 512 Kbytes of RAM.

Debug the ncc_init routine

1. Step through the code in the `ncc_init` routine that determines the size of RAM connected to CS1.
   - Verify that the RAM is sized correctly and that CS1 is set up properly.
   - Check the Chip Select Option and Chip Select Base Address registers for CS1.
2. Verify that CS2 is set up appropriately for RAM, if RAM is present.

   The `ncc_init` routine tries to determine whether RAM is connected to CS2 and, if so, to identify and size it.
3. Verify that the RAM test passes.

Do not use `print/x *0xffc00010` or `memview 0xffc00010` to see the contents of the NET+ARM registers. Add a variable to the source instead; for example:

```
unsigned long temp=*0xffc00010
```

Then use `print/x temp` in the debugger window to see the contents of the NET+ARM registers.

Hardware breakpoints

After stepping through the initialization code, delete the hardware breakpoint you initially set. You can then set another hardware breakpoint to another function or code address. You can set only one hardware breakpoint at a time.

Step 9: Update the POST routines

The power-on self-test (POST) routines are hardware-dependent. You can do one of the following:

- Build your application with the POST disabled.
- Remove the POST routines that check the hardware that is not supported on your target. Use the remaining POST routines as they are.
- Update the POST to fully support your target.

The POST routines are in the `ncc_post.c` files (see Chapter 3, "Initialization Process").

When you have updated the POST, rebuild your application with POST enabled. Use the MULTI debugger to test the POST.

Step 10: Add target-specific device drivers

You can implement device drivers to support special hardware on your target. By implementing a device driver, you can use standard C programming language I/O functions, such as `read` or `write`, to access or control the hardware on your target. Writing a device driver is useful if the device is used for stream-oriented I/O.

See Chapter 5, "Device Drivers," for a description of the NET+OS device driver requirements and an explanation of adding a device driver to the RTOS.

Hardware Dependencies

CHAPTER 2

This chapter discusses the hardware dependencies for porting NET+OS to application hardware:

- DMA channels
- Ethernet PHY chip
- Ethernet MAC address
- ENI controller
- Serial ports
- Software watch-dog
- Big Endian mode
- System clock
- System timers
- GPIO pins
- Interrupts
- Chip selects
- Cache
- Memory map

DMA channels

There are ten DMA channels; Table 2 describes how each is used when porting NET+OS.

Channel	Used by	What happens
1	Ethernet driver	Moves data from the Ethernet receiver to memory. The Ethernet driver code is in the `narmenet.c` file.
2	Ethernet driver	Moves data from memory to the Ethernet transmitter.
3	ENI driver	Receives data. If you are not using the ENI driver, this channel is used by the parallel port.
4	ENI driver	Transmits data. If you are not using the ENI driver, this channel is used by the parallel port.
5 and 6	Parallel driver	Moves output data from memory to the parallel ports. The parallel driver code is in the `narmpara.c` file.
7 and 8	HDLC driver	Receives data.
9 and 10	HDLC driver	Transmits data.

Table 2: DMA channel use when porting NET+OS

Ethernet PHY chip

The NET+ARM chip can support ENDEC PHY and MII PHY as indicated by `GEN_ID` bits 6 and 7:

- If both bits are set, the Ethernet driver configures itself to support a SEEQ 80C24 ENDEC PHY. This code is in the `eth_chip_reset` routine.
- If one bit is set but the other is not, or neither bit is set, the Ethernet driver tries to identify and configure the MII PHY.
- The current BSP code supports four MII PHYs:
  - FastCat by Lucent Technologies (also known as the 3-volt Enable PHY)
  - LXT970 by Level One

– LXT971A by Intel
– AM79C874 by AMD

The `mii.c` file contains the code that determines which MII PHY is present, and configures it. Other MII PHYs can be supported by modifying this code.

Ethernet MAC address

You must supply the Ethernet drive with a unique MAC address. The routines provided in the BSP are for use only with the NET+Works development board. Your company must purchase its own block of MAC addresses from the IEEE.

Generating a unique MAC address

When you have purchased the block of MAC addresses:

1. Implement code to generate a unique MAC address from your block of addresses.
2. Call `NAInitEthAddress` to pass the MAC address to the Ethernet driver — *before* you load the TCP/IP stack.

In the BSP, the `NABoardInit` routine calls `NAResolve_MAC` to generate a unique MAC address before the address is passed to the Ethernet driver. If necessary, rewrite `NAResolve_MAC`, which is in the `narmsrln.c` file, to generate MAC addresses from your block of addresses.

ENI controller

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

To use ENI, you must disable the parallel driver. Use one of these two methods:

- **Recommended method.** Undefine `BSP_INCLUDE_PARALLEL_DRIVER` in the `bspconf.h` file.

- **Alternate method.** Remove the parallel driver entries from the device driver table in the `devices.c` file.

Serial ports

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

To use the SPI controller, you must disable the serial driver. Use one of these two methods:

- **Recommended method.** Undefine `BSP_INCLUDE_SERIAL_DRIVER1` and `BSP_INCLUDE_SERIAL_DRIVER2` in the `bspconf.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 watch-dog

The BSP normally leaves the software watch-dog disabled.

The `NAReset` routine in the `nareset.c` file uses the software watch-dog to reset the system. In this file, the software watch-dog is enabled.

Big Endian mode

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 PLLTST* signal indicates the choice. The frequency of the selected source affects the BSP timing.

The system clock API consists of four compiler definitions and two functions, used throughout the BSP. The definitions are explained in the remainder of this section as well as in the sysClock.h file.

PLLTST_SELECT

Determines the clock source to be used. PLLTST_SELECT is the address input to the SYSCLK signal multiplexer. SYSCLK has two possible sources:

- TTL clock input applied to the XTAL1 pin
- Crystal oscillator and PLL (phase lock loop) circuit

Set PLLTST_SELECT to either of the following:

- SELECT_THE_XTAL1_INPUT
- SELECT_THE_CRYSTAL_OSCILLATOR_INPUT

The BSP defines this directive as the usesTheInternalOscillator function in the settings.c file, and references the NetarmInitData table in the settings.c file to return a value based on the usesInternalOscillator data marker from the appropriate table entry.

