

The NicheStack[®] TCP/IP Stack - Nios[®] II Edition is a small-footprint implementation of the 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 package that you can add to your board support package (BSP), available through the Nios II Software Build Tools (SBT). 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 program 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 for Understanding the NicheStack TCP/IP Stack”](#) on page 11-2
- [“Introduction to the NicheStack TCP/IP Stack - Nios II Edition”](#) on page 11-2
- [“Other TCP/IP Stack Providers for the Nios II Processor”](#) on page 11-3
- [“Using the NicheStack TCP/IP Stack - Nios II Edition”](#) on page 11-3
- [“Configuring the NicheStack TCP/IP Stack in a Nios II Program”](#) on page 11-9
- [“Further Information”](#) on page 11-10
- [“Known Limitations”](#) on page 11-10

Prerequisites for Understanding the NicheStack TCP/IP Stack

 To make the best use of information in this chapter, you should be familiar with these topics:

- Sockets. Several books are available 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 *Overview* chapter of the *Nios II Software Developer's Handbook* for more information about the Nios II EDS.
- The MicroC/OS-II RTOS. To learn about MicroC/OS-II, refer to the *MicroC/OS-II Real-Time Operating System* chapter of the *Nios II Software Developer's Handbook*, or to the *Using MicroC/OS-II RTOS with the Nios II Processor Tutorial*.

Introduction to the NicheStack TCP/IP Stack - Nios II Edition


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. Altera's 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. The Nios II processor system must also contain an Ethernet interface, or media access control (MAC). The Altera-provided NicheStack TCP/IP Stack includes driver support for the following two MACs:

- The SMSC lan91c111 device
- The Altera® Triple Speed Ethernet MegaCore® function

The Nios II Embedded Design Suite includes hardware for both MACs. 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. In 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 about writing an Ethernet device driver, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* 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 Nios II C/C++ program. 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_iniche/UCOSII` directory. For the sake of brevity, this chapter refers to this directory as `<iniche path>`.


Under `<iniche path>`, the original code is maintained—as much as possible—under the `<iniche path>/src/downloads` directory. This organization facilitates upgrading to more recent versions of the NicheStack TCP/IP Stack. The `<iniche path>/src/downloads/packages` directory contains the original NicheStack TCP/IP Stack source code and documentation; the `<iniche 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 on the [Literature: Nios II Processor](#) page of the Altera website, under **Other Related Documentation**.

Altera's implementation of the NicheStack TCP/IP Stack is based on version 3.1 of the protocol stack, with wrappers around the code to integrate it with the HAL.

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 [the Altera website](#).

-  You can license other protocol stacks directly from InterNiche. You can obtain details from InterNiche Technologies, Inc. (www.interniche.com)

Other TCP/IP Stack Providers for the Nios II Processor

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 about products available from third party providers, visit the [Embedded Software](#) page of the Altera website.

Using the NicheStack TCP/IP Stack - Nios II Edition

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 must include an Ethernet interface with interrupts enabled.
- The BSP must be based on MicroC/OS-II.
- The MicroC/OS-II RTOS must be configured to have the following settings:
 - 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. In addition to the tasks your program creates, the following tasks run continuously:

- **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.
- **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\(\)](#)”.




You can modify the task priority and stack sizes using `#define` statements in the configuration file `ippport.h`. You can create additional system tasks by enabling 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 the your code does not attempt further initialization until the 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.

 Ensure that your code does not use the sockets interface before `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](#).

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 operating system (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 with the `altera_iniche.iniche_default_if` BSP setting. 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 on-chip 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]);
```

You must implement the `get_mac_addr()` function to assign the MAC address to the `mac_addr` argument. Leave the `net` argument untouched.

The prototype for `get_mac_addr()` is in the header file `<iniche path>/incl/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()` must return -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 must 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 the 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 `<iniche 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 **Example 11-3**. In a real application, you might need to return -1 on error.

`INICHE_DEFAULT_IF`, defined in **system.h**, identifies the network interface that you defined at system generation time. You can control `INICHE_DEFAULT_IF` through 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 property with the `altera_iniche.dhcp_client` setting.

Calling the Sockets Interface

After you initialize 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 operating system (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);
```

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;
}
```

The prototype is defined 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 about 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 a Nios II Program

The NicheStack TCP/IP Stack has many options that you can configure using `#define` directives in the file `ippport.h`. The Nios II EDS allows you to configure certain options (that is, modify the `#defines` in `system.h`) without editing source code. The most commonly accessed options are available through a set of BSP options, identifiable by the prefix `altera_iniche`.

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

Some less-frequently-used options are not accessible through the BSP settings. If you need to modify these options, you must edit the `ippport.h` file manually.

You can find `ippport.h` in the `debug/system_description` directory for your BSP project.

The following sections describe the features that you can configure using the Nios II SBT. Both development flows provide 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.

Table 11-1. Protocol Options

Option	Description
TCP	Enables and disables the TCP.

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

Table 11-2. Global Options

Option	Description
Use DHCP to automatically assign IP address	If this option is turned on, the component uses DHCP to acquire an IP address. If this option is turned off, you must assign a static IP address.
Enable statistics	If 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 about <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. Refer to “ Known Limitations ” on page 11-10 .

IP Options

Table 11-3 shows the IP options.

Table 11-3. IP Options

Option	Description
Forward IP packets	If there is more than one network interface, this option is turned on, and the IP stack for one interface receives packets that are not addressed to it, the stack forwards the packet out of the other interface. Refer to “Known Limitations” on page 11-10.
Reassemble IP packet fragments	If this option is turned on, the NicheStack TCP/IP Stack reassembles IP packet fragments as 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 zero copy option, which is only available if the TCP option is turned on.

Table 11-4. TCP Options

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, refer to 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, refer to the NicheStack TCP/IP Stack reference manual, available on the [Literature: Nios II Processor](#) page of the Altera website, under **Other Related Documentation**.

Known Limitations

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

Document Revision History

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

Table 11-5. Document Revision History

Date	Version	Changes
May 2011	11.0.0	No change
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	<ul style="list-style-type: none"> ■ Introduced the Nios II Software Build Tools for Eclipse™. ■ Nios II IDE information removed to <i>Appendix A. Using the Nios II Integrated Development Environment</i>.
March 2009	9.0.0	<ul style="list-style-type: none"> ■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	<ul style="list-style-type: none"> ■ Minor clarifications added to content. ■ Added table of contents to Overview section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Initial release.

