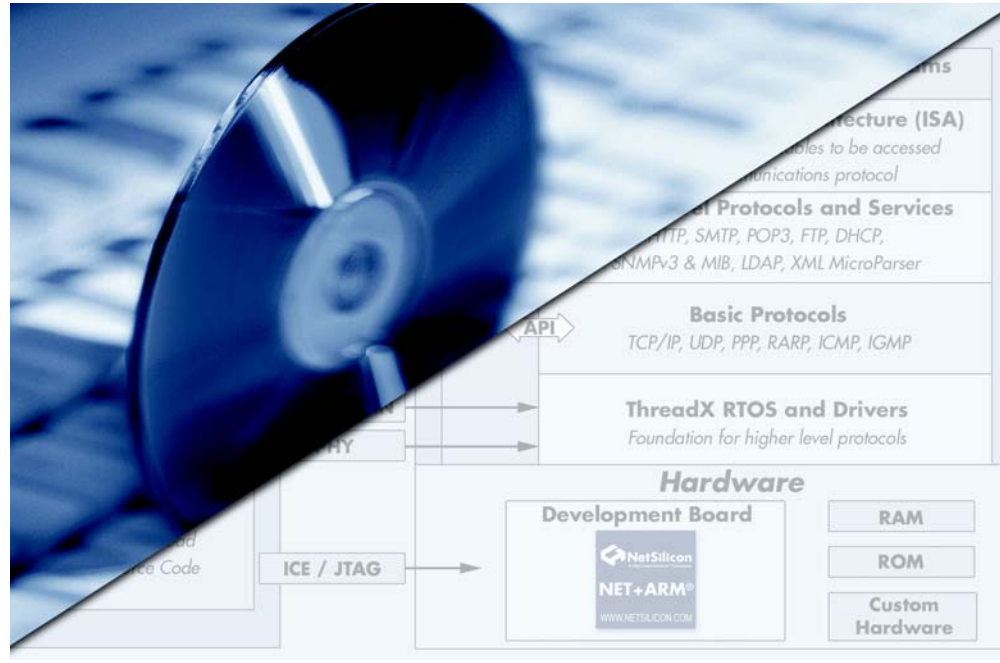
NetSilicon
A Digi International® Company

NET+OS BSP Software Reference Guide



NET+OS 5.0
8833442B

NET+OS BSP Software Reference Manual

Operating system/version: NET+OS 5.0
Part number/version: 8833442B
Release date: July 2002
www.netsilicon.com

Contents

Using This Guide

About this guide

This guide describes the application programmer interfaces (APIs) provided for the NET+OS board support package (BSP).

NET+OS, a network software suite optimized for the NET+ARM chip, is part of the NET+Works integrated product family.

Who should read this guide

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

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

- Be familiar with programming concepts and techniques, especially for drivers, network applications, and development systems
- Have sufficient system (user) privileges to perform the tasks described
- Have access to a computer system that meets NET+Works hardware and software requirements

What's in this guide

- *Chapter 1* is an overview of the how the various APIs are documented in this book and in other books in the NET+OS documentation set.
- *Chapters 2 – 14* describe the various BSP software APIs.

Conventions used in this guide

This table describes the typographic conventions used in this guide:

This convention...	Is used for...
italic type	Emphasis, new terms, variables, and document titles.
`monospaced type`	Filenames, pathnames, commands, and code examples.

Related documentation

- *NET+OS Getting Started Guide* explains how to install NET+OS with Green Hills or with GNU tools, and how to build your first application.
- *NET+OS User's Guide* describes how to use NET+OS to develop programs for your application and hardware.
- *NET+OS BSP Porting Guide* describes how to port the board support package (BSP) to a new hardware application, with either Green Hills Software or GNU tools.
- *NET+OS Application Software Reference Guide* provides descriptions of the APIs for network software support.
- *NET+OS Kernel User's Guide* describes the real-time NET+OS kernel services.

- Review the documentation CD-ROM that came with your development kit for information on third-party products and other components.
- Refer to the NET+Works hardare documentation for information appropriate to the chip you are using.

Customer support

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

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

Introduction

CHAPTER 1

This chapter describes the application programming interfaces (APIs) for the NET+OS board support package (BSP). These APIs provide interfaces to drivers, memory, utilities, and other board and system components.

Overview of NET+OS APIs

NET+OS provides three types of APIs, summarized in the following sections.

Core APIs

The *NET+OS Kernel User's Guide* descrbies core APIs which provide access to NET+OS kernel services:

- Task management for dynamic creation and deletion of tasks and control of task attributes
- Storage allocation of variable size segments and fixed size buffers

- Message queue service for general-purpose communication and synchronization
- Event and asynchronous signal devices
- Semaphore services
- Time management and timer services including maintenance of calendar time and date, timeout and wakeup of tasks, timeslice tracking and round-robin scheduling, and so on

Network APIs

The *NET+OS Software Application Reference Guide* describes network APIs which provide interfaces to Internet protocols and services:

- Advanced Web Server (AWS)
- FTP server
- FTP client
- E-mail services
- Domain name service (DNS)
- Simple network management protocol (SNMP)
- Point-to-point protocol (PPP)
- Fast IP
- Dynamic host configuration protocol (DHCP)
- Telnet server
- Sockets
- HTTP server
- LDAP services
- Management
- General-purpose and system access (security)

Board support APIs

This guide describes the following board support APIs in detail:

- Parallel port driver
- Serial port driver
- ENI driver
- LED control
- NVRAM support
- Flash memory support
- Serial number support
- Cache support
- ISR support
- System clock and timer support
- HDLC driver
- DMA driver
- SPI driver
- Serial EEPROM support

About the API descriptions

This guide describes the the API function calls using the following format:

- Name of the function and a brief description
- The format of the function call
- A table defining the function's arguments
- A table defining the function's return values

In some case, there are additional comments, usage notes, and code examples.

Deprecated functions

Some functions described in this manual are *deprecated* — that is, their use is not recommended, even though the functions may continue to be included for compatibility purposes. Deprecated functions are indicated with a dagger symbol (†).

Private structures

Routines or data structures described in header files but not discussed in this guide, are considered *private structures*. That is, they are *for NetSilicon internal use only* and their functioning is not guaranteed, nor are their prototypes. Using these private structures is strongly discouraged.

Parallel Port Driver API

CHAPTER 2

Overview

The board support package (BSP) includes a parallel port driver to support all four IEEE 1284 parallel ports implemented in the NetSilicon chip.