XTAL1_FREQUENCY

Indicates the frequency of the TTL clock input to the XTAL1 pin. If PLLTST_SELECT is set to SELECT_THE_XTAL1_INPUT, this signal generates the internal SYSCLK signal.

CRYSTAL_OSCILLATOR_FREQUENCY

Indicates the frequency of the crystal oscillator. If PLLTST_SELECT is set to SELECT_THE_CRYSTAL_OSCILLATOR_INPUT, this signal generates an input to the phase lock loop and, in conjunction with PLL_CONTROL_REGISTER_N_VALUE, determines the frequency of the internal SYSCLK signal.

PLL_CONTROL_REGISTER_N_VALUE

Indicates the *N* factor used in the divide-by circuits of the NET+ARM chip clock generation section. This value is stored in the PLLCNT field in the PLL Control

register (see the NETR+Works hardware documentation for the chip you are using). The *N* factor multiplies or divides clock sources. The range of values is 0 through 15; the suggested values are based on chip type and revision.

The BSP defines this directive as the `getPllValueBasedOnChipType` function in the `settings.c` file, and references the `NetarmInitData` table in the `settings.c` file to return the PLL count number from the appropriate table entry.

System timers

The code that supports the system timers is in the `bsptimer.c` file.

Timer 1: System heartbeat clock

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 `bspconf.h` file. This value should be between 1 and 1000, and determines the heartbeat rate. A value of 100, for example, provides a heartbeat rate of one every 10 milliseconds.

Timer 2: Parallel driver support

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 `narmpara.c` file will be affected.

Timer 2 normally is programmed to have a period of 217 microseconds.

General purpose I/O pins

There are 24 genearl purpose I/O (GPIO) pins, divided into three ports of 8 pins each. The individual pins can be configured for GPIO or to support serial port functions and external interrupts.

The initialization code in the ncc_init.c file configures all pins in PORTA and PORTB to support the serial ports. The code configures the pins in PORTC as described in Table 3.

Pin	Use
PC0	Connected to the Ethernet PHY on the development board. The BSP programs this pin to be an interrupt input, but never enables the interrupt and does not install an interrupt service routine (ISR) for it. The PHY can be programmed to assert this signal whenever the Ethernet cable is connected or disconnected.
PC1	Controls the green LED on the development board. The BSP programs this pin as an output, and initially sets it low to turn on the LED.
PC2	Controls the red or yellow LED on the development board. The BSP programs this pin as an output, and initially sets it high to turn off the red LED.
PC3	Enables the IEEE 1284 parallel ports on the development board. The BSP programs this pin as an output, and sets it low to enable the ports.
PC4	General-purpose input.
PC5	Transmitter clock for serial port B.
PC6	General-purpose input.
PC7	Transmitter clock for serial port A.

Table 3: PORTC pin use for BSP

LED routines

The code that blinks the LEDs is in the romstart.c and narmled.c files. The LED routines are called by routines in the BSP. To disable the LED function, do one of the following:

- Turn off the NETOS_LED switch in narmled.h.
- Rewrite the code in romstart.c and narmled.c.

The code in the `init.s`, `romstart.c`, `ncc_init.c`, and `ncc_post.c` files uses the LEDs to indicate error conditions. This code will need to be modified if PC1 and PC2 are used for different purposes.

Interrupts

Table 4 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 (NET + 50, NET + 20M, NET + 40, NET + 15)
29 (DMA 3)	ENI FIFO receive packet interrupt
28 (DMA 4)	ENI FIFO transmit packet interrupt
27 and 26 (DMA 5 and 6)	Not used
25 (DMA 7)	HDLC driver channel 1 receive frame interrupt
24 (DMA 8)	HDLC driver channel 1 transmit frame interrupt
23 (DMA 9)	HDLC driver channel 2 receive frame interrupt
22 (DMA 10)	HDLC driver channel 2 transmit frame interrupt
21–17 (ENI ports 1–4 and ENET RX)	Not used
16 (ENET TX)	Ethernet driver transmit interrupt
15 (SER 1 RX)	Serial driver port A receive interrupt
14 (SER 1 TX)	Serial driver port A transmit interrupt
13 (SER 2 RX)	Serial driver port B receive interrupt
12 (SER 2 TX)	Serial driver port B transmit interrupt
11–6	Not used
5 (Timer 1)	System clock tick interrupt
4 (Timer 2)	Not used
3–0 (PORTC)	Not used

Table 4: Interrupt level use in BSP

Chip selects

There are five chip selects — CS0 through CS4. The next sections describe how BSP uses each chip select.

CS0

The initialization code assumes CS0 is connected to ROM. The code in the `init.s` and `ncc_init.c` files senses whether 16-bit or 32-bit flash is connected to CS0, and then configures CS0 to support 1 Mbyte of 16-bit flash or 2 Mbytes of 32-bit flash. The ROM sizes are hardcoded in `init.s` and `ncc_init.c`.

By default, ROM is mapped to address `0x0200 0000`. The ROM base address is controlled by the BSP_ROM_BASE constant in `bspconf.h` and by hardcoded values in `init.s` and `ncc_init.c`.

The default configuration of the development board is16-bit mode. To configure the board for 32-bit mode:

1. Make sure CS00 of SW3 and CS01 of SW4 are OFF.
2. Move jumpers P27, P28, and P29 from pin 2-3 position to pin 1-2 position.

CS1

The initialization code assumes that at least 512 Kbytes of RAM is connected to CS1. The code in `init.s` automatically detects and configures CS1 to support:

- 32- or 16-bit SDRAM
- 32- or 16-bit EDO RAM
- 32- or 16-bit Fast Page RAM

The code in `ncc_init.c` determines how much RAM is attached to CS1, and sets the size in the chip select accordingly. The code tests for RAM from 512 Kbytes to 16 Mbytes on CS1, and maps RAM to address `0x0` by default.

