

In any stand-alone embedded system that contains a microprocessor, the processor runs a small piece of code called a boot copier, or boot loader, after the system resets. The boot copier locates the appropriate application software in non-volatile memory, copies it to RAM, initializes critical system components, and branches to the entry point of the application software image. The block of data in non-volatile memory that contains the application software is commonly referred to as the boot image. Boot copiers range in complexity from basic byte-for-byte copy routines to comprehensive applications that perform rigorous system tests, select among multiple software applications, and unpack, decompress, and perform error detection on the proper application.

This document teaches you how to implement your own custom boot copier software using the Nios<sup>®</sup> II processor and Nios II software build tools. In addition, this document provides the basic information needed to externally control the Nios II boot process.

This document addresses how to implement a custom boot copier for a Nios II processor already configured in the FPGA. It does not address custom methods of configuring Altera<sup>®</sup> FPGAs.

-  For information about custom methods of configuring Altera FPGAs, refer to the [Configuration Center](#) page of the Altera website.

## Assumptions About the Reader

This document assumes that you are an advanced Nios II user and that you are comfortable reading and writing embedded software. If you are not familiar with the Nios II hardware or software development flow, Altera strongly recommends that you first become familiar with building a Nios II microprocessor system.

-  For step-by-step procedures that build an example Nios II microprocessor system, refer to the [Nios II Hardware Development Tutorial](#).

This document also assumes you are familiar with the command line operation of the Nios II flash programmer.

-  For details about the Nios II flash programmer, refer to the [Nios II Flash Programmer User Guide](#).

## Implementing a Custom Boot Copier

Implementing a custom boot copier requires you to deviate from the normal Nios II software build tools development flow. You must edit source files by hand and run file conversion utilities from the Nios II command shell.

This document includes example source code for the following types of custom boot copiers:

- **Advanced boot copier**—Includes extra features such as dual boot image support and CRC error checking.
- **Small boot copier**—Is a bare-minimum boot copier that requires very little memory space.

The files referenced in this application note are available in two **.zip** files on the Altera website. One file contains the hardware design example and the other contains the software program files.

### Hardware Design Files

The [Nios II Ethernet Standard Design Example](#) page of the Altera website contains the hardware design files for a number of Altera development boards. Navigate to the web page and locate the Nios II Ethernet Standard design example **.zip** file that corresponds to your board. Download and unzip the file in a project directory of your choosing. The remainder of this application note refers to this directory as *<project>*.



For more information about the design example and supported Altera development boards, refer to the [Nios II Ethernet Standard Design Example](#) page of the Altera website.

### Software Files

The [Design Files for AN458](#) contains the software program files and board support package (BSP). Download and unzip the file into your *<project>* directory.

## Default Nios II Boot Copier

This section discusses the operation of the default Nios II boot copier, describing the workings of both the Common Flash Interface (CFI) flash memory and the Altera erasable programmable configurable serial (EPCS) or quad serial configuration (EPCQ) variant. If you are unfamiliar with the default boot copier, read this section before implementing a custom boot copier.

### Overview of the Default Nios II Boot Copier

The default boot copier included with the Nios II processor provides sufficient functionality for most Nios II applications and is convenient to implement with the Nios II software build tools development flow. The default boot copier is automatically and transparently added to your system when you convert your executable files to flash programming files.



Altera recommends that you use the default Nios II boot copier unless you require a custom boot copier with different or additional functionality. Implementing a custom boot copier can complicate your software build process and hinder Altera's ability to provide technical support.

The default Nios II boot copier has the following features:

- Supports CFI or EPCS/EPCQ flash memory
- Unpacks and copies boot image to RAM
- Automatically branches to application code in RAM

## The Default CFI Flash Boot Copier

The Nios II default boot copier is automatically included by the Nios II flash programmer during the execution of the **elf2flash** utility. Based on the processor reset address, the **elf2flash** utility determines the entry point of the application code and the address range of flash memory, whether or not a boot copier is needed. A CFI boot copier is needed whenever the processor's reset address points to CFI flash memory and the application's `.text` section points to somewhere other than CFI flash memory. When a boot copier is needed, **elf2flash** packs the application code in a boot record, and then creates a Motorola S-record Flash Programming File (**.flash**) containing the default boot copier and the boot record. The flash programmer downloads this boot record to CFI flash memory.

Immediately after the Nios II processor completes reset, the boot copier executes, reads the boot record as described in [“Boot Images” on page 6](#), and copies the application code to RAM. After copying is complete, the boot copier reads the entry point of the application code from the boot record. The boot copier executes the jump to that address, and the application software begins executing.

## The Default EPCS/EPCQ Boot Copier

When the Nios II processor reset address is set to the base address of an EPCS/EPCQ controller in the Qsys system integration tool, the default EPCS/EPCQ boot copier is implemented. The EPCS/EPCQ controller supports the Nios II processor boot sequence with a small block of on-chip memory mapped to the EPCS/EPCQ controller base address. During Quartus II compilation, the EPCS/EPCQ boot copier is designated as the initial contents of this on-chip memory. When booting from EPCS/EPCQ, the **elf2flash** utility does not include a boot copier in the **.flash**. Instead, it includes the application code, packaged into a boot record. The flash programmer downloads the data, which is read by the EPCS/EPCQ boot copier located in on-chip memory.

For IV-series and earlier devices, after the Nios II processor completes reset, the boot copier executes from the on-chip memory block in the EPCS/EPCQ controller. The boot copier reads the Control Block (CB) header to check if a Programmer Object File (POF) FPGA configuration image is located at the beginning of the EPCS device. If it finds such a file, the boot copier reads the POF data to extract the size of the FPGA configuration image. The boot copier then looks for the software application boot record at the EPCS/EPCQ offset immediately following the last byte of the FPGA configuration image.

For V-series and later devices, after the Nios II processor completes reset, the boot copier executes from the on-chip memory block in the EPCS/EPCQ controller. The boot copier reads the POF header to check if the image is a POF file. The POF header is the first three bytes of the FPGA configuration image generated by the sof2flash utility. The boot copier then reads the POF header for the size of the FPGA configuration image and looks for the software application boot record at the EPCS/EPCQ offset immediately following the last byte of the FPGA configuration image.

When a boot record is found, the boot copier reads it and copies the application code to RAM. After copying completes, the boot copier reads the entry point of the application code from the boot record. The boot copier executes the jump to that address, and the application software begins executing.

The source code for both variants of the default boot copier is included with the Nios II Embedded Design Suite (EDS) in the `<install directory>/<version>/nios2eds/components/altera_nios2/boot_loader_sources` directory.

## Advanced Boot Copier Example

This section describes an advanced boot copier example. You can build the example to run either out of CFI flash or out of on-chip memory, and to support boot images stored in CFI or EPCS/EPCQ flash devices. The example is written in C and is heavily commented, making it easy to customize. This example includes the following features in addition to those provided by the default boot copier:

- Supports two separate boot images
- Supports status messages using a JTAG UART
- Performs error-checking on the boot image data
- Supports non-word-aligned boot images

### Driver Initialization

To keep memory requirements low, the advanced boot copier example performs only the minimal driver initialization necessary to support the features of the boot copier itself. By default, the example initializes these drivers:

- System Clock Timer
- JTAG UART
- Processor Interrupt Handler

After the boot copier completes initialization of these drivers, it branches to the main application code in RAM, which performs a full initialization of the system drivers.

If you decide that you do not require these components during boot, the example allows you to disable the initialization of their drivers individually, reducing code size.

## Printing to the JTAG UART

The boot copier in this example prints information to the JTAG UART peripheral during the boot process. Printing is useful for debugging the boot copier, as well as for monitoring the boot status of your system. By default, the example prints basic information such as a start up message, the addresses in flash memory at which it is searching for boot images, and an indication of the image it ultimately selects to boot. You can add your own print messages to the code easily.