Although the chip supports four parallel ports, only the first two ports are physically on the NetSilicon development boards. However, the parallel port driver always supports all four ports internally.

The names of the parallel ports are set in the `devices.c` file. The default port names are:

- `/lpt/0`
- `/lpt/1`
- `/lpt/2`
- `/lpt/3`

To include the parallel port driver in your application, define the `BSP_INCLUDE_PARALLEL_DRIVER` constant in the `bspconf.h` file.

Include files

Using the parallel port driver API requires the following header files:

- `para_api.h`
- `netosio.h`

Resource usage

DMA channels 3 through 6 transfer data from the local device to the remote device (such as a printer). Although the chip reserves interrupts 17 though 21 and 26 through 29 for the parallel ports, they are not actually used by the driver.

The driver allocates approximately 40Kb of RAM for internal buffers and data structures. The parallel port driver uses timer 2 for internal timing.

The parallel ports and the ENI port use the same hardware in the NetSilicon chip. Therefore, you cannot use the parallel port driver and ENI driver simultaneously.

Bidirectional communications support

The parallel port driver implements support for 1284 bi-directional data transfer. Both 1284 nibble mode and extended capabilities port (ECP) mode are supported.

Supplied APIs for the parallel port driver

The standard device driver APIs are supported. By default, bidirectional communications are disabled. They must be enabled using an `ioctl` call before attempting to use the `read` function. To access the parallel ports, use the `open`, `write`, `read`, `close`, and `ioctl` functions.

Three `ioctl` calls are defined for the parallel port driver, as follows. The values are defined in `paradev.h`.

Setting 1284 nibble mode

Use LPT_SET_NYBBLE_MODE to select 1284 nibble mode. Set the *arg* parameter to 0. For example:

```
ioctl (fd, LPT_SET_NYBBLE_MODE, 0);
```

Setting 1284 ECP mode

Use LPT_SET_ECP_MODE to selects 1284 ECP mode. The *arg* parameter selects the ECP channel number — between 0 and 127.

For example, this call selects ECP mode on channel 0:

```
ioctl (fd, LPT_SET_ECP_MODE, 0);
```

If the printer does not support the selected mode, ioctl returns –1, and errno is set to LPT_MODE_NOT_SUPPORTED.

Initializing a printer

Use LPT_SEND_INIT to initialize an attached printer. The parallel port driver toggles the INIT signal on the parallel port, causing the attached printer to reinitialize.

Set the *arg* parameter should be set to 0.

For example, this code fragment generates an INIT pulse.

```
ioctl (fd, LPT_SEND_INIT, 0);
```

Serial Port Driver API

CHAPTER 3

Overview

The board support package (BSP) includes a serial port driver which supports two RS-232 serial ports implemented in the NetSilicon chip.

The chip supports two serial ports, both of which are physically on the NetSilicon development boards. The names of the serial ports are set in the `devices.c` file. The default port names are `/com/0` and `/com/1`, corresponding to `COM1` and `COM2`.

To include the serial port driver in your application, define the `BSP_INCLUDE_SERIAL_DRIVER` constant in the `bspconf.h` file.

Include files

Using the serial port driver API requires the following header files:

- `netosio.h`
- `netos_serl.h`

Resource usage

The NetSilicon chip reserves interrupts 12 though 15 and DMA channels 7 through 10 for the serial ports. DMA is used for high-speed baud rates. The serial port driver allocates approximately 4 Kb of RAM for internal buffers and data structures.

Bi-directional communications support

The serial port driver implements support for RS232 bi-directional data transfer. Hardware handshaking (RTS/CTS) is supported. On the development board, the serial ports are implemented as data terminal equipment (DTE) devices.

Supplied APIs for the serial port driver

The standard device driver APIs are supported. To access the serial ports, use the `open`, `write`, `read`, `close`, and `ioctl` functions.

Calling `open` initializes the serial device:

```
int open (const char *filename, int mode, ...);
```

By default, the filenames are `/com/0` and `/com/1`, corresponding to `COM1` and `COM2`.

The mode field specifies how a given file descriptor is opened. The BSP accepts these modes for this device:

```
O_RDONLY                /* open for reading only */
O_WRONLY                /* open for writing only */
O_RDWR                  /* open for reading or writing */
O_NONBLOCK              /* non-blocking */
O_DMA                   /* open for use DMA mode*/
```

When you use the O_DMA mode, the serial port read and write use the DMA channels to support the serial operation. The DMA operation for the serial port supports only hardware handshaking. If you use both serial ports, they should both choose to use or not use DMA at the same time. Internally, after DMA mode is chosen, DMA channels 7 and 8 are used for COM1 reading and writing, and DMA channel 9 and 10 are used for COM2 reading and writing. For DMA to work, hardware handshaking must be used for the SIO_HW_OPTS_SET option in the ioctl call.

DMA operations are generally used for speeds greater than 115200 baud. When the serial port is used under this speed, the standard serial port driver can handle most tasks. For example:

- To open the COM2 serial port with read/write access, in non-blocking mode, without DMA support:

  ```
  fd = open ("/com/1", (O_RDWR | O_ NONBLOCK));
  ```

  In this case, the serial port read and write return without waiting for the completion of the function call.

  Blocking mode is the default.

- To open COM2 with read/write access in DMA mode:

  ```
  fd = open ("/com/1", (O_RDWR | O_ DMA));
  ```

All mode constants are defined in netosio.h.

You can change the configuration of the serial port at any time by using the `ioctl` function. The following table provides a summary of the `ioctl` parameters:

Command	Argument type	Description
`SIO_BAUD_GET` `SIO_BAUD_SET`	address	Get/set the baud rate of serial port driver. Allowed rates are: 75, 150, 300, 600, 1200, 2400, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, and 115200. For DMA operations only, baud rates include 230400, 307200, and 460800.
`SIO_MODE_GET` `SIO_MODE_SET`	address integer	Get/set the operating mode of the serial port driver. `SIO_MODE_SET` is for polling mode. `SIO_MODE_INT` is interrupt mode. Currently, only interrupt mode is supported.
`SIO_HW_OPTS_GET` `SIO_HW_OPTS_SET`	address	Get/set the serial communication operation parameters (for example, number of data bits, number of stop bits, parity, and handshaking).
`SIO_GET_STATUS_DCD`	address	Get the state of the DCD bit of the serial status register.
`SIO_SET_STATUS_DTR`	address	Set the state of the DTR bit of the serial control register.
`SIO_TERM_SEND_CR`	address	If the argument is `TERM_SEND_CR_ON`, the serial port driver sends the `\r\n` pair whenever the current character to send is `\n`.
`SIO_TERM_ECHO`	address	If the argument is `TERM_ECHO_ON`, whenever a character is received, the same character is echoed back to the sender. If the argument is `TERM_ECHO_TRANSLATE`, the `\n` character is sent as `\r\n`.
`SIO_BAUD_ERROR_SET` `SIO_BAUD_ERROR_GET`	address	Get/set the acceptable baud rate error/ default to 1 (1%).