CS2

If CS2 is connected to RAM, it must be the same type of RAM as connected to CS1. The initialization code sizes the RAM connected to CS2 independently of that connected to CS1. The RAM is configured to start immediately after the RAM controlled by CS1, so a contiguous block of RAM is created. All CS2 initialization is done by the code in `ncc_init.c`.

CS3

The code in `ncc_init.c` assumes that CS3 is connected to 8-bit EEPROM if NVMEM is selected. The starting address and size of the device are controlled by the `BSP_NVRAM_BASE` and `BSP_NVRAM_SIZE` constants in `bspconf.h`. If NVMEM is not selected, the initialization code disables CS3.

CS4

The code in `ncc_init.c` assumes that CS4 is not used, and disables CS4 by setting the valid bit to 0 in the Chip Select Base Address register.

Memory Module Configuration register

The Memory Module Configuration register (`0xffc0 0000`) is configured by the code in `init.s` to set DRAM refresh with a 15-microsecond period using an external DRAM multiplexer.

Cache

During initialization, the `NABoardInit` routine in the `narmbrd.c` file enables cache — on processors that support it — by calling `NACacheSetDefaults`.

If the processor is executing code out of instruction cache, ICEs will not work. `NACacheSetDefaults` leaves instruction cache disabled for applications under development (that is, run from the debugger).

Disable cache completely to debug firmware in ROM with an ICE.

Memory map

NET+OS has the same memory map on all NET+Works development boards:

- Addresses from `0xf0000000` to `0xffffffff` are reserved for devices internal to the NET+ARM chip.
- RAM on CS1 and CS2 is mapped from address `0x0` to `0x01ffffff`.
- ROM on CS0 is mapped from address `0x02000000` to `0x021fffff`.
- NVRAM on CS3 is mapped from address `0x03000000` to `0x03001fff`.
- Cache is supported in software by creating four regions in memory:

Region	Addresses
0	0x00000000 – 0x07ffffff
1	0x08000000 – 0x09ffffff
2	0x0a000000 – 0x0bffffff
3	0x0c000000 – 0x0fffffff

The NET+OS BSP sets up the cache regions according to chip type:

- The NET+40 caches only the ROM-based instructions in region 2.
- The NET+50 caches instructions and data from ROM or RAM. The BSP sets up regions 1 and 2 for cache.

The same memory map is used on all processors for software compatibility, which allows you to write software that takes advantage of cache if the target supports it and runs correctly on targets that do not support cache.

Figure 1 shows the NET+OS memory map with cache enabled (for the NET+50).

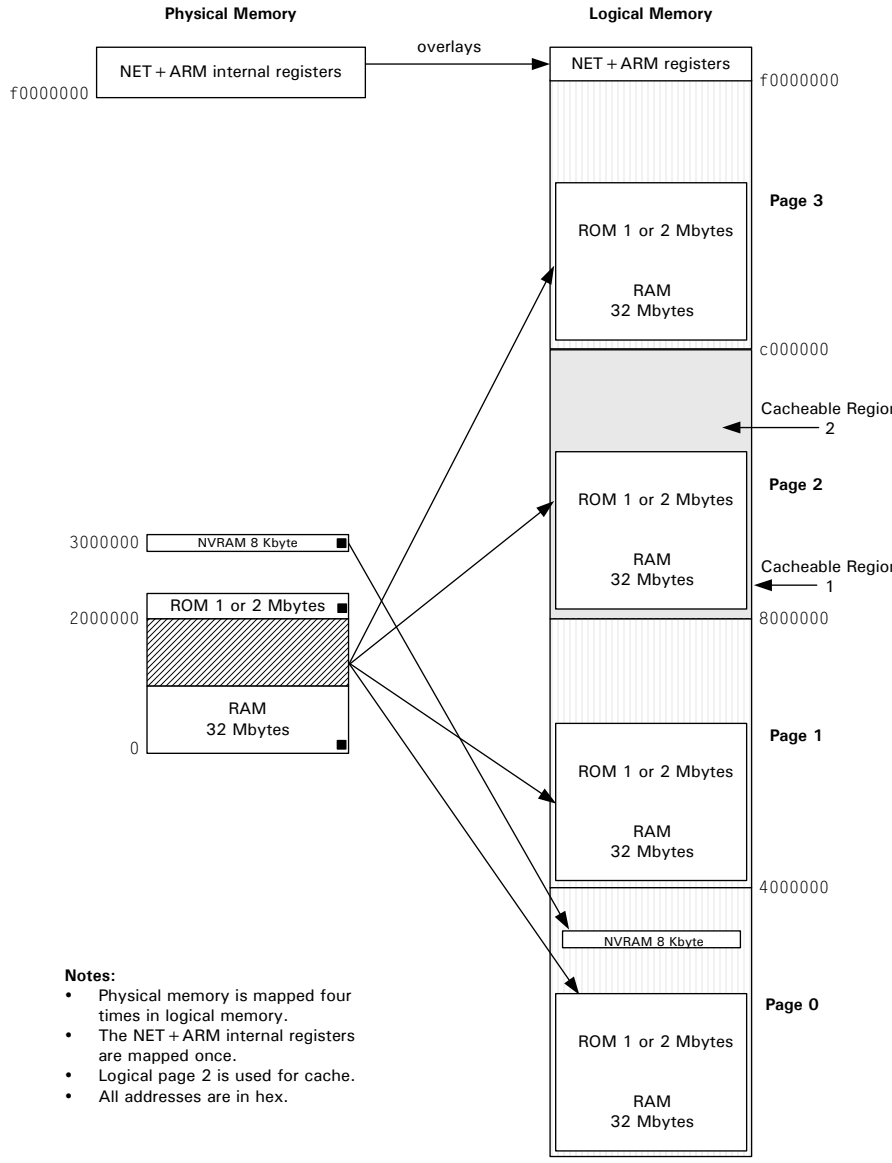


Figure 1: NET + OS memory map

Remapping

Page 0 contains a slot for up to 32Mbytes of RAM (using CS1 and CS2) at address `0x0` through `0x1ffffff`. Either 1 or 2 Mbytes of flash ROM on CS0 begins at `0x2000000`, and 8 Kbytes of NVRAM starts at `0x3000000`.