The advanced boot copier example avoids using the `printf()` library function, for the following reasons:

- The `printf()` library may cause the boot copier to stall if no host is reading output from the JTAG UART.
- The `printf()` library can potentially consume large amounts of program memory.

### Preventing Stalls by the JTAG UART

The JTAG UART behaves differently than a traditional UART. A traditional UART typically transmits serial data regardless of whether or not an external host is listening. If no host reads the serial data, the data is lost. The JTAG UART, on the other hand, writes its transmit data to an output buffer and relies on an external host to read from the buffer to empty it. By default, the JTAG UART driver stalls when the output buffer is full. The driver waits for an external host to read from the output buffer before writing more transmit data. This process prevents the loss of transmit data.

During boot, however, it is possible that no host is connected to the JTAG UART. In this case, no transmit data is read from the JTAG UART output buffer. When the output buffer fills, the `printf()` function stalls the entire program. This stalling is a problem, because the boot copier must continue bringing up the system regardless of whether an external host has connected to the JTAG UART.

To avoid this problem, the advanced boot copier example implements its own printing routine, called `my_jtag_write()`. This routine includes a user-adjustable timeout feature that allows the JTAG UART to stall the program for a limited timeout period. After the timeout period expires, the program continues without printing any more output to the JTAG UART. Using this routine instead of `printf()` prevents the boot copier from stalling if no host is connected to the JTAG UART.

### Reducing Memory Use for Printing

The advanced boot copier example also allows you to disable JTAG UART printing altogether. This can significantly reduce the memory requirements of the boot copier. To disable JTAG UART printing in the example, follow these steps:

1. Locate the following line in the `<project>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file:  

```
#define USING_JTAG_UART 1
```
2. Replace the line with the following line:  

```
#define USING_JTAG_UART 0
```

## Boot Images

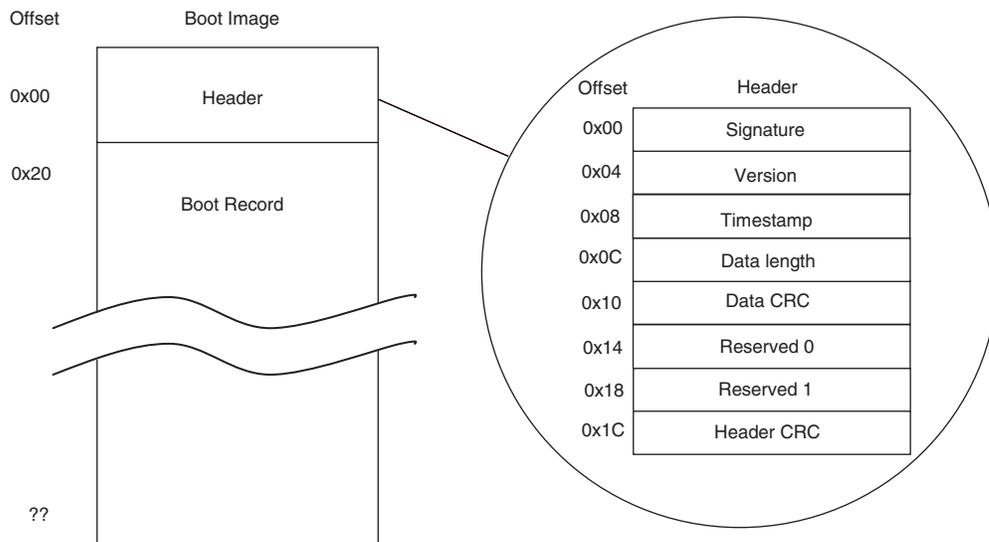
The advanced boot copier example expects to find boot images that conform to a specific format, and supports up to two boot images stored in flash memory. It does not assume the boot image starts at a 32-bit data boundary in flash.

## Boot Image Format

The advanced boot copier example expects to find a boot image that conforms to a specific format. The `make_flash_image_script.sh` script creates boot images that comply with the expected format. The `make_flash_image_script.sh` script runs the `elf2flash` utility to create a boot record of the application from the Executable and Linking Format File (`.elf`), and prepends some header information to that boot record.

Figure 1 shows the format of a boot image created using the `make_flash_image_script.sh` script.

**Figure 1. Example Boot Image Format**



## Boot Image Header Format

Each boot image includes a header at offset `0x0`. The example boot copier uses the header information attached to each boot image to extract information about the image and to make various decisions during the boot process. The `make_flash_image_script.sh` command-shell script automatically adds the header information to any boot image. Table 1 lists the information contained in the boot image header.

**Table 1. Example Boot Image Header Format (Part 1 of 2)**

Field	Size	Description	Location
Signature	32 bits	The signature used to locate the header in flash memory.	Defined in <code>make_flash_image_script.sh</code> . The default boot signature is <code>0xa5a5a5a5</code> .
Version	32 bits	A binary encoded version identifier for the application.	Defined in <code>make_flash_image_script.sh</code> .
Timestamp	32 bits	The time the header was created. Uses the standard C time integer value, seconds since JAN 01, 1970.	Generated by <code>make_flash_image_script.sh</code> .

**Table 1. Example Boot Image Header Format (Part 2 of 2)**

Field	Size	Description	Location
Data length	32 bits	The length of the application data contained in the boot, in bytes.	Generated by <code>make_flash_image_script.sh</code> .
Data CRC	32 bits	The CRC32 value for the entire application data	Generated by <code>make_flash_image_script.sh</code> .
Unused 0	32 bits	Unspecified purpose.	Defined in <code>make_flash_image_script.sh</code> .
Unused 1	32 bits	Unspecified purpose.	Defined in <code>make_flash_image_script.sh</code> .
Header CRC	32 bits	The CRC32 value for the header data.	Generated by <code>make_flash_image_script.sh</code> .

## Boot Record Format

The boot record immediately follows the boot image header. A boot record is a representation of the application that is loaded by the boot copier. The boot record contains an individual record for each code section of the application. A code section is a consecutive piece of the code that is linked to a unique region in memory. The boot copier reads the boot record to determine the destination address for each section of the application software code, and performs the appropriate copy operations.

The boot record is necessary because the code sections of a software application might not all be linked to one contiguous region in memory. Often, code sections of an application are scattered all over the memory map. To boot the application, the flash memory must contain the entire application and information about where its parts should be copied in memory. However, the flash memory is too small to contain a copy of the entire memory. The boot record representation packs all the code sections of the application in a single, contiguous block of flash memory.

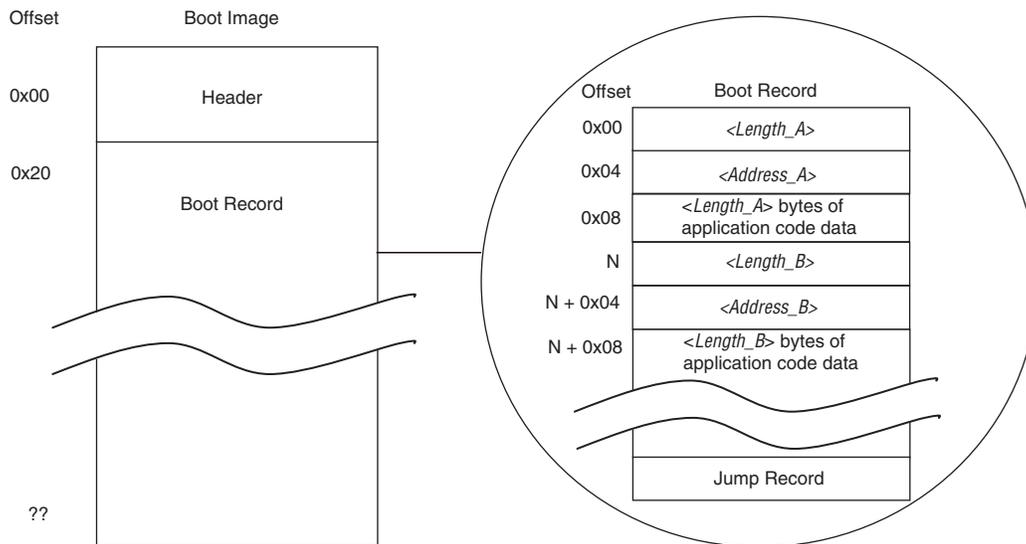
The boot record contains all the code sections of the software application in a contiguous block of data, regardless of where those code sections are linked in RAM. The boot record is a sequence of individual records, each containing the data for a code section, preceded by its destination address and its length. While booting, the boot copier reads the destination address (*<destination address>*) and the length (*<numbytes>*) from the boot record, then copies the following *<numbytes>* bytes from the boot record to *<destination address>*.

