

Overview

The NicheStack[®] TCP/IP Stack - Nios[®] II Edition is a small-footprint implementation of the transmission control protocol/Internet protocol (TCP/IP) suite. The focus of the NicheStack TCP/IP Stack implementation is to reduce resource usage while providing a full-featured TCP/IP stack. The NicheStack TCP/IP Stack is designed for use in embedded systems with small memory footprints, making it suitable for Nios[®] II processor systems.

Altera[®] provides the NicheStack TCP/IP Stack as a software component, available through the Nios II Integrated Development Environment (IDE), and the Nios II board support package (BSP) generator, which you can add to your system library or BSP. The NicheStack TCP/IP Stack includes these features:

- Internet Protocol (IP) including packet forwarding over multiple network interfaces
- Internet control message protocol (ICMP) for network maintenance and debugging
- User datagram protocol (UDP)
- Transmission Control Protocol (TCP) with congestion control, round trip time (RTT) estimation, and fast recovery and retransmit
- Dynamic host configuration protocol (DHCP)
- Address resolution protocol (ARP) for Ethernet
- Standard sockets application programming interface (API)

This chapter discusses the details of how to use the NicheStack TCP/IP Stack for the Nios II processor only. This chapter contains the following sections:

- [“Prerequisites” on page 11-1](#)
- [“Introduction” on page 11-2](#)
- [“Other TCP/IP Stack Providers” on page 11-3](#)
- [“Using the NicheStack TCP/IP Stack” on page 11-3](#)
- [“Configuring the NicheStack TCP/IP Stack in the Nios II IDE” on page 11-10](#)
- [“Further Information” on page 11-12](#)
- [“Known Limitations” on page 11-12](#)

Prerequisites

To make the best use of information in this chapter, you need have basic familiarity with these topics:

- Sockets. There are a number of books on the topic of programming with sockets. Two good texts are *Unix Network Programming* by Richard Stevens and *Internetworking with TCP/IP Volume 3* by Douglas Comer.
- The Nios II Embedded Design Suite (EDS). Refer to the *Nios II Software Developer's Handbook* for full information on the Nios II EDS.
- The MicroC/OS-II real time operating system (RTOS). To learn about MicroC/OS-II, refer to the *Using MicroC/OS-II RTOS with the Nios II Processor Tutorial*.

Introduction

Altera provides the Nios II implementation of the NicheStack TCP/IP Stack, including source code, in the Nios II EDS. The NicheStack TCP/IP Stack provides you with immediate access to a stack for Ethernet connectivity for the Nios II processor. The Altera implementation of the NicheStack TCP/IP Stack includes an API wrapper, providing the standard, well documented socket API.

The NicheStack TCP/IP Stack uses the MicroC/OS-II RTOS multithreaded environment. Therefore, to use the NicheStack TCP/IP Stack with the Nios II EDS, you must base your C/C++ project on the MicroC/OS-II RTOS. Naturally, the Nios II processor system must also contain an Ethernet interface, or media access controller (MAC). The Altera-provided NicheStack TCP/IP Stack includes driver support for the SMSC lan91c111 MAC/PHY device and Altera Triple Speed Ethernet MegaCore function. The Nios II Embedded Design Suite includes hardware for both MACs, plus an evaluation copy of the Triple Speed Ethernet MegaCore. The NicheStack TCP/IP Stack driver is interrupt-based, so you must ensure that interrupts for the Ethernet component are connected.

Altera's implementation of the NicheStack TCP/IP Stack is based on the hardware abstraction layer (HAL) generic Ethernet device model. By virtue of the generic device model, you can write a new driver to support any target Ethernet MAC, and maintain the consistent HAL and sockets API to access the hardware.



For details on writing an Ethernet device driver, refer to the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*.

The NicheStack TCP/IP Stack Files and Directories

You need not edit the NicheStack TCP/IP Stack source code to use the stack in a C/C++ program using the Nios II IDE. Nonetheless, Altera provides the source code for your reference. By default the files are

installed with the Nios II EDS in the *<Nios II EDS install path>/components/altera_niche/UCOSII* directory. For the sake of brevity, this chapter refers to this directory as *<niche path>*.