The ROM and RAM spaces are remapped on pages 1, 2, and 3. For example:

- Physical address `0x100`, which is RAM, can be accessed at `0x4000100`, `0x8000100`, and `0xC000100`.
- Physical address `0x2000100`, which is flash ROM, can be accessed at `0x6000100`, `0xA000100`, and `0xE000100`.
- Physical address `0x3000100`, which is NVRAM, can be accessed only at that address — `0x3000100`.

Remapping memory allows the NET+ARM to select which data or instructions get placed in cache.

Address mapping

The linker command files generated for each application set up an address map so instructions are executed out of the memory region that has been set up for instruction cache, and normal data is located in non-cache memory.

If the address map changes, you need to change:

- The constants in `bspconf.h` and `nacache.h`.
- CS0, because the ROM sizes are hardcoded in `init.s`.
- The `rom.lx`, `romzip.lx`, and `debug.lx` files for each application.

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

Initialization Process

CHAPTER 3

This chapter describes the board initialization process, which includes the following steps:

- Hardware reset
- ROM startup
- C library startup
- NABoardInit routine
- DHCP state transition hooks

Hardware reset

At powerup, the NET+ARM chip reads the state of all external address lines to determine how to configure itself. All NET+ARM address lines have input current sources so that, if left unchanged, the address lines will be in the high state.

NET+OS requires these configuration settings:

Configuration	Address	Mode	State
Endian	A27	Big Endian	High
CPU bootstrap	A28	ARM CPU enabled	High

Table 5: NET+OS hardware reset configurations

At powerup, the NET+ARM chip:

- Resets all internal devices.
- Disables all chip selects except CS0.
- Configures CS0 to map the contents of the ROM to all memory.
- Disables cache.

At reset, the ARM processor core:

- Enables ARM mode.
- Sets `supervisor` mode.
- Disables IRQ and FIRQ interrupts.
- Sets the program counter to 0.
- Starts executing at 0 (512 clocks after reset is deasserted).

ROM startup

If the initialization code is running from ROM, the first code executed is in the `reset.s` file at entry. This is to correct problems with 32-bit flash.

The reset vector then jumps to the `Reset_Handler_ROM` routine in the `init.s` file. The `Reset_Handler_ROM` routine starts the initialization process:

1. Checks whether code execution is in debug mode (running from RAM). If so, conditionally branches to `Reset_Handler`.
2. Relocates the program counter to the correct area of memory for flash.
3. Sets the processor mode to `SVC` and disables interrupts.
4. Does a software reset to put most of the NET+ARM chip into a known state.
5. Configures CS0 to the correct values for the installed ROM. The port size is maintained (according to the bootstrap resistors).
6. Runs a series of tests to determine the type of RAM connected to CS1.

   The NET+15, NET+40, NET+50, and NET+20M support Fast Page, EDO, synchronous DRAM (SDRAM), and static RAM (SRAM). Both 16- and 32-bit bus sizes of all types of RAM are supported.

   If no RAM is detected, `crash_for_debug` is called.
7. Programs CS1 to support 512 Kbytes of RAM.

   The real size is eventually set in `ncc_init`.
8. Sets up the temporary `SVC` stack for a call to `ncc_init`.
9. Calls the `ncc_init` to continue the hardware setup.
10. Creates stacks for all processor modes.
11. Calls `romstart`.
12. Calls the C library initialization routine.

ncc_init routine

The `ncc_init` routine continues the hardware initialization process:

1. Sets up the following registers:
   - System Control register (`0xffb0 0000`)
   - Interrupt Enable register (`0xffb0 0030`)
   - PLL register (`0xffb0 0008`)
   - PORTA, PORTB, and PORTC registers

     PORTA and PORTB are set up to support the serial ports on the development board. The PORTC register is set up to support the two LEDs on the development board. In addition, the parallel port output buffers are enabled.
2. Sets up the chip select regions as follows:
   - Sets up and sizes CS0.
   - Determines the amount of RAM available, and configures CS1 and CS2 accordingly. The initialization code assumes that RAM in CS2 matches the type of RAM in CS1.
   - Programs CS3 to support NVRAM if it is selected in `appconf.h`.

   CS4 currently is disabled.
3. Tests RAM.

   While it is not required, NetSilicon strongly recommends performing a RAM test. The test takes less than 30 seconds to execute.

romstart routine

The `romstart` routine performs initialization and the power-on self-test (POST) for the NET+ARM chip. Most of the test is in the `ncc_post` routine in `ncc_post.c`.

The `romstart` routine:

1. Duplicates the ROM-based vector table in RAM address 0.
2. Copies the ROM-based initial values into the RAM-based initialized data structures.
3. Clears uninitialized RAM.
4. Initializes the IRQ tables (see `irqreg.c`).

5 Bypasses the ROM checksum; that is, SKIP_CHECKSUM_TEST is defined.

6 Resets the MII PHY, if it exists, in preparation for the test.

7 Performs the ncc_post test if the POST is enabled in the root.c user application (APP_ROOT).

The ncc_post sequence includes:

- Registering the POST IRQ handlers
- Creating a new RAM-based vector table
- Testing the DMA RAM area of the chip
- Using memory-to-memory data transfers to test the DMA system
- Testing the parallel ports by writing data to the ports using DMA and then reading it back. (The test does not require a loopback connector and does not write any data to printers connected to the parallel ports.)
- Testing the Ethernet MAC by placing it into loopback mode, then transmitting and receiving packets. (No loopback connector is needed and no data is sent out the Ethernet PHY.)
- Testing the GEN module to ensure it can generate interrupts
- Testing the MII interface between the Ethernet MAC and PHY
- Testing cache RAM (if the chip supports cache)

8 Re-initializes the IRQ tables (overwrites the POST test values).

9 Initializes serial ports, parallel ports, and the Ethernet media interface.

C library startup

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

1 Copies initialized data from ROM to RAM, and resets uninitialized data to 0 (zero).

2 Calls tx_kernel_enter to initialize the I/O system.

3 Calls the main routine.

main routine

The `main` routine:

1. Calls `SetupVectorTable` to set up the system vector table.

   The IRQ (`0x38`) is set up by the ThreadX RTOS library.