The final individual record in the boot record is a special jump record. Reading this record informs the boot copier it has completed copying application code, and that it now needs to jump to the 32-bit address stored in the following four bytes. The 32-bit address is the entry point of the application. Jump records are always encoded as 0x00000000.

The third type of individual record is a halt record. A halt record instructs the boot copier to halt its execution. Halt records are encoded as 0xFFFFFFFF. Erased areas of flash memory contain the value 0xFF in each byte. Therefore, if the boot copier ever encounters an erased area of flash, it interprets it as a halt record and stops execution.

Figure 2 shows the memory map in an example boot record.

**Figure 2. Example Boot Record Memory Map**



### Choosing a Boot Image

The advanced boot copier example supports up to two boot images stored in flash memory. The boot copier inspects two locations in flash memory, looking for a valid boot image at each location, then chooses one of the images to copy to RAM and execute. The two locations are predesignated as location numbers 1 and 2. To choose a boot image, the boot copier uses the following criteria, in the order in which they appear.

- Image validity
  - If only one valid boot image is found, the boot copier boots using that image.
  - If no valid boot images are found, the boot copier waits five seconds, then jumps back to the Nios II reset address.
- Revision number
  - If both boot images are valid, the boot copier looks at the version number of each image.
  - The boot copier chooses the boot image with the highest version number.
- Timestamp
  - If both boot images have the same version number, the boot copier looks at the timestamp of each image.
  - The boot copier chooses the boot image with the most recent timestamp.
- Default
  - If both boot images have the same timestamp, the boot copier chooses the image in location number 2.

## Word Alignment

In most cases, you can program a Nios II boot image starting at a 32-bit data boundary in flash memory. This placement allows the boot copier to copy application data with 32-bit word transfers. However, the advanced boot copier example does not assume this alignment. If the boot copier finds a valid boot image that is not 32-bit word-aligned in flash memory, the boot copier can still copy the application to RAM accurately. The boot copier uses the `memcpy()` library function to perform the copying. The `memcpy()` function requires little memory, and using `memcpy()` is a fast and robust method for copying data regardless of its alignment in memory.

## Boot Methods

The advanced boot copier example supports the following boot methods:

- Directly from CFI flash—Boot from CFI flash memory, copy the advanced boot copier from CFI flash memory to RAM, run the advanced boot copier which copies the application from CFI flash memory, and run the application image from RAM.
- From CFI flash, running from on-chip memory—Boot from on-chip RAM, copy the application from CFI flash memory, and run the application image from RAM.
- From EPCS/EPCQ flash, running from on-chip memory—Boot from on-chip RAM, copy the application from EPCS/EPCQ flash memory, and run the application image from RAM.

### Booting Directly From CFI Flash

This method uses a two-stage boot copier. The first boot copier runs from CFI flash and the second runs from RAM. The Nios II reset address is set to an address in CFI flash memory. The default CFI boot copier and advanced boot copier are then programmed at that address in flash. The default CFI boot copier begins executing when the Nios II processor is reset. The default CFI boot copier loads the second boot copier from CFI Flash into RAM and branches to the advanced boot copier's entry point. The advanced boot copier copies the application from CFI flash to RAM, and then branches to the application entry point.



Make sure the code space for the advanced boot copier does not overlap with the application's code space. In this example, we prevent overlap by dividing the SDRAM into two sections, an upper and lower region. The advanced boot copier loads in the upper region of the SDRAM and the application loads in the lower region of the SDRAM.

### Booting From CFI Flash, Running From On-Chip Memory

In this method, the Nios II reset address is set to the base address of a boot ROM implemented as FPGA on-chip memory. The boot copier executable is loaded in the boot ROM when the FPGA is configured, after the hardware design is compiled in the Quartus II software. The boot copier begins executing when the Nios II processor is reset. It copies the application code from CFI flash memory to RAM, and then branches to the application entry point.

## Booting From EPCS/EPCQ Flash, Running From On-Chip Memory

This method is very similar to the previous method. The difference is the boot images are stored in EPCS/EPCQ flash, not CFI flash. In this method, set the Nios II reset address to the base address of a boot ROM implemented as FPGA on-chip memory. Load the boot copier executable into the boot ROM when the FPGA is configured, after the hardware design is compiled in the Quartus II software. The boot copier begins executing when the Nios II processor is reset, copies the application code from EPCS/EPCQ flash memory to RAM, and then branches to the application entry point.

## Setting the Boot Method

The advanced boot copier example supports all three boot methods described above. The following line in the `<project>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file controls the method that is implemented:

```
#define BOOT_METHOD <boot method>
```

The options available for `<boot method>` are:

- `BOOT_FROM_CFI_FLASH`
- `BOOT_CFI_FROM_ONCHIP_ROM`
- `BOOT_EPCS_FROM_ONCHIP_ROM`

## Preventing Overlapping Data in Flash

When you set up your system to boot from a flash memory, consider other data that is also stored in that flash memory. Altera development boards are designed to support storing FPGA configuration images and software boot images together in either type of flash device, CFI or EPCS/EPCQ. When storing multiple images in flash memory, you must ensure that none of the images overlap one another.

### Overlapping Data in CFI Flash

Use the `nios2-elf-size` utility to compute the size of each of your flash images, then choose offsets in flash memory for those images based on their sizes (or estimated future sizes) that ensure they do not overlap.

### Overlapping Data in EPCS/EPCQ Flash

In EPCS/EPCQ flash, the FPGA configuration image must always start at offset 0x0. To avoid programming any boot images on top of the FPGA configuration image, you must determine the end offset of the FPGA configuration image. Convert your FPGA configuration image SRAM Object File (`.sof`) to a `.flash` image using the `sof2flash` utility, then run `nios2-elf-size` on that flash image. The result is the offset at the end of the FPGA configuration image in EPCS/EPCQ flash. Ensure that any software boot images you program into EPCS/EPCQ flash begin at an offset beyond the end of the FPGA configuration image.

## Boot Copier Code Size

The advanced boot copier example, without modification, compiles to an executable file of size approximately 8500 bytes. If you turn off all the JTAG UART and system clock timer functionality, the example executable size is reduced to approximately 2000 bytes.

By comparison, the code size of the default boot copier, described in “[Default Nios II Boot Copier](#)” on page 2, is approximately 200 bytes when compiled to boot from CFI flash, and approximately 500 bytes when compiled to boot from EPCS/EPCQ flash.

If you require a customizable boot copier that is smaller than 2000 bytes, refer to “[Small Boot Copier Example](#)” on page 19. The small boot copier is written in Nios II assembly language, and includes very few features. When executed, it simply copies a boot record located in CFI flash to RAM, and then branches to the copied application. The compiled code size is approximately 200 bytes.

## Implementing the Advanced Boot Copier Example

This section describes the steps required to build and run the advanced boot copier example on the Nios II Ethernet Standard design example.

### Setting Up the Software Tools and Development Board

To build and run the advanced boot copier example, you must first perform the following steps:

1. Ensure that you have Nios II EDS version 11.0 (or later) and Quartus II software version 11.0 (or later) installed on your computer.
2. Connect power and a USB-Blaster™ cable to your Altera development board.