The directory format of the stack tries to maintain the original code as much as possible under the *<niche path>/src/downloads* directory for ease of upgrading to more recent versions of the NicheStack TCP/IP Stack. The *<niche path>/src/downloads/packages* directory contains the original NicheStack TCP/IP Stack source code and documentation; the *<niche path>/src/downloads/30src* directory contains code specific to the Nios II implementation of the NicheStack TCP/IP Stack, including source code supporting MicroC/OS-II.



The reference manual for the NicheStack TCP/IP Stack is available at www.altera.com/literature/lit-nio2.jsp, under **Other Related Documentation**.

Altera's implementation of the NicheStack TCP/IP Stack is based on version 3.0 of the protocol stack, with wrappers placed around the code to integrate it to the HAL system library.

Licensing

The NicheStack TCP/IP Stack is a TCP/IP protocol stack created by InterNiche Technologies, Inc. You can license the NicheStack TCP/IP Stack from Altera by going to www.altera.com/nichestack.



You can license other protocol stacks directly from InterNiche. Refer to the InterNiche website, www.interniche.com, for details.

Other TCP/IP Stack Providers

Other third party vendors also provide Ethernet support for the Nios II processor. Notably, third party RTOS vendors often offer Ethernet modules for their particular RTOS frameworks.



For up-to-date information on products available from third party providers, visit Altera's Embedded Software Partners page at: www.altera.com/products/software/partners/embedded/embedded-partners.html.

Using the NicheStack TCP/IP Stack

This section discusses how to include the NicheStack TCP/IP Stack in a Nios II program.

The primary interface to the NicheStack TCP/IP Stack is the standard sockets interface. In addition, you call the following functions to initialize the stack and drivers:

- `alt_iniche_init()`
- `netmain()`

You also use the global variable `iniche_net_ready` in the initialization process.

You must provide the following simple functions, which the HAL system code calls to obtain the MAC address and IP address:

- `get_mac_addr()`
- `get_ip_addr()`

Nios II System Requirements

To use the NicheStack TCP/IP Stack, your Nios II system must meet the following requirements:

- The system hardware generated in SOPC Builder must include an Ethernet interface with interrupts enabled
- The system library must be based on MicroC/OS-II
- The MicroC/OS-II RTOS must be configured to have the following enabled:
 - TimeManagement / OSTimeTickHook must be enabled
 - Maximum Number of Tasks must be 4 or higher
- The system clock timer must be set to point to an appropriate timer device.

The NicheStack TCP/IP Stack Tasks

The NicheStack TCP/IP Stack, in its standard Nios II configuration, consists of two fundamental tasks. Each of these tasks consumes a MicroC/OS-II thread resource, along with some memory for the thread's stack. These tasks run continuously in addition to the tasks that your program creates.

1. The NicheStack main task, `tk_netmain()` — After initialization, this task sleeps until a new packet is available for processing. Packets are received by an interrupt service routine (ISR). When the ISR receives a packet, it places it in the receive queue, and wakes up the main task.
2. The NicheStack tick task, `tk_nettick()` — This task wakes up periodically to monitor for time-out conditions.

These tasks are started when the initialization process succeeds in the `netmain()` function, as described in [“netmain\(\)” on page 11–5](#).



You can modify the task priority and stack sizes by using `#define` statements in the configuration file `ippport.h`. Additional system tasks might be created if you enable other options in the NicheStack TCP/IP Stack by editing `ippport.h`.

Initializing the Stack

Before you initialize the stack, start the MicroC/OS-II scheduler by calling `OSStart()` from `main()`. Perform stack initialization in a high priority task, to ensure that your code does not attempt further initialization until RTOS is running and I/O drivers are available.

To initialize the stack, call the functions `alt_iniche_init()` and `netmain()`. Global variable `iniche_net_ready` is set `true` when stack initialization is complete.