2. Calls `NABoardInit` in `narmbrd.c` to complete the hardware initialization.

3. Calls `DDIFirstLevelInitialize` to perform preliminary initialization of the device drivers, if necessary.

   `DDIFirstLevelInitialize` calls the `deviceEnter` functions defined for each channel in the device driver table in `devices.c`.

4. Calls `tx_kernel_enter` to start the kernel.

   After the kernel has initialized, it calls `tx_application_define` to create the root thread. The stack size and priority for the root thread are set by constants defined in each application's `appconf.h` file. The root thread starts by executing the function `netosStartup`.

The `main`, `DDIFirstLevelInitialize`, `tx_application_define`, and `netosStartup` routines are in the `bsproot.c` file.

NABoardInit routine

The `NABoardInit` routine in the `narmbrd.c` file:

1. Reads the chip revision and stores it in the `g_NAChipRevision` global variable.
2. Shuts down all DMA channels, as well as the Ethernet controller and all serial ports.
3. Masks all interrupts.
4. Initializes the NVRAM read and write API.
5. Calls `NAResolve_MAC` to generate a unique Ethernet MAC address.

   This routine is specific to the development board and will need to be rewritten for your hardware. `NAResolveMAC` is in `narmsrln.c`.

6 Calls NAInitEthAddress to pass the MAC address to the Ethernet driver.

7 Calls NACacheSetDefaults to initialize cache (if the chip supports it).

netosStartup routine

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

1 Loads default NVRAM parameters from the appconf.h file.

2 Calls DDISecondLevelInitialize to load the system device drivers.

This function calls the deviceInit function defined for each device in the device table.

3 Creates the TCP timer thread to trigger calls to tcp_down_function, which calls the TcpDown application each second.

4 Calls NADialog to prompt the user with a configuration dialog box.

5 Calls netosConfigStdio.

If APP_STDIO_PORT is defined in appconf.h as a valid device name, netosConfigStdio redirects stdin, stdout, and stderr to the indicated device.

6 Calls netosStartTCP to create and initialize resources (for example, semaphores and memory), and to start up all threads required by the TCP/IP stack.

IP parameters are loaded into the stack or DHCP is invoked.

7 Calls TcpDown once every tick until the TCP/IP stack is initialized.

8 Calls applicationStart after the TCP/IP stack has started.

The netosStartup, DDISecondLevelInitialize, and netosConfigStdio routines are in the bsproot.c file.

The NADialog routine is in the dialog.c file.

DHCP state transition hooks

Two functions are included in the `dhcpstub.c` file to allow BSP modifications to alert the user application layer of the major DHCP state transitions (loss of IP address lease and new IP address leases).

DhcpNowBound function

`DhcpNowBound` is called when the local DHCP client obtains an IP address lease from a DHCP server. This stub routine can be customized to support specialized processing due to obtaining or changing IP parameters using DHCP.

This function is called after the DHCP ACK packet is received from DHCP server.

DhcpLostLease function

`DhcpLostLease` is called when the local DHCP client fails to extend an IP address lease from a DHCP server. A unit that does not have an IP address is limited to UDP broadcasts only. If a client fails to extend a DHCP lease, there may be a fatal network failure.

Use this function for an orderly shutdown of critical services that require constant network communication.

Processor Modes and Exceptions

CHAPTER 4

This chapter discusses the modes in which NET+OS operates, including how interrupts are handled.

Processor Modes

The processor supports six modes:

- `user`
- `supervisor`
- `abort`
- `undefined`
- `interrupt (IRQ)`
- `fast interrupt (FIRQ)`

NET+OS operates in `supervisor` and `interrupt` modes. All threads and the kernel scheduler operate in `supervisor` mode.

Hardware interrupts cause the processor to switch to `interrupt` mode.

The IRQ handler switches back to `supervisor` mode before calling 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 is temporarily halted; for example, to service an interrupt. Each exception causes the ARM processor to save some state information, then jump to a location in low memory. Low 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 the memory location, which implements a 32-bit jump to an interrupt service routine (ISR).

Table 6 shows the vector address for each exception type:

Exception	Vector address
Reset	0x00000000
Undefined instruction	0x00000004
Software interrupt	0x00000008 (Not used by NET + OS)
Pre-fetch abort	0x0000000c
Data abort	0x00000010
Interrupt request (IRQ)	0x00000018
Fast interrupt request (FIRQ)	0x0000001c

Table 6: Exception vector table

NET+OS considers the following to be fatal errors: pre-fetch aborts, data aborts, undefined instructions, and fast interrupts. Although NET+ARM does not allow fast interrupts to be triggered by external devices, the watch-dog timer can be programmed to trigger a fast interrupt.

The default FIRQ handler normally calls `netosFatalError` with an error type of `NETOS_ERR_FIRQ`.

IRQ handler

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

- 23 interrupt signals support devices inside the chip
- 4 interrupt signals support external devices
- 5 interrupt signals are not used, and are considered *reserved*.

Software can selectively enable or disable any of the 32 interrupt signals.

An IRQ handler is provided as part of the BSP. An IRQ is generated when one or more devices assert their interrupt signal. The IRQ handler reads the Interrupt Status register and determines which devices need to be serviced.

The IRQ handler has a prioritized interrupt scheme. If more than one device is requesting 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. This allows the device's service routine to be interrupted if a higher priority device requests service.

Servicing interrupts

The NET+OS IRQ handler uses the following procedure to service an interrupt:

1. A device requests service by asserting its interrupt signal.
2. The NET+ARM chip latches the request into the Interrupt Status register — Raw (ISRR). Once the signal has been latched, the chip generates the interrupt even if the device stops asserting its interrupt line.
3. When corresponding bits in the Interrupt Enable register and the ISSR are set, the chip asserts the IRQ signal to the NET+ARM CPU.
4. The NET+ARM CPU switches to IRQ mode and jumps to the IRQ handler when the IRQ signal is asserted while interrupts are enabled.
5. The IRQ handler calls `NAIrqHandler` in `na_isr.c`, which calls `NAGetInterruptLevel` to find the highest priority interrupt.
6. In a loop, `NAIrqHandler` masks all lower priority interrupts, enables interrupts, and then calls the function registered during the `NAInstallIsr` call.