### Creating a Suitable Hardware Design

In the following steps, you can open, modify, and generate a Nios II system on which you can run the advanced boot copier example. You must also decide which boot method you want to implement. Several of the following steps require you to take slightly different actions depending on the boot method you use.

To open the example project:

1. Verify that you have downloaded and unzipped the file described in “[Hardware Design Files](#)” on page 2 to your <project> directory on your computer.
2. In the Quartus II software, on the File menu, click **Open Project**, and open the <project>\niosii\_ethernet\_standard\_<board>.qpf project file.

If you intend to boot directly from CFI flash, the standard design example works without additional memory. You can proceed to “[Building the Advanced Boot Copier](#)” on page 13.

To add on-chip boot ROM to the system:

1. On the Tools menu, click **Qsys** to start Qsys and then select **eth\_std\_main\_system.qsys**. Click **Open** when prompted to open the Qsys design file.

2. In Qsys, on the **System Contents** tab, expand **Memories and Memory Controllers**, expand **On-Chip**, and select **On-Chip Memory (RAM or ROM)**.
3. Click **Add** to add the component to the system. Use the following settings in specifying the memory:
  - **Memory Type: RAM (Writable) (not ROM (Read-only))**
  - **Data width: 32 bits**
  - **Total memory size: 16384 Bytes**

The specified peripheral size ensures that it can hold the entire code image for the largest version of the example boot copier. This image includes the following code:

- Reset code in the `.entry` section
- The `crt0.s` startup code
- The `.text` section containing the `alt_main` entry point
- The `.rodata` section holding any initialized read only data
- The `.rdata` section holding any initialized read/write data
- The `.bss` section holding initialized and static variables.
- The exception handler located in the `.exception` section.

Some of these sections are copied to the exception RAM—the RAM that contains the exception vectors—when the `crt0.s` startup code executes, but all of the sections are stored initially in this on-chip memory.

4. In the Qsys connection matrix, ensure that the slave port of the on-chip memory is connected to the Nios II instruction master and to the Nios II data master, and that the reset port of the on-chip memory is connected to the reset output clock source.
5. If Qsys reports an error in the bottom window caused by the address of the new on-chip memory overlapping another peripheral, select a suitable base address for the on-chip memory that does not overlap anything else.
6. Modify the clock entry for the new on-chip memory to ensure that this memory is clocked by the same clock as the **cpu** component.
7. Right-click the new **On-Chip Memory** component, and click **Rename**. Rename the component with a descriptive name such as `boot_rom`.
8. To enable running the boot copier from on-chip memory, right-click the **cpu** component in your system and click **Edit**.
9. In the Nios II Processor settings window, set the **Reset Vector Memory** to `boot_rom.s1` with an **Offset** of `0x00000000`.
10. Click **Finish** to exit the Nios II Processor settings window.
11. Click **Generate** to generate the Qsys system.

## Building the Advanced Boot Copier

To build the example advanced boot copier in a new Quartus II project directory, perform the following steps:

1. Verify that you have downloaded and unzipped the file described in “[Software Files](#)” on page 2 to your `<project>` directory on your computer.
2. Open the file `<project>/boot_copier_sw/bsp/advanced_boot_copier_bsp/bootcopier_bsp_settings.tcl` in a text editor.
3. To ensure the proper bsp settings and memory linker locations, edit the following settings in `bootcopier_bsp_settings.tcl`:
  - a. Set `ONCHIP` to 1 if you are booting from on-chip memory, or 0 otherwise.
  - b. Edit `EXCEPTION_OFFSET` and `RESET_OFFSET` to match values of the CPU component in Qsys.
  - c. Update `SDRAM_SIZE` and `FLASH_SIZE` to match the memory sizes in Qsys.
4. Open the file `<project>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` in a text editor.
5. Edit the line:

```
#define BOOT_METHOD <boot_method>
```

to indicate the boot method you intend to use. The following options are available for `<boot_method>`:

- `BOOT_FROM_CFI_FLASH`
- `BOOT_CFI_FROM_ONCHIP_ROM`
- `BOOT_EPCS_FROM_ONCHIP_ROM`

This `#define` directs the Compiler to build the boot copier appropriately for the boot method you are using.

6. To prevent the application from printing messages to the JTAG UART during boot, edit the line:
 

```
#define USING_JTAG_UART 1
```

 to read:
 

```
#define USING_JTAG_UART 0
```

 This `#define` directs the Compiler to build the boot copier leaving out all JTAG UART code.
7. Open a Nios II command shell. (On Windows, click **Start > All Programs > Altera > Nios II EDS > Nios II Command Shell**).
8. Change to the directory `<project>/boot_copier_sw/app/advanced_boot_copier`.
9. To create and build the BSP and application projects, type the following command:
 

```
./create-this-app ↵
```

You now have an executable boot copier that is ready to run on the Nios II processor. Next, you must create an application to boot using the new boot copier.

## Building a Test Application to Boot

To build a test application to boot with the advanced boot copier, perform the following steps:

1. Open a Nios II command shell. (On Windows, click **Start** > **All Programs** > **Altera** > **Nios II EDS** > **Nios II Command Shell**).
2. Change to the directory `<project>/boot_copier_sw/app/hello_world`.
3. To create and build the test application BSP and application projects, and generate an executable `hello_world.elf` file, type the following command:

```
./create-this-app ↵
```

## Packing the Test Application in a Boot Record

In this section, package the test application to boot in a boot record that the boot copier can understand. To package the application, run a script from a Nios II command shell. The following scripts are included with the design files:

- `make_flash_image_script.sh`
- `make_header.pl`
- `read_flash_image.pl`

To package the test application to boot using the advanced boot copier, perform the following steps:

1. Open `<project>/boot_copier_sw/app/hello_world/make_flash_image_script.sh` in a text editor and update the `flash_base` and `flash_end` parameters to match your system.
2. In your Nios II command shell, run the `make_flash_image_script.sh` script to package the `.elf` file in a boot record, by typing the following command:

```
./make_flash_image_script.sh hello_world.elf ↵
```



Running this script might issue a warning about an empty loadable segment and display the name of an intermediate file `fake_flash_copier.srec`. You can safely ignore these messages.

The script creates the files `hello_world.elf.flash.bin` and `hello_world.elf.flash.srec` in the current directory. You now have all the binary images needed to boot a test application with the example boot copier. Next, you program these images in the appropriate locations.

## Booting Directly From CFI Flash Memory

To use the Nios II flash programmer to program the boot copier and the test application in CFI flash memory, perform the following steps:



If you intend to boot from on-chip memory, this section does not apply. Skip ahead to [“Booting CFI or EPCS/EPCQ Flash From On-Chip Memory”](#) on page 16.

1. In the Quartus II software, on the Tools menu, click **Programmer**.
2. Make sure the `<project>\niosii_ethernet_standard_<board>.sof` filename appears in the **File** column.
3. Make sure the **Program/Configure** option is turned on.

4. Click **Start** to configure your FPGA with the **.sof** file.
5. In a Nios II command shell, change to the directory `<project>/boot_copier_sw/app/hello_world`.
6. Set the offset in flash memory at which to locate the `hello_world` boot image, by typing the command:

```
bin2flash --input=hello_world.elf.flash.bin \
          --output=hello_world.flash \
          --location=0x00240000 ←
```

Set the offset to `0x00240000` or `0x00440000`, because in boot from CFI flash mode, these are the two locations where the boot copier expects boot images 1 and 2, respectively. The two addresses work equally well.

You can also change these default locations by editing the `#define` statements for `BOOT_IMAGE_1_OFFSET` and `BOOT_IMAGE_2_OFFSET` in the `<project>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file, and then rebuilding the boot copier.

7. Program the `hello_world` boot image in flash memory by typing the following command:

```
nios2-flash-programmer --base=<flash_base> \
                      hello_world.flash←
```

where `<flash_base>` is the base address of the CFI flash component in your Qsys system.