If the high-speed values for DMA mode are calculated based on an 18.432 MHz crystal and a system PLL multiplier of 5, the SYSCLK value is 29.491 MHz.

If SYSCLK changes, you must calculate the actual speed using the equation provided in the NetSilicon hardware documentation for the chip you are using. If you enter an incorrect baud rate setting, you may get an ERANGE error.

Options for setting the serial port

Serial baud rate options are in the form SIO_*n*_BAUD where *n* is one of the following:

1200	14400	115200
2400	19200	234000
4800	28800	307200
7200	38400	460800
9600	57600	1036800

Rates from 230400 and higher are for DMA only.

Include the following options in the `ioctl` option field for `SIO_HW_OPTS_GET` and `SIO_HW_OPTS_SET`. Select one option from each category:

Option category		Setting
Handshake method:	SIO_NO_HANDSHAKING	0x1
	SIO_HW_RTS_CTS_HANDSHAKING	0x0
	SIO_SW_HANDSHAKING	0x3
Data bit width:	SIO_FIVE_BIT_DATA_WIDTH	0x0
	SIO_SIX_BIT_DATA_WIDTH	0x4
	SIO_SEVEN_BIT_DATA_WIDTH	0x8
	SIO_EGHT_BIT_DATA_WIDTH	0xc
Data bit width:	SIO_ONE_STOP_BIT	0x0
	SIO_TWO_STOP_BITS	0x20
Parity option:	SIO_NO_PARITY	0x0
	SIO_EVEN_PARITY	0x40
	SIO_ODD_PARITY	0x80

Configuring the serial port

To configure the serial port, see the `serParam.h` file which contains the legal values used to set up the port. This header file includes the baud rates used in `SIO_BAUD_SET` and `SIO_BAUD_GET`. It also includes the values needed to set:

- Port parity
- Number of stop bits
- Handshaking
- Data width

These settings should be used in `SIO_HW_OPTS_SET` or returned in `SIO_HW_OPTS _GET`.

The following table lists some settings and their required SIO_HW_OPTS_SET parameters:

Settings	Required values
8 bit No parity 2 stop bits No handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_NO_PARITY SIO_TWO_STOP_BITS SIO_NO_HANDSHAKING
8 bit Odd parity 2 stop bits No handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_ODD_PARITY SIO_TWO_STOP_BITS SIO_NO_HANDSHAKING
8 bit Even parity 2 stop bits No handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_EVEN_PARITY SIO_TWO_STOP_BITS SIO_NO_HANDSHAKING
8 bit No parity 2 stop bits Hardware handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_NO_PARITY SIO_TWO_STOP_BITS SIO_HW_RTS_CTS_HANDSHAKING
8 bit No parity 1 stop bit Hardware handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_NO_PARITY SIO_ONE_STOP_BIT SIO_HW_RTS_CTS_HANDSHAKING
8 bit No parity 1 stop bit Hardware handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_ODD_PARITY SIO_ONE_STOP_BIT SIO_HW_RTS_CTS_HANDSHAKING
8 bit Even parity 1 stop bit Software handshaking	SIO_EIGHT_BIT_DATA_WIDTH SIO_EVEN_PARITY SIO_ONE_STOP_BIT SIO_SW_HANDSHAKING

Settings	Required values
7 bit	SIO_SEVEN_BIT_DATA_WIDTH
Even parity	SIO_EVEN_PARITY
2 stop bit	SIO_TWO_STOP_BITS
Software handshaking	SIO_SW_HANDSHAKING

ENI Driver API

CHAPTER 4

Overview

The embedded network interface (ENI) driver API lets you access the ENI hardware without having to deal with specific hardware details.

For example, the ENI hardware requires configuration and manipulation of the internal registers. The shared RAM interface requires that the RAM address and size be configured in one of the internal registers. The size of RAM allowed depends on the operation mode selected. The FIFO interface interacts with the DMA hardware, which requires configuration. Although you would need to deal with these issues when programming the ENI hardware directly, the ENI driver API hides much of the detail.

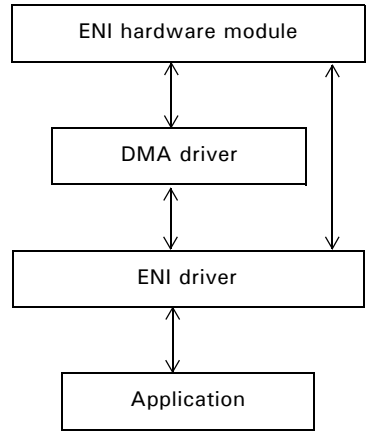


Figure 1: Relation of ENI hardware and internal registers

Include file

Using the ENI driver API requires the following header file:

```
eni_api.h
```

Also, your application should link with the following library file:

```
eni_lib.o
```

Summary of ENI API functions

Function	Description
`eniTransmitType`	Provided by the application and used by the ENI driver to release a message after it has been transmitted.
`eniReceiveType`	Provided by the application to process incoming data. The ENI driver calls this function whenever it receives data from the FIFO channel.
`eniTriggerType`	Provided by the application and called whenever an ENI interrupt is received.
`eniLoadDriver`	Initializes the ENI channels.