   When this routine completes, interrupts are first disabled, then enabled for the next highest pending interrupt. The loop repeats until all interrupt levels have been serviced.

Changing interrupt priority

You can change the interrupt priority level by changing the order of the `NAInterruptPriority` array in the `na_isr.c` file. This list is ordered from lowest to highest priority. Each priority can be specified only once.

Incorrect ordering is treated as a fatal error (that is, calls `crash_for_debug`), which halts the system.

Interrupt sources and default priorities

Table 7 lists the supported interrupt sources, their bit positions in the Interrupt Enable register, and the default NET+OS priority for each (as specified in the `NAInterruptPriority` array in the `na_isr.c` file). Interrupt sources with a higher-numbered priority level can interrupt the service routines of devices with lowered-numbered priority levels.

Interrupt source	Bit	Priority
DMA1 (Ethernet receiver)	31	31
DMA2 (Ethernet transmitter)	30	30
DMA3 (Parallel port 1/ENI FIFO receiver)	29	29
DMA4 (Parallel port 2/ENI FIFO transmitter)	28	28
DMA5 (Parallel port 3)	27	27
DMA6 (Parallel port 4)	26	26
DMA7 (Serial port 1 receiver)	25	25
DMA8 (Serial port 1 transmitter)	24	24
DMA9 (Serial port 2 receiver)	23	23
DMA10 (Serial port 2 transmitter)	22	22
Parallel port 1	21	21
Parallel port 2	20	20
Parallel port 3	19	19
Parallel port 4	18	18
Ethernet receive	17	17
Ethernet transmit	16	16
Serial port 1 receive	15	15
Serial port 2 receive	14	14
Serial port 1 transmit	13	13
Serial port 2 transmit	12	12

Table 7: Interrupt sources

Interrupt source	Bit	Priority
Reserved	11 – 7	11 – 7
Watch-dog	6	6
Timer 1	5	5
Timer 2	4	4
Port C PC3	3	3
Port C PC2	2	2
Port C PC1	1	1
Port C PC0	0	0

Table 7: Interrupt sources

Interrupt service routines

The IRQ handler calls interrupt service routines (ISRs) to service interrupts generated by external devices. ISRs can be implemented as standard C functions — they must clear the interrupt condition (usually by acknowledging it) and service the interrupt, and then can return as a standard C function.

Interrupts are enabled for higher priority interrupt levels when the ISR is called, so one ISR may be interrupted by another ISR with a higher priority.

An ISR can be installed by calling `NAInstallIsr`. After this routine returns, the ISR is installed and the interrupt associated with it is enabled. An ISR can be disabled and removed by calling `NAUninstallIsr`, which disables the interrupt and uninstalls the ISR handler.

FIRQ handlers

A FIRQ is a higher priority interrupt than an IRQ, which allows an IRQ to be interrupted by a FIRQ.

NET+OS does not use FIRQs. The default handler installed by the BSP treats a FIRQ exception as a data error (that is, calls `call_for_debug`), which halts the system.

You can program the watch-dog timer and the two general-purpose timers to generate a FIRQ interrupt. Enable these interrupts by setting the corresponding bits in the Interrupt Enable register. (See the explanations of the System Control register, Timer 1 and Timer 2 Control registers, and Interrupt Enable register in the NET+Works documentation for the chip you are using.)

The Interrupt Enable and Interrupt Status registers function identically for both IRQ and FIRQ levels. Because NET+OS does not use FIRQs, the IRQ handler does not contain special handling for FIRQs; rather, the IRQ handler dispatches FIRQs according to their interrupt source default priorities.

Installing a FIRQ handler

To install a FIRQ handler, follow these steps:

1. Install an application FIRQ handler by writing its address to memory location `0x0000003C`.
2. Enable the FIRQs in the Interrupt Enable register.
3. Modify the IRQ handler routine to exclude the FIRQs from being dispatched with the IRQs.

   The IRQ handler code is in `na_isr.c`, `reset.s`, and `init.s`.

Device Drivers

CHAPTER 5

This chapter provides information about device drivers and device definition.

Device table

NET+OS integrates device drivers with the low-level I/O functions provided in the Green Hills Software system library. Each entry in the `deviceTable` array in the `devices.c` file defines a device supported by the system.

Adding devices

To add new devices, add new entries to the `deviceTable` array. When a device has been added to the device table, the application programs can access it through the standard C programming language I/O functions — `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:

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

deviceInfo structure fields

Table 8 defines the fields in the `deviceInfo` structure.

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

Table 8: deviceInfo structure fields

Device driver functions

These device driver functions are in the `deviceInfo` structure:

- `deviceEnter`
- `deviceInit`
- `deviceOpen`
- `deviceClose`
- `deviceConfig`
- `deviceRead`
- `deviceWrite`
- `deviceIoctl`

Each function is described in this section.

deviceEnter

First-level initialization function for a device table.

Format

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

Arguments

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

Procedure

When the C library initializes its I/O functions, `deviceEnter` is called for each entry in the device table. This function is called only once for each channel, and performs the basic initialization needed by the device driver. This function is called before the kernel has started (that is, kernel services are not available at this time).

Return values

None.

deviceInit

Second initialization function for the device channel.

Format

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

Arguments

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

Procedure

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

Return values

None.

deviceOpen

Called when an application opens a channel. `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.

Format

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

Arguments

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

Procedure

When `deviceOpen` is called, the driver does the following:

1. Checks that the channel number is valid, the channel is open, and 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. Does any other initialization required by the device.
4. Returns `EXIT_SUCCESS`.

Return values

EBUSY
EINVAL
ENOENT
ENOMEM
EROFS
EXIT_SUCCESS

deviceClose

Informs the device driver that the application is closing its session.