8. In a Nios II command shell, change to the directory `<project>/boot_copier_sw/app/advanced_boot_copier`.
9. Create the flash memory file for the boot copier by typing the following command:

```
make flash ←
```

This command creates the file `<flash_component>.flash`, where `<flash_component>` is the name of the CFI flash component in your Qsys system.

10. Program the boot copier to flash memory by typing the following command:

```
nios2-flash-programmer --base=<flash_base> \
                      <flash_component>.flash ←
```

11. Skip ahead to [“Running the Advanced Boot Copier Example” on page 18](#).

## Booting CFI or EPCS/EPCQ Flash From On-Chip Memory

In this section, use the Quartus II software to program the boot copier in the `boot_rom` memory of the FPGA, and then use the Nios II flash programmer to program the test application boot record in either CFI or EPCS/EPCQ flash memory.



If you intend to boot directly from CFI flash memory, this section does not apply. Booting directly from CFI flash memory is covered in [“Booting Directly From CFI Flash Memory” on page 15](#).

To program the boot copier in the FPGA’s `boot_rom` memory, perform the following steps:

1. In a Nios II command shell, change to the subdirectory `<project>/boot_copier_sw/app/advanced_boot_copier` of your Quartus II project directory.
2. To generate the memory initialization file for the **boot\_rom** memory that contains the boot copier, type the following command:
 

```
elf2hex advanced_boot_copier.elf <boot_rom_start_address>
<boot_rom_end_address> --width=32 --create-lanes=0
../../../../boot_rom.hex ←
```
3. On the Quartus II Processing menu, click **Start Compilation** to compile the project.
4. When compilation is complete, on the Tools menu, click **Programmer**.
5. Make sure the `<project>\niosii_ethernet_standard_<board>.sof` filename appears in the **File** column.
6. Make sure the **Program/Configure** option is turned on.
7. Click **Start** to configure your FPGA with the **.sof** file.

The **boot\_rom** memory on the FPGA now contains an executable image of the example boot copier.

To program the test application in flash memory, perform the following steps:

1. In a Nios II command shell, change to the subdirectory `<project>/boot_copier_sw/app/hello_world` of your Quartus II project directory.
2. Set the offset in flash memory at which to locate the `hello_world` boot image, by typing one of the following commands:

- If you are booting from CFI flash memory, type the following command:

```
bin2flash --input=hello_world.elf.flash.bin \
--output=hello_world.flash \
--location=0x00240000 ←
```

- If you are booting from an EPCS/EPCQ device, type the following command:

```
bin2flash --input=hello_world.elf.flash.bin \
--output=hello_world.flash \
--location=0x00060000 ←
```

Set the offset to `0x00240000` or `0x00440000` when booting from CFI flash memory, and to `0x00060000` or `0x00080000` when booting from an EPCS/EPCQ device, because in boot from the respective flash memory, these are the two locations where the boot copier expects boot images 1 and 2, respectively. In both cases, the two addresses work equally well.

You can also change these default locations by editing the `#define` statements for `BOOT_IMAGE_1_OFFSET` and `BOOT_IMAGE_2_OFFSET` in the `<project>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file, and then rebuilding the boot copier.

-  If you edited the flash image offsets in `advanced_boot_copier.c`, specify the `--location` value as one of the image offsets you defined in `advanced_boot_copier.c`, not the offsets mentioned here.

3. Program the hello\_world boot image in flash memory by typing the one of the following commands:

- If you are booting from CFI flash memory, type the following command:

```
nios2-flash-programmer --base=<flash_base> \  
hello_world.flash ↵
```

- If you are booting from an EPCS/EPCQ device, type the following command:

```
nios2-flash-programmer --base=<flash_base> \  
--epcs hello_world.flash ↵
```

where *<flash\_base>* is the base address of the CFI or EPCS/EPCQ flash component in your Qsys system.

## Running the Advanced Boot Copier Example

To run the advanced boot copier example on your development board, perform the following step:

- After the flash programmer completes, in a Nios II command shell, type the following command to reset the Nios II processor:

```
nios2-download -r -g ↵
```

The boot loader and the test application both print status messages to the JTAG UART if it is enabled. If the JTAG UART and SYS\_CLK\_TIMER are initialized in the `bsp/alt_sys_init.c` file, and USING\_JTAG\_UART remains at value 1 in the `<project>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file, you can view these messages.

To see the messages, perform the following step:

- In a Nios II command shell, run the **nios2-terminal** utility by typing the following command:

```
nios2-terminal ↵
```

 If **nios2-terminal** cannot connect to the JTAG UART with the default settings, run it with the `--help` option for a listing of the command line switches that might be needed.

 If your **nios2-terminal** displays truncated output from the boot copier, followed by the boot image output, press the CPU Reset button on your development board to repeat the boot process and view the full output. Refer to [Figure 3](#) for the expected output if you boot CFI flash memory from on-chip RAM.

If the boot copier runs successfully, you see output from **nios2-terminal**, as shown in [Figure 3](#). Your output differs slightly if booting from external memory or booting EPCS/EPCQ flash.

**Figure 3. Advanced Boot Copier Output**

```
Altera Nios II EDS 11.0 [gcc4]
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>
*****
Example Custom Boot Copier Starting
  Booting CFI from On-Chip RAM
*****
This copier expects application images
to be located at offset 0x00240000 or
0x00440000 in flash memory.
*****
Now attempting to boot.
Picked image in flash at location 1
Now attempting to load and jump to the
application.
Hello from Boot Image 2!
```

## Small Boot Copier Example

This section describes a small code size boot copier example for users interested in using as little memory as possible.

### Small Boot Copier Features

The small boot copier example is a minimal program designed to use very little program memory. It performs only the following operations:

1. Reads an application boot record from flash memory
2. Copies it to RAM
3. Jumps to the application's entry point

The small boot copier supports only one boot image, does not perform any error checking, and does not support printing messages to the JTAG UART. If you are interested in a more advanced boot copier, refer to [“Advanced Boot Copier Example” on page 4](#).



This example is designed to run on the Altera Nios II Ethernet Standard design example. For more information about the design example and supported Altera development boards, refer to the [Nios II Ethernet Standard Design Example](#) page of the Altera website.

## Implementation in Nios II Assembly Language

To keep the code size as small as possible, write the small boot copier example in Nios II assembly language. All the variables that the boot copier uses are implemented in Nios II processor general purpose registers, not in RAM. Therefore, the boot copier itself has no data memory requirement. The small boot copier has no `.rodata`, `.rwdata`, `stack`, or `heap` section. Because it does not require data memory, this boot copier can easily be relocated anywhere in memory and can even run directly from non-volatile flash memory without setting up a data memory section.

## System Initialization

The small boot copier performs only the minimum necessary system initialization. The following initialization tasks are performed by the boot copier:

- Clears the processor status register to disable interrupts
- Flushes the instruction cache
- Flushes the processor pipeline

## Code Size

The small boot copier compiles to an executable file that is only 200 bytes long. This boot copier is small enough to fit in one M9K block, the smallest unit of memory in a Cyclone III FPGA.

## Implementing the Small Boot Copier Example

This section describes the steps required to build and run the small boot copier example on the Nios II Ethernet Standard design example. This boot copier is a bare-minimum, small-code-size version written in assembly language. If you want to build a more full-featured boot copier, refer to [“Implementing the Advanced Boot Copier Example” on page 12](#).

The small boot copier example is built in a Nios II command shell using the `make` utility.

## Setting Up the Software Tools and Development Board

To build and run the small boot copier example, you must first perform the following steps:

1. Ensure that you have Nios II EDS version 11.0 (or later) and the Quartus II software version 11.0 (or later) installed on your computer.
2. Connect power and a USB Blaster to your Altera development board.

## Creating a Suitable Hardware Design

In the following steps, you open, modify, and generate a Nios II system on which you can run the small boot copier example.