Function	Description
eniOpenChannel	Opens a channel for the ENI peripheral. The shared RAM is configured and ENI interrupt handler is registered.
eniCloseChannel	Closes the ENI channel. The channel cannot be closed if the ENI driver is still holding buffers owned by the application.
eniEnableFIFO	Enables the FIFO channel. Data activity in the channel resumes where it was left off before it halted.
eniDisableFIFO	Disables the FIFO channel. Data movement in the channel is halted until the channel is enabled again.
eniSupplyFIFO	Supplies received messages to the ENI driver. Each message contains a buffer used to store incoming data from the external device.
eniReadFIFO	Retrieves data received from the FIFO channel. If no message is pending, the application can specify the amount of time to poll for incoming data.
eniWriteFIFO	Transmits a message through the ENI FIFO. Each message must contain a buffer that is 32-bit aligned.
eniFlushFIFO	Flushes any buffers owned by the application that are still held by the ENI driver. The FIFO channel must be disabled before it can be flushed.
eniTriggerInterrupt	Triggers an ENI interrupt to the external device. The interrupt provides a way to synchronize information or communication between two processes running on separate processors.
eniEnableTrigger	Enables the ENI interrupt, which allows an external drive to trigger an interrupt.
eniDisableTrigger	Disables the ENI interrupt, which prevents an external device from triggering an interrupt.
eniGetSharedRAM	Returns the location and size of the ENI shared memory.

Using the ENI driver API

Calling `eniLoadDriver` initializes the ENI driver. Next, the internal hardware needs to be configured.

The `eniOpenChannel` function lets the application set up shared memory and the FIFO peripheral. Access to the shared memory and FIFO is determined by the mode selected.

Retrieving shared memory size and location

Shared memory is at a specified location in RAM that can be read from and written to by both the internal and external device. The address and size of shared memory can be retrieved by calling `eniGetSharedRAM`. Each device can use the memory to exchange important information with the other, which can include the current state and status of the device.

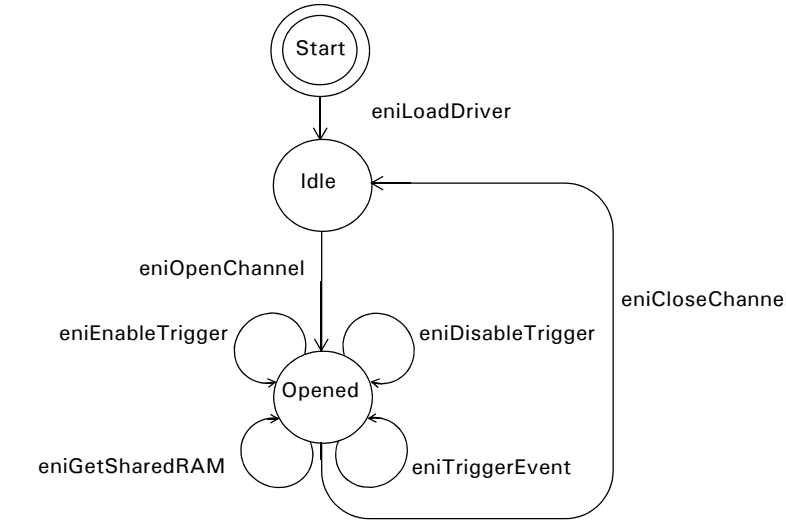


Figure 2: State diagram for the API used for shared RAM

Exchanging data using the FIFO interface

Another way to exchange data with an external device is the FIFO interface, as shown here:

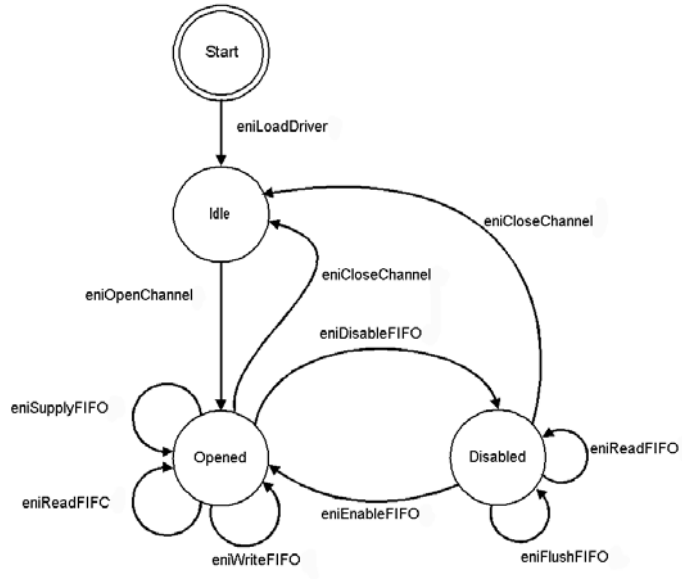


Figure 3: State diagram for the API used for FIFO

Data exchanged between the application and the ENI driver must be passed in the form of an `eniMessageType`:

```
typedef struct eniMessageType
{
    struct eniMessageStruct *next;
    void *src_addr
    void dst_addr
    long length
    long status
    long error_value
    long reserved [4]
} eniMessageType
```

The status field contains these values:

- ENI_SUCCESS
- ENI_DATA_ABORT
- ENI_CHANNEL_ERROR

Depending on the usage of the message, the *src_addr* and *dst_addr* fields contain the locations that data to be read from or written to. The buffer must be aligned on a 32-bit boundary. The *length* field contains the buffer size, which must be a multiple of 4 bytes. The *next* field enables messages to be chained, and is valid for outgoing messages only.

Triggering, disabling, and enabling an ENI interrupt

To avoid polling from each device, you can generate an interrupt whenever data is updated in a memory location. Calling `eniTriggerInterrupt` triggers the external device.

If a device is in a state from which it does not want to be disturbed by the external device, you can keep the interrupt from occurring by calling `eniDisableTrigger`.

When the device is ready to accept interrupts, you can re-enable it by calling `eniEnableTrigger`.

Sending messages

To send data out, call `eniWriteFIFO`. The *src_addr* field contains the location of outgoing data. The number of messages the FIFO handles depends on the transmit queue size configured. After the data has been transmitted, the ENI driver releases the buffer back to the application. The `eniTransmitRtn` callback provided by the application determines how the buffer is released.

Supplying messages

Before any data can be received from the external device, the application must supply the ENI driver with messages that contain allocated buffers using `eniSupplyFIFO`. The maximum number of messages that contain allocated buffers depends on the size of the receive queue configured. The ENI driver is ready to accept incoming data after allocated buffers have been provided. If an `rx_event_rtn` callback routine is defined, it is called automatically when incoming data is received.

Receiving messages

An application can manually retrieve the buffer pending in a queue by calling `eniReadFIFO`.

Disabling and enabling the FIFO channel

At times, the application may not want to receive or transmit any data to the external device. The FIFO channel can be disabled any time by calling `eniDisableFIFO`.