Format

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

Arguments

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

Procedure

When `deviceClose` is called, the driver does the following:

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. Sets the channel semaphore, or returns `EBUSY` if the semaphore is set already.
3. Updates internal flags to indicate the session has been closed.
4. Does any other processing necessary.
5. Returns `EXIT_SUCCESS`.

Return values

`EBADF`
`EBUSY`
`EXIT_SUCCESS`

deviceConfig

Passes configuration information to the device.

Format

```
int deviceConfig (int channel, void *config);
```

Arguments

Argument	Description
channel	Channel number as set in the channel's device table entry.
config	Pointer to device-dependent configuration data.

Procedure

When `deviceConfig` is called, the driver does the following:

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

   If an error condition is detected, the driver returns an error without sending any information.
2. Sets the channel semaphore, or returns `EBUSY` if the semaphore is set already.
3. Reconfigures the device.
4. Releases the semaphore.
5. Returns `EXIT_SUCCESS`.

Return values

`EBADF`
`EBUSY`
`EINVAL`
`EXIT_SUCCESS`

deviceRead

Reads data from the device to the caller's buffer.

Format

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

Arguments

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

Procedure

When `deviceRead` is called, the driver does the following:

1. Sets `*bytesRead` to zero.
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 is set already.
5. Performs as follows if no data is available:
   - **In blocking mode.** Waits until some data is received.

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

8 Releases the channel semaphore.

9 Returns a completion code.

Return values

EAGAIN
EBADF
EBUSY
EINVAL
EIO
EXIT_SUCCESS

deviceWrite

Writes a buffer of data to a device.

Format

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

Arguments

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

Procedure

When `deviceWrite` is called, the driver does the following:

1. Sets `*bytesWritten` to zero.
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 is set already.
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 only to blocking mode.* If an error condition is detected, aborts the transmission and returns an appropriate completion code.

Note: If there is more data in the caller's buffer and more buffers, repeat steps 5 through 7 until there is no more data.

8. Updates `*bytesWritten` to indicate the number of bytes transmitted.

9 Releases the channel semaphore.

10 Returns a completion code.

Return values

EAGAIN
EBADF
EBUSY
EINVAL
EIO
EXIT_SUCCESS

deviceIoctl

Sends commands to the device.

Format

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

Arguments

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

Procedure

When `deviceIoctl` is called, the driver does the following:

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

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

Return values

`EBADF`
`EBUSY`
`EINVAL`
`EXIT_SUCCESS`
Other codes as defined by device type and command type.

Modifications to the Green Hills system library

The Green Hills system library has been modified to support NET+OS device drivers. The changes to the system library allow applications to use standard C I/O calls to access NET+OS devices.

The system library was modified using the Green Hills `ind_io.c` source file, and then rebuilt. You can make additional modifications to `ind_io.c`, to add support for a file system or for additional device drivers.

Making modifications

In Green Hills version 2.1

The NET+OS version of the library is in the `ghsrcs` directory. The original Green Hills version should be in the `green\libsrc` directory.

To add to the system library:

1. Modify the NET+OS version of `ind_io.c`.
2. Copy the modified version into `\green\libsrc`.
3. Build a new system library using the instructions in the Green Hills Software *Embedded ARM/Thumb Development Guide*.

In Green Hills version 3.5

The NET+OS version of the library is in the `ghsrcs` directory. The original Green Hills version can be in one of three directories:

- `GHS\arm35`
- `NETOS5_GH35\Green Hills`
- `GHS\arm35\libsys`

Linker Files

CHAPTER 6

This chapter describes the linker files provided for sample projects, including both the Green Hills software and NET+OS sections of the linker files.

ARM linker

The ARM linker takes one or more object modules and combines them into a single executable output module. The relocatable sections of each module resolve the module's external text and data reference with the modules that contain the required text and data.

Linker files provided for sample projects

Four linker files are provided for each sample project:

- `debug.lx`, for generating an executable file for debugging
- `rom.lx`, for generating an executable file for the `rom.bin` image
- `romzip.lx` and `ramimagezip.lx`, for generating a compressed binary image

Linker files have both a Green Hills section and a NET+OS section.

Green Hills section of the linker files

Table 9 summarizes the Green Hills section of the linker files.

Section	Description
picbase	Base of the text sections, relocatable in `-pic` mode
text	Text section
syscall	Syscall section, for host I/O under MULTI
fixaddr/fixtype	For PIC/PID fixups
rodata	Read-only data
romdata	ROM image of `.data`
romsdata	ROM image of `.sdata`
secinfo	Section information section, used by the startup code

Table 9: Green Hills linker file sections

Section	Description
pidbase	Base of the data sections, relocatable in `-pid` mode
sdabase	Base of the small data area section pointer
sbss	Small BSS (zeroed data) section
sdata	Small data section
data	Non-zeroed writeable data section
bss	Zeroed data section
heap	Heap, grows upward
stack	System stack, grows downward

Table 9: Green Hills linker file sections

NET+OS section of the linker files

Table 10 summarizes the NET+OS section of the linker files.

Section	Description
netosstack	Stack for each processing mode; grows downward. See the `init.c` file.
free_mem	Used for the kernel to create the timer thread and root thread. DO NOT USE THIS SECTION FOR ANY OTHER PURPOSES.

Table 10: NET+OS linker file sections

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

The `free_mem` function creates the root thread responsible for starting NET+OS and the IP stack. Do not pass any other address to create the root thread.

Note: `first_unused_memory` is a global variable set up by the kernel loading.

Updating Flash Support

CHAPTER 7

This chapter describes how to update flash memory.

Flash programming APIs

NET+OS includes multiple APIs, for reading, writing, and erasing flash memory attached to the NET+ARM chip. See the *NET+OS Application Software Reference Guide* and the *NET+OS BSP Software Reference Guide* for more information about the APIs.

The internals of the API rely on `flash_id_table` in the `naflash.c` file to define the known flash parts. The API is guaranteed to function only with parts defined in this table. If the part is not recognized, you need to update the flash_id_table.