To open the example project and add on-chip ROM to the system:

1. Verify that you have downloaded and unzipped the file described in “[Hardware Design Files](#)” on page 2 to your `<project>` directory on your computer.
2. In the Quartus II software, on the File menu, click **Open Project**, and open the `<project>\niosii_ethernet_standard_<board>.qpf` project file.
3. On the Tools menu, click **Qsys** to start Qsys.
4. In Qsys, on the **System Contents** tab, expand **Memories and Memory Controllers**, and select **On-Chip Memory (RAM or ROM)**.
5. Click **Add** to add the component to the system. Specify the following memory settings:
  - **Memory Type: ROM (Read-only)**
  - **Data width: 32 bits**
  - **Total memory size: 512 Bytes**

The specified on-chip memory size ensures that no memory space is wasted. The smallest usable block of memory in a Cyclone III FPGA is 512 bytes (one M9K block). Although the small boot copier example requires only 200 bytes of memory, the remainder of the M9K block can be used only after you enable it. Therefore, Altera recommends that you enable the entire block, rather than waste it.

6. Right-click the new **On-Chip Memory** and click **Rename**. Specify a descriptive name such as `boot_rom`.
7. In the Qsys connection matrix, ensure that the slave port of the on-chip memory is connected to the Nios II instruction master and to the Nios II data master, and that the reset port of the on-chip memory is connected to the reset output of the clock source.
8. If Qsys reports an error in the bottom window caused by the address of the new on-chip memory overlapping another peripheral, select a suitable base address for the on-chip memory that does not overlap anything else.
9. Modify the clock entry for the new on-chip memory to ensure that this memory is clocked by the same clock as the **cpu** component.
10. To enable running the boot copier from on-chip memory, right-click the **cpu** component in your system and click **Edit**.
11. In the Nios II Processor settings window, set the **Reset Vector Memory** to `boot_rom.s1` with an offset of `0x00000000`.
12. Click **Finish** to exit the Nios II Processor settings window.
13. Click **Generate** to generate the Qsys system.

## Building the Small Boot Copier Using 'make'

This section describes how to build the example small boot copier from the Nios II command shell.

To build the example small boot copier in a new Quartus II project directory, perform the following steps:

1. Verify that you have downloaded and unzipped the file described in “[Software Files](#)” on page 2 to your `<project>` directory on your computer.
2. Open a Nios II command shell. (On Windows, click **Start** > **All Programs** > **Altera** > **Nios II EDS** > **Nios II Command Shell**).
3. Change to the directory `<project>/boot_copier_sw/app/small_boot_copier`.
4. In Qsys, determine the base address of your `ext_flash` component (`<flash_base_address>`).
5. In the Nios II command shell, type the following command:

```
make all FLASH_BASE=<flash_base_address> \  
      BOOT_IMAGE_OFFSET=0x00240000
```

This command builds the small boot copier, hardcoding it to look for a boot image at offset 0x00240000 in flash memory and assumes no other important data is located there. You can freely modify this offset to a value more relevant to your application, but when you program the boot image in flash memory (in step 2 on page 23), ensure that you program it to the same offset you choose in the current step.



This example uses a makefile in place of the Nios II software build tools because you are only compiling a single assembly file, with no associated drivers.

You now have an executable boot copier named `small_boot_copier.hex` that is ready to run on the Nios II processor. Next, you must create an application to boot using the new boot copier.

## Building a Test Application to Boot

To build a test application to boot using the small boot copier, perform the steps in “[Building a Test Application to Boot](#)” on page 14.

## Booting From On-Chip Memory

In this section, use the Quartus II software to program the small boot copier in the `boot_rom` memory of the FPGA, and then use the Nios II flash programmer to program the test application boot record in CFI flash memory.

To program the boot copier in the FPGA’s `boot_rom` memory, perform the following steps:

1. Change to the `<project>/boot_copier_sw/app/small_boot_copier` directory.
2. Copy `small_boot_copier.hex` to the Quartus II project directory and rename it `boot_rom.hex` using the following command:

```
cp small_boot_copier.hex ../../../../boot_rom.hex ↵
```



You may see a warning that a file by that name already exists in that directory. If you are asked to replace the old file, click **Yes**.

The next Quartus II compilation implements the boot copier executable as the contents of **boot\_rom**.

3. If Qsys is still open, return to it and click **Exit** to close it.
4. In the Quartus II window, on the Assignments menu, click **Settings**.
5. In the **Category** list, click **Compilation Process Settings**, then turn on **Use Smart Compilation**. This option prevents recompilation of the entire design when only an update to the on-chip memory contents is required. The first Quartus II compile, however, must be a full compile, because adding an on-chip memory to the system changed the design.
6. On the Processing menu, click **Start Compilation** to compile the Quartus II project.
7. When compilation is complete, on the Tools menu, click **Programmer**.
8. Make sure the `<project>\niosii_ethernet_standard_<board>.sof` filename appears in the **File** column.
9. Make sure the **Program/Configure** option is turned on.
10. Click **Start** to configure your FPGA with the **.sof** file.

The **boot\_rom** memory on the FPGA now contains an executable image of the example boot copier.

To program the test application in CFI flash memory, perform the following steps:

1. In a Nios II command shell, change to the directory `<project>/boot_copier_sw/app/hello_world`.
2. Set the offset in flash memory at which to locate the hello\_world boot image, by typing the command:

```
bin2flash --input=hello_world.elf.flash.bin \
          --output=hello_world.flash \
          --location=0x00240000 ↵
```

Set the location to 0x240000, because in boot from CFI flash mode, this is the location where the small boot copier expects to find the boot image. The correct value for the `--location` parameter is the value specified in the command in step 5 on [page 22](#).

3. Program the hello\_world boot image in flash memory by typing the following command:

```
nios2-flash-programmer --base=<flash_base> \
                      hello_world.flash ↵
```

where `<flash_base>` is the base address of the CFI flash component in your Qsys system.

## Running the Small Boot Copier Example

To run the small boot copier example on your development board:

1. After the flash programmer completes, in a Nios II command shell, type the following command to reset the Nios II processor:

```
nios2-download -r -g ↵
```

The boot copier should now boot the test application.

2. To test that the test application actually loads and executes, run the **nios2-terminal** utility in the Nios II command shell by typing the following command:

```
nios2-terminal ↵
```

If the boot copier runs successfully, you see output from **nios2-terminal**, as shown in [Figure 4](#).



If **nios2-terminal** cannot connect to the JTAG UART with the default settings, run it with the `--help` option for a listing of the command line switches that might be needed.

**Figure 4. Small Boot Copier Output**

```
Altera Nios II EDS 11.0 [gcc4]
bash-3.1$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-01]", device 1, instance 0
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>
Hello from Boot Image 2!
```

## Debugging Boot Copiers

Some special considerations should be made when attaching the Nios II SBT for Eclipse™ debugger to a processor running boot copier code. The following section discusses the requirements for debugging boot copiers.

## Hardware and Software Breakpoints

Boot copiers often run from non-volatile memory, which affects the types of breakpoints that can be set in the code. The two types of breakpoints used by the Nios II debugger are software breakpoints and hardware breakpoints. Software breakpoints replace the processor instruction at the breakpoint location with a different instruction that transfers control to the debugger. This replacement method requires that the program memory be writable so that the breakpoint instruction can be written. Because boot copiers often run from non-volatile memory such as flash memory, software breakpoints cannot be set in boot copiers.

Hardware breakpoints detect the address value of the breakpoint on the instruction address bus, and then transfer control to the debugger using hardware. Therefore, a hardware breakpoint can be set in non-volatile memory. Only a hardware breakpoint can be set in a boot copier that runs from flash memory.

## Enabling Hardware Breakpoints

To enable hardware breakpoints in the Nios II processor:

1. In Qsys, open the Nios II MegaWizard interface by double clicking the system's Nios II processor.
2. In the Nios II MegaWizard interface, click the **JTAG Debug Module** page.
3. Select **Level 2** or greater, which allows two simultaneous hardware breakpoints that the Nios II debugger can automatically use.