When the device is disabled, it can enable the FIFO by calling `eniEnableFIFO`.

Flushing messages

Any messages that are still held by the ENI driver can be retrieved before they are processed. The `eniFlushFIFO` releases all pending outgoing messages by calling the `eniTransmitRtn` callback. The allocated buffers for incoming messages are released automatically to the application if the `eniReceiveRtn` callback is defined. Any message flushed contains the value `ENI_DATA_ABORT` in the status field. The FIFO channel must be disabled before it can be flushed.

Closing the ENI channel

When the application no longer requires the ENI channel, it can be closed by calling `eniCloseChannel`. Before the ENI channel can be closed, the application must flush all unused and unprocessed buffers still held in the ENI driver for that channel.

Callback routines

The ENI driver supports several callback routines, which are registered when the ENI channel is opened. Because these routines are executed during an ISR, they cannot execute instructions that require extensive processing time. If you are unsure which functions can be called, you may want to have a thread running in the background to service your events. See the *NET+OS Kernel User's Guide* for a list of service calls that are allowed during an ISR.

This figure illustrates when `eniTriggerRtn`, `eniTransmitRtn`, and `eniTransmitRtn` are called:

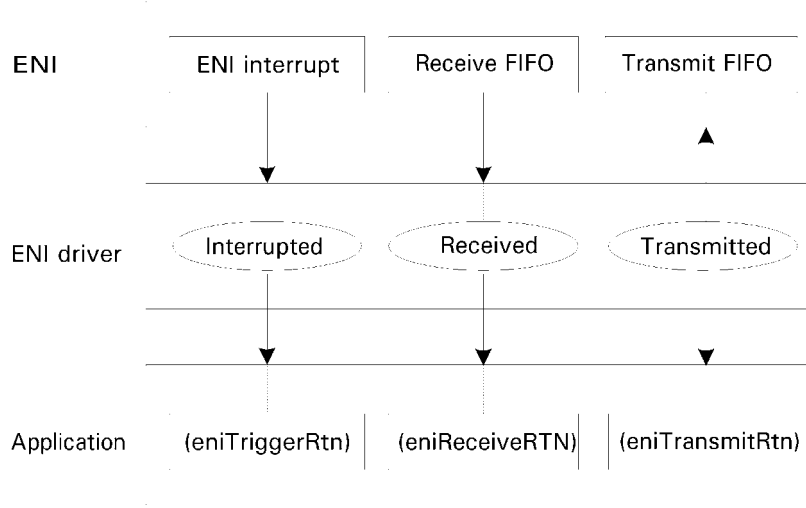


Figure 4: Application callback invoked by the ENI driver

ENI driver API functions

The following pages describe the ENI driver API functions.

eniTransmitType

This callback routine is provided by the application and used by the ENI driver to release a message.

After a message is transmitted, the ENI driver needs to return it to the application. This function is called within an ISR routine.

You cannot use function calls that could a require long execution time. This includes sleep time and waiting for semaphores.

Format

```
void (*eniTransmitType) (long channel_id,
       eniMessageType *message);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
message	Pointer to the message that held the recently transmitted data.

Return values

none

eniReceiveType

This callback routine is provided by the application to process incoming data. The ENI driver calls this routine whenever it receives data from the FIFO channel. Because this routine is called within an interrupt, code in the callback routine must not require extensive execution time.

Upon return of this routine, the ENI driver no longer owns the message. The application is responsible for releasing the buffer after it is done with it.

If the application does not want to process incoming data immediately, it can delay processing by not defining this routine. The application can access pending data by calling `eniReadFIFO` whenever it has time to process it.

Format

```
void (*eniReceiveType) (long channel_id,
        eniMessageType *message);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
message	Pointer to the message containing incoming data from the external device.

Return values

none

eniTriggerType

This callback routine is provided by the application and is called whenever an ENI interrupt is received.

The interrupt is triggered by an external device to synchronize communication between the two processes. The ENI interrupt is cleared and reset upon return from this routine.

You cannot use function calls that could a require long execution time.

Format

```
void (*eniTriggerType) (long channel_id);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.

Return values

none

eniLoadDriver

Initializes the ENI channels.

Format

```
int eniLoadDriver (void);
```

Arguments

none

Return values

ENI_SUCCESS

eniOpenChannel

Opens a channel for the ENI peripheral. The shared RAM is configured, and ENI interrupt handler is registered. If FIFO mode is selected, the FIFO channel is set up for receiving messages from and transmitting messages to the external device. Currently, only one ENI channel is supported.

The ENI driver overrides one of the four ENI modes configured by hardware. The channel cannot be opened if it is already configured for IEEE 1284 host port hardware.

Format

```
int eniOpenChannel (long channel_id,long mode,
        long *ram_addr,long ram_size, long rx_queue_size,
        long tx_queue_size, long data_size,
        long transfer_size,
        eniTriggerType trigger_event_rtn,
        eniTransmitType tx_event_rtn,
        eniReceiveType rx_event_rtn);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
mode	Mode of operation for ENI: ■ ENI_SHARED_RAM_16_BIT ■ ENI_SHARED_RAM_8_BIT ■ ENI_FIFO_MODE_16_BIT ■ ENI_FIFO_MODE_8_BIT
ram_addr	Pointer to the base location of shared RAM; must start on a 4 Kb boundary.

Argument	Description
ram_size	Amount of shared RAM available, which depends on the mode selected: ■ ENI_64K_RAM ■ ENI_32K_RAM ■ ENI_16K_RAM ■ ENI_8K_RAM ■ ENI_4K_RAM FIFO mode is currently restricted to 8 Kb of shared memory.
trigger_event_rtn	Routine called by the ENI driver when an interrupt is triggered by an external device.
The following parameters are used only in FIFO mode:	
rx_queue_size	Maximum number of messages in receive queue (1-127).
tx_queue_size	Maximum number of messages in transmit queue (1-127).
data_size	Ssize of data transaction: ■ ENI_8_BIT ■ ENI_16_BIT ■ ENI_32_BIT
transfer_size	Burst transfer size: ■ ENI_NO_BURST ■ ENI_8_BYTE ■ ENI_16_BYTE
tx_event_rtn	Callback routine used by the ENI driver to release a message back to the application after it has been transmitted. This routine is required.
rx_event_rtn	Optonal callback routine; the application is signaled immediately by the ENI driver when incoming data is received. Specifying NULL prevents the incoming data from being processed immediately (in the ISR). The application must poll for receive data by calling eniReadFIFO.