Supported flash parts

This version of NET+OS supports these flash ROM parts:

Manufacturer	Part numbers
AMD	AM29DL323DB-90EI AM29F800BB AM29F800BT
Atmel	29C040 29C040A 49BV8011 49BV8011T 49BV1614A
Fujitsu	29LV800BA
Macronix	MX28F4000
Sharp	LH28F800SG
SST	28SF040 39VF800
ST Micro	M29W800AB-90NIT M29W320DB

Table 11: Supported flash ROM parts

Flash table data structure

The `flash_id_table_t` data structure is defined in the `flash.h` file, as shown. Note the following data types:

- `WORD8` is an unsigned byte
- `WORD16` is an unsigned short
- `WORD32` is an unsigned long

Data structure in flash.h file

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

typedef struct
{
                WORD16mcode;
                WORD16mcode_addr
                WORD16dcode;
                WORD16dcode_addr;
                WORD16total_sector_number;
                WORD32sector_size;
                WORD16prog_size;
                WORD16access_time;
                flash_cmd_t*id_enter_cmd;
                WORD16id_enter_len;
                flash_cmd_t*id_exit_cmd;
                WORD16id_exit_len;
                flash_cmd_t*erase_cmd;
                WORD16erase_len;
                flash_cmd_t*write_cmd;
```

```
                    WORD16write_len;
                    flash_cmd_t*sector_erase_cmd;
                    WORD32*sector_size_array;
} flash_id_table_t;
```

Structure fields

Table 12 summarizes the fields in the `flash_id_table_t` data structure.

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

Table 12: flash_id_table_t fields

Adding new flash

To add support for new flash ROM:

1. Edit the `flash.h` file and add definitions for the new flash device, such as number of flash sectors, flash sector size, and program load size.

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

```
/* 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
```

2. In `flash.h`, update the ROM type value; for example:

```
#define STM_29W800AB 0x0D
```

3. Edit the `naflash.c` file and update the `flash_id_table` definition.

   For the STM_29W800AB, for example, add the following entry to the end of the 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_flsh_block_erase_cmd, (WORD32
*)STM_M29W800AB_flash_block_size_array
}
```

The values in the example are defined as follows:

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

4. Update other command sequences such as `id_enter_cmd`, `id_exit_cmd`, and so on. These command sequences are usually available in the flash data sheet.
5. Rebuild the `NA1` library in the top-level directory, which includes the `flash.bld` file.

Supporting larger flash

Three constants in `flash.h` need to be changed to support larger flash configurations:

- `MAX_SECTORS`. The maximum number of flash sectors supported. Increase `MAX_SECTORS` to support flash parts with a larger sector count.
- `MAX_SECTOR_SIZE`. The maximum sector size supported. Increase `MAX_SECTOR_SIZE` to support flash parts with a larger sector size.
- `MAX_FLASH_BANKS`. The maximum number of flash banks supported. Increase `MAX_FLASH_BANKS` to support a larger number of flash banks.

After these constants are changed, rebuild the `NA1` library in the top-level directory, which includes `flash.bld`.

ROM Image Compression and Decompression

CHAPTER 8

This chapter describes the NET+OS data compression and decompression techniques for ROM image compression, storage, and expansion.

Understanding loss-less compression

Loss-less compression consists of techniques guaranteed to generate a duplicate of the input data stream after a compress/expand cycle. In real-time embedded applications, the loss of even a single bit can be catastrophic.

Loss-less data compression generally is implemented using one of two types of modeling:

- **Statistic modeling** reads a single symbol at a time, using the probability of that character's appearance.
- **Dictionary-based modeling** uses a single code to replace strings of symbols. The dictionary-based LZSS (also called *sliding window compression*) algorithm has high compression ratio and compression rate, and its expansion code is simple and fast.

NET + OS implementation

NET+OS implementation is summarized in three steps:

1. Compresses executable code before downloading it into flash.
2. Decompresses the code into RAM during powerup reset.
3. Executes the application code from RAM.

This process reduces the amount of flash needed, and the application code runs faster (in ARM) in RAM.

Compression and decompression files

When you install NET+OS, the `compress.exe` file is copied into the Green Hills Software environment. This file is the executable file for compression.

The expansion (decompression) code is in `decompress.c` in the `src\bsp` directory. You must compile `decompress.c` into the application you are building.

Build files

NET+OS provides three kinds of build files:

- **debug.bld.** Builds the application for debugging, so you can use JEENI or Raven to debug the code.
- **rom.bld.** Builds the application into an executable binary `.bin` file, which you then download into flash — when the application will run from flash all the time.
- **romzip.bld.** Builds a compressed image.

Building a compressed image

The `romzip.bld` build file defines `ENABLE_FLASH_COMPRESSION`. The file contains the initialization code and a subproject (`ramimagezip.bld`), which outputs a `ram.bin` file.

During the build process, `ram.bin` is compressed (by `compress.exe`) into `ramimagezip.bin`. This file is then changed into `ramimagezip.o` (the real application) and linked to form the compressed binary image `romzip.bin`.

Download the `romzip.bin` file into flash the same way you download `rom.bin` for a normal, uncompressed build.

When the board is reset, the loader code boots from flash. It then checks for debugger mode or software reset mode. If neither of these modes apply, the code tests the RAM on the board, performs the power-on self-test, expands the read-only data into RAM, and does a software reset by jumping to that RAM location.

The application then boots from RAM, and will run faster than in flash.

Special considerations

Pay special attention to the following details:

- The PC7 (Port C, bit 7) data direction bit is used to distinguish between a hardware reset and a software reset. This bit can be cleared using either direct control or a hardware reset condition.

- In the `ramimagezip.lx` linker file, the application image is linked at `.picbase`. For example, if `.picbase` is linked at `0x800000`, the zipped application image is unzipped to this RAM address.

  Be sure that this address is sufficiently above the data portion (linked at `0x0`) of the `ramimagezip.bld` application, and make sure you use `#define RAMIMAGE_START` *`address`* (in this example, *`address`* is `0x800000`) to match this address in `decompress.c` before rebuilding `romzip.bld`.

- To run code in RAM with instruction cache on, change `.picbase` to `0x8800000` in `ramimagezip.lx` and rebuild `romzip.bld`.

Index

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W

X


NetSilicon
A Digi International® Company