Make sure that your code does not use the sockets interface until `iniche_net_ready` is set to `true`. For example, call `alt_iniche_init()` and `netmain()` from the highest priority task, and wait for `iniche_net_ready` before allowing other tasks to run, as shown in Example 11-1 on page 11-6.

alt_iniche_init()

`alt_iniche_init()` initializes the stack for use with the MicroC/OS II operating system. The prototype for `alt_iniche_init()` is:

```
void alt_iniche_init(void)
```

`alt_iniche_init()` returns nothing and has no parameters.

netmain()

`netmain()` is responsible for initializing and launching the NicheStack tasks. The prototype for `netmain()` is:

```
void netmain(void)
```

`netmain()` returns nothing and has no parameters.

iniche_net_ready

When the NicheStack stack has completed initialization, it sets the global variable `iniche_net_ready` to a non-zero value.



Do not call any NicheStack API functions (other than for initialization) until `iniche_net_ready` is `true`.

Example 11-1 illustrates the use of `iniche_net_ready` to wait until the network stack has completed initialization:

Example 11-1. Instantiating the NicheStack TCP/IP Stack

```
void SSSInitialTask(void *task_data)
{
    INT8U error_code;

    alt_iniche_init();
    netmain();

    while (!iniche_net_ready)
        TK_SLEEP(1);

    /* Now that the stack is running, perform the application
       initialization steps */

    .
    .
    .
}
```

Macro `TK_SLEEP()` is part of the NicheStack TCP/IP Stack OS porting layer.

get_mac_addr() and get_ip_addr()

The NicheStack TCP/IP Stack system code calls `get_mac_addr()` and `get_ip_addr()` during the device initialization process. These functions are necessary for the system code to set the MAC and IP addresses for the network interface, which you select through **MAC interface** in the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box. Because you write these functions yourself, your system has the flexibility to store the MAC address and IP address in an arbitrary location, rather than a fixed location hard coded in the device driver. For example, some systems might store the MAC address in flash memory, while others might have the MAC address in onchip embedded memory.

Both functions take as parameters device structures used internally by the NicheStack TCP/IP Stack. However, you do not need to know the details of the structures. You only need to know enough to fill in the MAC and IP addresses.

The prototype for `get_mac_addr()` is:

```
int get_mac_addr(NET net, unsigned char mac_addr[6]);
```

Inside the function, you must fill in `mac_addr` with the MAC address.

The prototype for `get_mac_addr()` is in the header file `<iniche path>/inc/alt_iniche_dev.h`. The `NET` structure is defined in the `<iniche path>/src/downloads/30src/h/net.h` file.

Example 11–2 shows an implementation of `get_mac_addr()`. For demonstration purposes only, the MAC address is stored at address `CUSTOM_MAC_ADDR` in this example. There is no error checking in this example. In a real application, if there is an error, `get_mac_addr()` returns -1.

Example 11–2. An Implementation of `get_mac_addr()`

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
#include <io.h>
int get_mac_addr(NET net, unsigned char mac_addr[6])
{
    int ret_code = -1;

    /* Read the 6-byte MAC address from wherever it is stored */
    mac_addr[0] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 4);
    mac_addr[1] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 5);
    mac_addr[2] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 6);
    mac_addr[3] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 7);
    mac_addr[4] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 8);
    mac_addr[5] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 9);
    ret_code = ERR_OK;

    return ret_code;
}
```

You need to write the function `get_ip_addr()` to assign the IP address of the protocol stack. Your program can either assign a static address, or request for DHCP to find an IP address. The function prototype for `get_ip_addr()` is:

```
int get_ip_addr(alt_iniche_dev* p_dev,
               ip_addr*      ipaddr,
               ip_addr*      netmask,
               ip_addr*      gw,
               int*          use_dhcp);
```

`get_ip_addr()` sets the return parameters as follows:

```
IP4_ADDR(ipaddr, IPADDR0, IPADDR1, IPADDR2, IPADDR3);
IP4_ADDR(gw, GWADDR0, GWADDR1, GWADDR2, GWADDR3);
IP4_ADDR(netmask, MSKADDR0, MSKADDR1, MSKADDR2, MSKADDR3);
```

For the dummy variables `IP_ADDR0-3`, substitute expressions for bytes 0-3 of the IP address. For `GWADDR0-3`, substitute the bytes of the gateway address. For `MSKADDR0-3`, substitute the bytes of the network mask. For example, the following statement sets `ip_addr` to IP address 137.57.136.2:

```
IP4_ADDR ( ip_addr, 137, 57, 136, 2 );
```

To enable DHCP, include the line:

```
*use_dhcp = 1;
```

The NicheStack TCP/IP stack attempts to get an IP address from the server. If the server does not provide an IP address within 30 seconds, the stack times out and uses the default settings specified in the `IP4_ADDR()` function calls.

To assign a static IP address, include the lines:

```
*use_dhcp = 0;
```

The prototype for `get_ip_addr()` is in the header file `<niche path>/incl/alt_iniche_dev.h`.

Example 11-3 shows an implementation of `get_ip_addr()` and shows a list of the necessary include files.

There is no error checking in this example. In a real application, you might need to return -1 on error.

Example 11-3. An Implementation of `get_ip_addr()`

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
int get_ip_addr(alt_iniche_dev *p_dev,
               ip_addr* ipaddr,
               ip_addr* netmask,
               ip_addr* gw,
               int* use_dhcp)
{
    int ret_code = -1;
```

```

/*
 * The name here is the device name defined in system.h
 */
if (!strcmp(p_dev->name, "/dev/" INICHE_DEFAULT_IF))
{
    /* The following is the default IP address if DHCP
       fails, or the static IP address if DHCP_CLIENT is
       undefined. */
    IP4_ADDR(&ipaddr, 10, 1, 1, 3);
    /* Assign the Default Gateway Address */
    IP4_ADDR(&gw, 10, 1, 1, 254);
    /* Assign the Netmask */
    IP4_ADDR(&netmask, 255, 255, 255, 0);

#ifdef DHCP_CLIENT
    *use_dhcp = 1;
#else
    *use_dhcp = 0;
#endif /* DHCP_CLIENT */

    ret_code = ERR_OK;
}
return ret_code;
}

```

INICHE_DEFAULT_IF, defined in `system.h`, identifies the network interface that you defined in SOPC Builder. In the Nios II IDE, you can set INICHE_DEFAULT_IF through the **MAC interface** control in the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box. In the Nios II BSP generator, use the `iniche_default_if` BSP setting.

DHCP_CLIENT, also defined in `system.h`, specifies whether to use the DHCP client application to obtain an IP address. You can set or clear this setting in the Nios II IDE (with the **Use DHCP to automatically assign IP address** check box), or through the Nios II BSP generator (with the `dhcp_client` setting).

Calling the Sockets Interface

After initializing your Ethernet device, use the sockets API in the remainder of your program to access the IP stack.

To create a new task that talks to the IP stack using the sockets API, you must use the function `TK_NEWTASK()`. The `TK_NEWTASK()` function is part of the NicheStack TCP/IP Stack OS porting layer. `TK_NEWTASK()` calls the MicroC/OS-II `OSTaskCreate()` function to create a thread, and performs some other actions specific to the NicheStack TCP/IP Stack.

The prototype for `TK_NEWTASK()` is:

```
int TK_NEWTASK(struct inet_task_info* nettask);
```

It is in `<iniche path>/src/downloads/30src/nios2/osport.h`. You can include this header file as follows:

```
#include "osport.h"
```

You can find other details of the OS porting layer in the `osport.c` file in the NicheStack TCP/IP Stack component directory, `<iniche path>/src/downloads/30src/nios2/`.



For more information on how to use `TK_NEWTASK()` in an application, refer to the *Using the NicheStack® TCP/IP Stack - Nios II Edition Tutorial*.