Return values

ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_CHANNEL_UNAVAILABLE
ENI_INVALID_RAM_SIZE
ENI_IEEE_1284_CONFIGURED
ENI_TRANSMIT_UNDEFINED
ENI_INVALID_RAM_ADDRESS
ENI_INVALID_QUEUE_SIZE
ENI_INVALID_DATA_SIZE
ENI_INVALID_TRANSFER_SIZE
ENI_INVALID_MODE

eniCloseChannel

Closes the ENI channel.

The channel cannot be closed if the ENI driver is still holding buffers owned by the application. The FIFO channel must be flushed before it is permitted to close.

Format

```
int eniCloseChannel (long channel_id);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
ENI_CHANNEL_UNFLUSHED
ENI_CHANNEL_UNREAD
```

eniEnableFIFO

Enables the FIFO channel. Data activity in the channel resumes where it was left off before it was halted.

Format

```
int eniEnableFIFO (long channel_id, long direction);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
direction	Direction of the FIFO channel: ■ ENI_RECEIVE_CHANNEL ■ ENI_TRANSMIT_CHANNEL

Return values

ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI _CHANNEL_UNOPENED
ENI_INVALID_STATE
ENI_FIFO_UNCONFIGURED

eniDisableFIFO

Disables the FIFO channel. Data movement in the channel is halted until the channel is enabled again.

Format

```
int eniDisableFIFO (long channel_id, long direction);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
direction	Direction of the FIFO channel: ■ ENI_RECEIVE_CHANNEL ■ ENI_TRANSMIT_CHANNEL

Return values

ENI_SUCCESS
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
ENI_FIFO_UNCONFIGURED
ENI_INVALID_STATE

eniSupplyFIFO

Supplies receive messages to the ENI driver.

Each message contains a buffer that stores incoming data from the external device. The address of the buffer must be aligned on a 32-bit boundary. The buffer size must be a multiple of 4 bytes. This restriction enables the receive FIFO to work efficiently.

The maximum number of messages that contain allocated buffers depends upon the size of the receive queue configured.

The application replenishes the receive buffers for the ENI driver. The maximum number of buffers that can be queued is determined by the value defined for *rx_queue_size-1* in `eniOpenChannel`. A chained message is not allowed.

Format

```
int eniSupplyFIFO (long channel_id, eniMessageType *message);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
message	Pointer to the allocated empty messages.

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
ENI_CHANNEL_BUSY
ENI_INVALID_STATE
ENI_FIFO_UNCONFIGURED
ENI_INVALID_MESSAGE
```

eniReadFIFO

Retrieves data received from the FIFO channel.

If no message is pending, the application can specify the amount of time to poll for incoming data. Data can be read from the FIFO channel if it is disabled. This routine cannot be used to retrieve messages if the receive callback is defined.

Format

```
int eniReadFIFO (long channel_id, eniMessageType **message,
        long wait_time);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
message	Pointer to a pointer to the message that contains incoming data in buffer received from FIFO.
wait_time	Amount of time to wait for incoming data until time expires. The value of wait time is defined in seconds. A zero (0) indicates no waiting, and –1 is treated as forever.

Return values

ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
ENI_CHANNEL_EMPTY
ENI_FIFO_UNCONFIGURED
ENI_INVALID_MESSAGE
ENI_INVALID_STATE

eniWriteFIFO

Transmits a message through the ENI FIFO.

Each message must contain a buffer, which is 32-bit aligned. The buffer size must be a multiple of 4 bytes. The number of messages the FIFO handles depends on the transmit queue size configured.

The ENI driver owns the message until it is transmitted. After the message is transmitted, the ENI driver returns ownership of the buffer to the application by executing the `tx_event_rtn` routine registered during `eniOpenChannel` call.

Data cannot be written to the FIFO if the channel is disabled. The maximum number of buffers that can be queued is determined by the value defined by *`tx_queue_size`* in `eniOpenChannel`. A chained message is treated as multiple messages. The transmit queue must have enough space for each message in the chain; otherwise, the message is rejected.

Format

```
int eniWriteFIFO (long channel_id, eniMessageType *message);
```

Arguments

Argument	Description
`channel_id`	ID associated with the ENI channel.
`message`	Pointer to a pointer to the message that contains outgoing data.

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
ENI_INVALID_STATE
ENI_CHANNEL_BUSY
ENI_FIFO_UNCONFIGURED
ENI_MESSAGE_OVERFLOW
ENI_INVALID_MESSAGE
```

eniFlushFIFO

Fushes any buffers owned by the application that are still held by the ENI driver.

The FIFO channel must be disabled before it can be flushed. Messages pending in the transmit channel are returned back to the application by the `eniTransmitRtn` callback. The allocated buffers supplied to the receive channels are returned automatically by using the `eniReceiveRtn` callback if defined. Otherwise, the application must manually retrieve the buffers by calling `eniReadFIFO`. Each message that has been aborted contains the `ENI_DATA_ABORT` in the message's status field.

Format

```
int eniFlushFIFO (long channel_id, long direction);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
direction	Direction of the FIFO channel: ■ ENI_RECEIVE_CHANNEL ■ ENI_TRANSMIT_CHANNEL

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
ENI_INVALID_STATE
ENI_CHANNEL_BUSY
ENI_FIFO_UNCONFIGURED
```

eniTriggerInterrupt

Triggers an ENI interrupt to the external device. The interrupt provides a way to synchronize information or communication between two processes running on separate processors.

Format

```
int eniTriggerInterrupt (long channel_id);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_INVALID_CHANNEL
ENI_CHANNEL_UNOPENED
```

eniEnableTrigger

Enables the ENI interrupt, which allows an external device to trigger an interrupt to the NetSilicon chip.

When an interrupt event occurs, the trigger callback routine defined during `eniLoadDriver` is called by the ENI driver. If no callback routine is defined, this routine is inactive.

Format

```
int eniEnableTrigger (long channel_id);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_CHANNEL_UNOPENED
ENI_INVALID_CHANNEL
ENI_TRIGGER_UNDEFINED
```

eniDisableTrigger

Disables the ENI interrupt, which prevents an external device from triggering an interrupt to the NetSilicon chip.

The trigger callback routine registered during `eniLoadDriver` is not called. If trigger callback routine is not defined, this routine is inactive.