## Breaking Before main()

When debugging a boot copier, you may want to start debugging immediately after reset, instead of waiting until reaching the function `main()`. Some boot copiers do not contain a function `main()` at all. In these cases, instruct the debugger to set a breakpoint at the program entry point.

## Setting Up the Debugger

To configure the Nios II debugger for debugging a boot copier:

1. Import your boot copier project to the Nios II SBT for Eclipse by performing the following steps:
  - a. Open the Nios II SBT for Eclipse (selecting the workspace folder of your choice).
  - b. On the File menu, click **Import**. The **Import** dialog box appears.
  - c. Expand the **Nios II Software Build Tools Project** folder, and select **Import Nios II Software Build Tools Project**.
  - d. Click **Next**.
  - e. Under **Project location**, browse to your boot copier project folder.
  - f. Enter a project name.
  - g. Turn off **Clean project when importing** and **Managed project**.
  - h. Click **Finish**.

2. In the Nios II SBT for Eclipse, highlight the name of the imported boot copier project, and on the **Run** menu, click **Debug Configurations**.
3. In the **Debug** configuration dialog box, click the **New** icon to create a new debug configuration.
4. Click the **Target Connection** tab.
5. In the **Download** box:
  - a. If your boot copier runs from ROM, turn on **Start processor**, turn on **Reset the selected target system**, and turn off **Download ELF to selected target system**.
  - b. If your boot copier runs from RAM, turn on **Start processor**, turn off **Reset the selected target system**, and turn on **Download ELF to selected target system**.
6. Click the **Debugger** tab.
7. Turn on **Stop on Startup at: main**.
8. Click **Apply**.
9. Click the **Debug** button to start the debugger. Once connected, the debugger breaks at the entry point of the boot copier.

## Externally Controlling the Nios II Boot Process

Another way to boot the Nios II processor is to have a different component, such as another processor, control the boot process externally. In this situation, the external processor reads the Nios II application code from some source and loads it into Nios II program memory. The external processor can retrieve the Nios II application code from various sources. For example, it might read the code from some non-volatile storage medium such as hard disk, or download the code over an Ethernet connection.

The method by which the external processor retrieves the Nios II application code is outside the scope of this document. This section focuses on the process of safely loading the application code in Nios II program memory, then directing the Nios II processor to properly execute the application.

### Overview

Two different methods are available to implement an externally controlled boot of a Nios II system.

- The external processor unpacks the Nios II boot image and writes the executable application code to Nios II program memory.
- The external processor only copies the boot image to RAM. The Nios II processor takes over from there and unpacks the boot image itself.

The latter method, letting the Nios II processor unpack and load the application from the boot image, is similar to the process of running a normal boot copier on the Nios II processor. The only difference is that instead of a flash programmer placing the boot image in flash memory, an external processor copies the boot image to RAM. After the external processor releases the Nios II processor from reset, everything happens just as if the Nios II processor were booting from flash memory.

This section focuses on the first method, in which the external processor unpacks the Nios II boot image, copies the application code to Nios II program memory, and then directs the Nios II processor to the application's entry point.

One common requirement, regardless of external boot method, is that you must prevent the Nios II processor from executing any code in the memory space being written by the external processor during the copying and unpacking processes. Otherwise, you may encounter race condition and data corruption problems. The process described in this section prevents the Nios II processor from executing code by holding it in reset while the application code is copied to Nios II program memory. After the application code is copied, the Nios II processor is released from reset to execute the application.

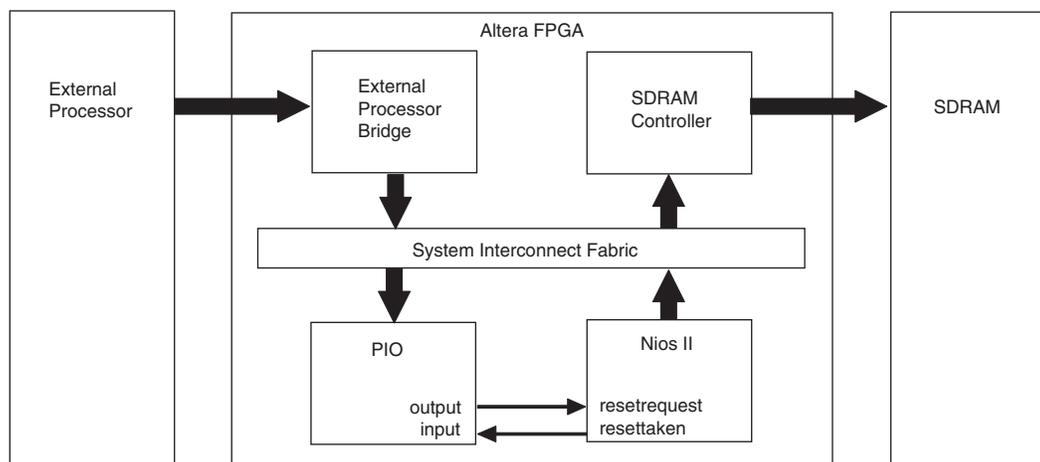
## Building an Appropriate Qsys System

Before you can successfully implement an externally controlled Nios II boot, you must ensure your Qsys system contains the necessary hardware. An external processor must be able to access the appropriate system peripherals and control the reset state of the Nios II processor. The following list describes the minimum hardware elements required to support an externally controlled Nios II boot.

- External Processor Bridge
- Nios II processor with the following features:
  - A `cpu_resetrequest` signal
  - A reset address that points to RAM
  - A one-bit parallel I/O (PIO) peripheral device

Figure 5 shows the block diagram of a system that can control the boot of a Nios II processor externally.

**Figure 5. Block Diagram of Externally Controlled Nios II Boot System**



## External Processor Bridge

To allow an external processor to access peripherals in your Qsys system, the system must include a bridge between the Avalon fabric and the external processor bus.

Bridges to external processors can be acquired as intellectual property (IP) or developed internally. Many designers develop their own external processor bridge components for Qsys because it is usually relatively straightforward to bridge the Avalon fabric architecture to other bus protocols. The Component Editor tool, available in Qsys, is useful for creating IP such as external processor bridges.



For a list of bridge IP available from Altera, refer to the [Intellectual Property & Reference Designs](#) page of the Altera website.

## The `cpu_resetrequest` Signal

In versions 6.0 and later of the Nios II processor, an optional `cpu_resetrequest` signal is available to control the reset state of the processor. This signal differs from the normal Qsys system-wide reset signal `reset_n`—the `cpu_resetrequest` signal resets the Nios II processor only. The rest of the Qsys system remains operational. This signal holds the Nios II processor in reset while code is moved into the Nios II program memory.

The `cpu_resetrequest` signal does not cause the Nios II processor to enter the reset state immediately. When `cpu_resetrequest` is held high, the Nios II processor finishes executing any instructions currently in the pipeline, then enters reset. This process may take an indeterminate number of clock cycles, so a status signal `cpu_resettaken` is driven high by the Nios II processor when it reaches the reset state. The processor holds this signal high for one cycle. The `cpu_resettaken` signal continues to assert periodically while the `cpu_resetrequest` signal is held high.

To enable the `cpu_resetrequest` signal, open a project in Qsys that contains a Nios II processor. Double-click the Nios II component to open the Nios II MegaWizard interface, then click the **Advanced Features** page. Turn on **Include `cpu_resetrequest` and `cpu_resettaken` signals** to enable the signals. They appear as ports on your top-level Qsys system after you regenerate the system.

## Nios II Reset Address

The Nios II reset address is the address of the first instruction the processor executes after it is released from reset. Therefore, in a Nios II system capable of externally controlled boot, the Nios II reset address must point to a writable memory (RAM). This class of reset address is typically not what you want in a traditional boot scenario, but in the external boot control situation described in this section, it is important that the Nios II reset address point to RAM.