Configuring the NicheStack TCP/IP Stack in the Nios II IDE

The NicheStack TCP/IP Stack has many options that you can configure using `#define` directives in the file `ipport.h`. The Nios II integrated development environment (IDE) allows you to configure certain options (i.e. modify the `#defines` in `system.h`) without editing source code. The most commonly accessed options are available through the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box.

There are some less frequently used options that are not accessible through the IDE. If you need to modify these options, you must use the Nios II BSP Generator, or edit the `ipport.h` file manually.



For further information about the Nios II BSP Generator, refer to the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

You can find `ipport.h` in the `debug/system_description` directory for your system library project.



If you modify the `ipport.h` file directly, be careful not to select the **Clean Project** build option in the Nios II IDE. Selecting **Clean Project** results in your modified `ipport.h` file being replaced with the starting template version of this file.

The following sections describe the features that you can configure via the Nios II IDE. The IDE provides a default value for each feature. In general, these values provide a good starting point, and you can later fine tune the values to meet the needs of your system.

NicheStack TCP/IP Stack General Settings

The ARP, UDP and IP protocols are always enabled. [Table 11–1](#) shows the protocol options.

<i>Table 11–1. Protocol Options</i>	
Option	Description
TCP	Enables and disables the transmission control protocol (TCP).

[Table 11–2](#) shows the global options, which affect the overall behavior of the TCP/IP stack.

<i>Table 11–2. Global Options</i>	
Option	Description
Use DHCP to automatically assign IP address	When on, the component use DHCP to acquire an IP address. When off, you must assign a static IP address.
Enable statistics	When this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in <code>mib</code> structures defined in various header files in directory <code><iniche path>/src/downloads/30src/h</code> . For details on <code>mib</code> structures, refer to the NicheStack documentation.
MAC interface	If the IP stack has more than one network interface, this parameter indicates which interface to use. See “Known Limitations” on page 11–12 .

IP Options

[Table 11–4](#) shows the IP options.

<i>Table 11–3. IP Options</i>	
Option	Description
Forward IP packets	When there is more than one network interface, if this option is turned on, and the IP stack for one interface receives packets not addressed to it, it forwards the packet out of the other interface. See “Known Limitations” on page 11–12 .
Reassemble IP packet fragments	If this option is turned on, the NicheStack TCP/IP Stack reassembles IP packet fragments into full IP packets. Otherwise, it discards IP packet fragments. This topic is explained in <i>Unix Network Programming</i> by Richard Stevens.

TCP Options

Table 11–4 shows the TCP options, which are only available with the TCP option is turned on.

<i>Table 11–4. TCP Options</i>	
Option	Description
Use TCP zero copy	This option enables the NicheStack zero copy TCP API. This option allows you to eliminate buffer-to-buffer copies when using the NicheStack TCP/IP Stack. For details, see the NicheStack reference manual. You must modify your application code to take advantage of the zero copy API.

Further Information

For further information about the Altera NicheStack implementation, refer to the *Using the NicheStack® TCP/IP Stack - Nios II Edition Tutorial*. The tutorial provides in-depth information about the NicheStack TCP/IP Stack, and illustrates how to use it in a networking application.

For details about NicheStack, see the NicheStack TCP/IP Stack reference manual, available at www.altera.com/literature/lit-nio2.jsp, under **Other Related Documentation**.

Known Limitations

Although the NicheStack code contains features intended to support multiple network interfaces, these features are not tested. See the NicheStack TCP/IP Stack reference manual and source code for information about multiple network interface support.

Referenced Documents

This chapter references the following documents:

- *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*
- *NicheStack TCP/IP Stack documentation* available at *Literature: Nios II Processor, Other Related Documentation*
- *Using the NicheStack TCP/IP Stack - Nios II Edition Tutorial*
- *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 11–5 shows the revision history for this document.

Date and Document Version	Changes Made	Summary of Changes
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	<ul style="list-style-type: none">• Chapter 10 was formerly chapter 9.• Minor clarifications added to content.• Added table of contents to Overview section.• Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Initial Release.	