Format

```
int eniDisableTrigger (long channel_id);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.

Return values

```
ENI_SUCCESS
ENI_SYSTEM_ERROR
ENI_CHANNEL_UNOPENED
ENI_INVALID_CHANNEL
ENI_TRIGGER_UNDEFINED
```

eniGetSharedRAM

Returns the location and size of the ENI shared memory.

Format

```
int eniGetSharedRAM (long channel_id, void *ram_address,
   long *ram_size);
```

Arguments

Argument	Description
channel_id	ID associated with the ENI channel.
ram_address	Pointer to the memory location of the ENI shared RAM.
ram_size	Pointer to the size of the ENI shared RAM.

Return values

ENI_SUCCESS
ENI_CHANNEL_UNOPENED
ENI_INVALID_CHANNEL

LED Control API

CHAPTER 5

Overview

The LED control API allows applications to turn the LEDs on and off and to blink the LEDs.

Calling restrictions

Some of the LED control API functions are intended to be called from application tasks only.

The blink functions call routines that are not allowed to be called from drivers.

The on/off functions are driver-safe.

Memory usage

Calling the LED API functions results in no additional memory usage.

Include file

Using the LED control API requires the following header file:

```
narmled.h
```

Summary of LED control API functions

Function	Description
NALedRedOn	Turns the red LED on.
NALedRedOff	Turns the red LED off.
NALedGreenOn	Turns the green LED on.
NALedGreenOff	Turns the green LED off.
NALedBlinkRed	Blinks the red LED for a specified number of times and speed.
NALedBlinkGreen	Blinks the green LED for a specified number of times and speed.

LED control API functions

The following pages describe the LED control API functions.

NALedRedOn

Turns the red LED on.

Format

```
void NALedRedOn (void);
```

Arguments

none

Return values

none

NALedRedOff

Turns the red LED off.

Format

```
void NALedRedOff (void);
```

Arguments

none

Return values

none

NALedGreenOn

Turns the green LED on.

Format

```
void NALedGreenOn (void);
```

Arguments

none

Return values

none

NALedGreenOff

Turns the green LED off.

Format

```
void NALedGreenOff (void);
```

Arguments

none

Return values

none

NALedBlinkRed

Blinks the red LED for the specified number of times and the specified speed.

Format

```
void NALedBlinkRed (unsigned long red_blinks, int speed);
```

Arguments

Argument	Description
red_blinks	Number of blinks to perform.
speed	0 — for fast blinks (every .2 seconds) 1 — for slow blinks (every .7 seconds)

Return values

none

NALedBlinkGreen

Blinks the green LED for the specified number of times and the specified speed.

Format

```
void NALedBlinkRed (unsigned long green_blinks, int speed);
```

Arguments

Argument	Description
green_blinks	Number of blinks to perform.
speed	0 — for fast blinks (every .2 seconds) 1 — for slow blinks (every .7 seconds)

Return values

none

Non-Volatile RAM API

CHAPTER 6

Overview

The NVRAM API provides a set of functions that allow an application to store and retrieve data from non-volatile memory.

Addressing

The `nabrdinit` function (in the `narmbrd.c` file) initializes chip select 3 (`CS3`) to place NVRAM at address `0x03000000` with a range of 8192 bytes.

Tasks

Only single tasks should read and write to NVRAM. These functions are not re-entrant and should be called by one task at a time. You must implement a higher level of function to make these functions re-entrant.

Memory usage

No additional memory is allocated when you use the NVRAM API.

Using flash as NVRAM

To use flash as NVRAM, define `BSP_USE_FLASH_FOR_NVRAM` in the the `bspconf.h` file. Then rebuild `bsp.bld`.

This lets you use the first part of the last sector of flash as NVRAM. The amount of flash used depends on the `BSP_NVRAM_SIZE` constant (also defined in `bspconf.h`). Therefore, make sure `BSP_NVRAM_SIZE` is not larger than than the last sector of flash.

Also, `NAFlashInit` must be called before `NANVInit`.

Include file

Using the NVRAM API requires the following header file:

```
narmnvrm.h
```

Summary of NVRAM API functions

API	Description
`NANVInit`	Initializes some global variables.
`NANVWrite`	Writes a buffer to NVRAM.
`NANVRead`	Reads data from NVRAM.
`NANVmemset`	Initializes NVRAM with a specified character.

NVRAM API functions

The following pages describe the NVRAM API functions.

NANVInit

Initializes some global variables.

If you are using flash as NVRAM, the `NANVInit` paramenters are ignored.

Format

```
void NANVInit (char *basep, unsigned long range);
```

Arguments

Arguments	Description
basep	Points to the start address of NVRAM.
range	Size of NVRAM. The address range is from *basep* to (*basep* + *range* - 1).

Return values

none

NANVWrite

Writes a buffer to NVRAM. Simple bounds checks are made for the destination and length.

This function can cause the calling task to sleep while verifying that the data has been successfully written to NVRAM.

Format

```
int NANVWrite (unsigned long offset, char *buffer,
        unsigned long length);
```

Arguments

Arguments	Description
offset	Starting offset from the base of NVRAM.
buffer	Pointer to the buffer containing data to be written.
length	Number of bytes data to be written.

Return values

Return value	Description
0	Data successfully written
-1	A bounds error occurred

NANVRead

Reads data from NVRAM. Simple bounds checks are made before reading the data.

Format

```
int NANVRead (unsigned long offset, char *buffer,
        unsigned long maxlength);
```

Arguments

Arguments	Description
offset	Starting offset from the base of NVRAM.
buffer	Pointer to the destination buffer.
maxlength	Number of bytes of data to be read.

Return values

Value	Description
0 or greater	Total number of bytes read
-1	Invalid arguments were supplied

NANVmemset

Initializes NVRAM with the specified character.

Format