The Nios II reset address must point to RAM because, to direct the Nios II processor to the application code that was just copied into RAM, the external processor must be able to write the first instruction (or instructions) that the Nios II processor executes upon reset. Typically, the instruction written to the reset address is an unconditional branch (`br`) to the entry point of the application.

You can choose any unused 32-bit location in RAM as the reset address for the Nios II processor, but the base address (offset 0x0) of the Nios II program memory—the memory region that contains the `.text` section—is usually a good choice. By default, the Nios II exception table is placed at offset 0x20 in the program memory, and the remainder of the application code is placed following the exception table in consecutive memory. This arrangement leaves offsets 0x0 through 0x1C available. A reset address at offset 0x0 guarantees that the difference between the reset address and the application entry point—assumed to be early in the application code—never exceeds 64 Kbytes, as required for this process to work. For a description of why the difference cannot exceed 64 Kbytes, see the discussion of instruction step 1 on [page 31](#).

## One-Bit PIO Peripheral

A one-bit PIO peripheral is needed to control the Nios II `cpu_resetrequest` signal from the external processor. The external processor accesses the Avalon-mapped PIO peripheral through the external processor bridge. The external processor writes the value 1 to the PIO to assert the `cpu_resetrequest` pin, or the value 0 to de-assert it.

The external processor can also read the state of the `cpu_resettaken` signal using the same PIO peripheral. However, the Nios II processor asserts the `cpu_resettaken` signal for only one clock cycle at a time. Therefore, sampling this signal from software to see when reset has been achieved does not work. The signal can easily assert and de-assert again between samples, so that a valid assertion of `cpu_resettaken` by the Nios II processor might never be captured by the external processor.

The PIO component included with Qsys includes an edge-capture feature to use in this situation. The edge-capture feature sets a bit in the edge-capture register of the PIO whenever an edge of the predefined type is seen on that bit of the PIO's input port. The external processor can read the edge-capture register any time after it asserts `cpu_resetrequest`. If the `cpu_resettaken` signal was asserted any time since the `cpu_resetrequest` assertion, the relevant bit in the PIO's edge-capture register is set.

To add a PIO component configured to use the edge-capture feature to detect assertions of `cpu_resettaken` to your system, perform the following steps:

1. Open your system in Qsys.
2. On the **System Contents** tab, under **Peripherals**, and then under **Microcontroller Peripherals**, click the **PIO (Parallel I/O)** component.
3. Click **Add**.
4. In the **PIO MegaWizard** interface, set the width to one bit and select **InOut for Directions**.
5. Under the **Edge capture register** section, check the **Synchronously Capture** box, and then select **RISING** for Edge type.
6. Click **Finish** to add the PIO component to your system.

Your system now contains a PIO component capable of asserting the Nios II `cpu_resetrequest` signal and detecting rising edges on the `cpu_resettaken` signal.



Qsys does not automatically connect the input and output ports of the PIO component to the Nios II `cpu_resettaken` and `cpu_resetrequest` signals. After Qsys generation, you must make these connections at the top level in the Quartus II project.

## The Boot Process

Now that you have learned the important hardware aspects of externally controlling the Nios II boot process, this section describes the entire boot process from the perspective of the software running on the external processor.

### Boot Images

The procedure described here assumes you have a Nios II boot image in the format described in [“Boot Images” on page 6](#).

### Example C Code

In the directory `boot_copier_src/app/external_boot`, you can find sample C source code that you can run on an external processor to control the boot of a Nios II processor. The code is heavily commented, making it relatively easy to modify and customize. The example code happens to retrieve the boot image from offset 0x0 of a CFI flash, but in a real system, the boot image could come from anywhere. That part of the process is left to your discretion.

### External Boot Flow

The following section describes the boot flow implemented in the example C code mentioned in the previous section. These steps are written from the perspective of software running on an external processor which is responsible for controlling the Nios II boot process.

1. Retrieve the Nios II boot image.

The software can retrieve the Nios II boot image any number of ways. Common methods include reading the boot image from non-volatile storage such as hard disk or flash memory, downloading it over an Ethernet connection, or passing in a pointer to its location in RAM. Most important is that the image be locally accessible in its entirety before you attempt to unpack it and copy it to Nios II program memory.

1. Hold the Nios II processor in reset using the one-bit PIO, by performing the following actions:
  - Write any 32-bit value to offset 0x3 of the PIO component to clear the edge-capture register. Using the edge-capture register to detect when the `cpu_resettaken` signal goes high requires that you clear the edge-capture register first to ensure the register value does not represent an edge event that occurred in the past.
  - Write the value 1 to offset 0x0 in the PIO component to assert the Nios II `cpu_resetrequest` signal.
  - Continuously poll offset 0x3 of the PIO component until it holds the value 1. This value indicates that the Nios II `cpu_resettaken` signal transitioned high, which ensures the Nios II processor is now in a reset state and you can safely begin copying application code to its program memory.
2. Copy the application to its destination address in memory space.

Parse the boot record to copy each section of the application code to its appropriate location in Nios II program memory. The final individual record in the boot record is a jump record. Be sure to save the jump value, which contains the entry point of the application code. In the next step, you must direct the Nios II processor to the application entry point.

1. Construct a branch instruction to place at the Nios II reset address.

Constructing a Nios II branch (*br*) instruction allows Nios II to branch from its reset address to the entry point of the application code. Because the Nios II branch instruction is relative, meaning it branches relative to the current instruction, you need to know both the Nios II reset address, and the application entry point address.

In the example code, the Nios II reset address is simply defined at the top of the file. If your Nios II reset address changes, you must also change the relevant `#define` in the example code.

Subtract the reset address from the entry point address (saved in the previous step) to obtain the offset. For Nios II pipelining, the branch instruction actually requires the offset to be relative to the next instruction, so subtract 4 from the offset to obtain the actual offset needed for the instruction.

 Because all Nios II instructions are 32 bits, every address and offset must be a multiple of 4.

 The offset used in the branch instruction is 16 bits, so your reset address must be less than 64Kbytes away from the application entry point in memory space.

Using the branch instruction encoding shown in [Table 2](#), construct a branch instruction from the offset. The following C statements create a properly encoded instruction from the reset address and entry point:

```
int offset = entry_point - reset_address;
unsigned int inst = ((offset - 4) << 6) | 0x6;
```

**Table 2. Nios II Branch Instruction Encoding**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0				0				16-bit offset - 4												0x06											

1. Write the branch instruction to the Nios II reset address.
2. Release the Nios II processor from reset.

Write a zero to offset `0x0` of the PIO peripheral to deassert the Nios II `cpu_resetrequest` signal. The Nios II processor should come out of reset, execute the branch instruction, branch to the entry point of the application, and begin to execute it.

Booting is now complete. The Nios II processor is off and running, so the external processor can go about its other system tasks.

## Document Revision History

Table 3 shows the revision history for this document.

**Table 3. Document Revision History**

Date	Version	Changes
March 2014	2.2	<ul style="list-style-type: none"> <li>■ Updated “Default Nios II Boot Copier” section to include the EPCQ variant.</li> <li>■ Updated “The Default EPCS/EPCQ Boot Copier” section to describe behavior for IV-series and earlier and V-series and later devices.</li> <li>■ Globally changed EPCS to EPCS/EPCQ</li> </ul>
May 2011	2.1	<ul style="list-style-type: none"> <li>■ Replaced SOPC Builder with Qsys</li> <li>■ Updated Figure 3 on page 18 and Figure 4 on page 23</li> </ul>
August 2010	2.0	Updated for the Quartus II v10.0 software, the Nios II Software Build Tools for Eclipse, and the Nios II Ethernet Standard design example.
September 2008	1.1	Updated for the Quartus II v8.0 software and the Nios II software build tools development flow. New design examples target the Altera Nios II Embedded Evaluation Kit, Cyclone III Edition.
November 2007	1.0	Initial release.