```
void NANVmemset (char data);
```

Arguments

Arguments	Description
data	Character used to initialize NVRAM.

Return values

none

Flash Memory API

CHAPTER 7

Overview

The flash driver API lets you read and write flash memory on the NetSilicon development board. Flash cannot be accessed while it is being written. Therefore, the flash APIs must be loaded into and run from RAM. The functions program flash mapped only to chip select 0 (CS0).

To configure the development board to 32-bit mode flash, move jumpers P27, P28, and P29 from pin 1-2 position to pin 2-3 position. Make sure CS00 of SW3 is off, and CS01 of SW4 is on.

Include file

Using the flash memory API requires the following header files:

- naflash.h
- flash.h

Summary of flash memory API functions

Function	Description
NAFlashBase	Retrieves the flash memory base address.
NAFlashCreateSemaphores	Creates the semaphores used to synchronize access to the flash driver API.
NAFlashEnable	Enables flash memory so it can be read.
NAFlashErase	Erases the flash sectors in the specifed range.
NAFlashEraseStatus	Checks the erase status of the flash sectors in the specified range.
NAFlashInit	Initializes the flash driver.
NAFlashRead	Reads data from flash memory.
NAFlashSectorOffsets	Retrieves the flash sector offsets from the flash base address.
NAFlashSectors	Returns the number of physical sectors for the flash memory parts.
NAFlashSectorSizes	Retrieves the physical sector sizes for flash memory.
NAFlashWrite	Writes data to flash memory.
NAflash_write †	Reprograms logical sectors of flash memory.
NAprogram_flash †	Reprograms flash memory for 16- or 32-bit mode.

† = Deprecated function

Flash memory API functions

The following pages describe the flash memory API functions.

NAFlashBase

Retrieves the flash memory base address.

Format

```
unsigned short * NAFlashBase();
```

Arguments

none

Return values

Return value	Description
unsigned integer	Pointer to the base address of flash memory.

NAFlashCreateSemaphores

Creates the semaphores used to synchronize access to the flash memory API. This function is called by the user application once after powerup.

The following functions can be semaphore protected by calling `NAFlashCreateSemaphores` from the root application thread:

- `NAFlashErase`
- `NAFlashEraseStatus`
- `NAFlashRead`
- `NAFlashWrite`

Format

```
int NAFlashCreateSemaphores();
```

Arguments

none

Return values

Return value	Description
`NAFLASH_SUCCESS`	Successfully created flash driver semaphores
`NAFLASH_SEMAPHORE_CREATE_FAILED`	Flash driver semaphores could not be created
`NAFLASH_DUPLICATE_CALL`	Flash driver semaphores have already been created

NAFlashEnable

Enables flash memory so it can be read.

Format

```
void NAFlashEnable();
```

Arguments

none

Return values

none

NAFlashErase

Erases the flash sectors in the specifed range.

Format

```
int NAFlashErase (unsigned long firstSectorNumber,
        unsigned long lastSectorNumber);
```

Arguments

Argument	Description
firstSectorNumber	0-index-based first sector number to erase.
lastSectorNumber	0-index-based last sector number to erase.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully erased flash sectors.
NAFLASH_UNKNOWN_FLASH	Unknown flash part
NAFLASH_SECTORNUMBER_INVALID	One or more sector numbers invalid
NAFLASH_ERASE_FAILED	Erase operation failed

NAFlashEraseStatus

Checks the erase status of the flash sectors in the specified range.

A flash sector is either erased (all 0xFFs) or not erased. If the sectors are erased, the variable or array element pointed to by *status* is 0. If the sectors are not erased, the variable or array element pointed to by *status* is 1.

For example: Your board has a flash part with 19 sectors, indexed from 0 to 18. To check the status of a range of sectors from 3 to 7, you call:

```
NAFlashEraseStatus(3, 7, status)
```

where *status* is an array of 5 elements (that is, `char status[5]`) and the erase status of the first sector in the range is stored in `status[0]`.

If you use a `char` array of 19 elements (that is, `char status[19]`) to store the erase status of the sectors and want to check the status of sectors 3 to 7, you can call:

```
NAFlashEraseStatus(3, 7, &status[3])
```

The erase status of the first sector is stored in `status[`*firstSectorNumber*`]`.

Format

```
int NAFlashEraseStatus (unsigned long firstSectorNumber,
        unsigned long lastSectorNumber, char *status);
```

Arguments

Argument	Description
firstSectorNumber	0-index-based first sector number to check.
lastSectorNumber	0-index-based last sector number to check.
status	Pointer to a variable or array to store the erase status of one or more sectors.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully checked the erase status
NAFLASH_UNKNOWN_FLASH	Unknown flash part
NAFLASH_SECTORNUMBER_INVALID	One or more sector numbers invalid

NAFlashInit

Initializes the flash driver by performing the following:

- Enables flash memory so it can be read
- Sets the number of flash memory banks

This function is called by the board startup or initialization code.

Format

```
int NAFlashInit (unsigned long flashBanks);
```

Arguments

Argument	Description
flashBanks	Number of flash memory banks.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully initialized flash driver
NAFLASH_FLASHBANKS_INVALID	Invalid number of flash memory banks specified
NAFLASH_DUPLICATE_CALL	Flash driver has already been initialized

NAFlashRead

Reads data from flash memory, starting from a specified sector number and offset location, and stores the data in a buffer. This allows reading one or more consecutive sectors (including the entire flash memory), partials sectors, or a combination.

Format

```
int NAFlashRead(unsigned long sectorNumber,
        unsigned long sectorOffset,
        unsigned long bytesToRead, char *buffer);
```

Arguments

Argument	Description
sectorNumber	0-index-based flash sector to read from.
sectorOffset	0-index-based flash sector offset.
bytesToRead	Number of bytes to read.
buffer	Pointer to the buffer to store the data. The buffer must be large enough to store the bytes requested.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully read the data
NAFLASH_UNKNOWN_FLASH	Unknown flash part
NAFLASH_SECTORNUMBER_INVALID	One or more sector numbers invalid
NAFLASH_SECTOROFFSET_INVALID	Invalid sector offset
NAFLASH_MEMORY_OUTOFBOUND	Read operation exceeed flash memory boundaries

NAFlashSectors

Returns the number of physical sectors for the flash memory parts.

The development board allows configuring flash memory in 16-bit or 32-bit mode. In either mode, the number of physical sectors for flash memory is identical (regardless of the number of flash chips on the board).

If the flash memory consists of multiple banks, the number of sectors returned is a multiple of the physical sectors for the flash memory part. For example, if a flash memory part has 19 sectors and two flash banks are used, `NAFlashSectors` returns 38.

Format

```
int NAFlashSectors();
```

Arguments

none

Return values

Return value	Description
integer	Number of physical sectors in flash memory
NAFLASH_UNKNOWN_FLASH	Unknown flash part

NAFlashSectorOffsets

Retrieves the flash sector offsets from the flash base address in bytes.

This function must be called after `NAFlashSectors` so an appropriately sized array is passed to the function to store the data.

Format

```
int NAFlashSectorOffsets (unsigned long *sectorOffsetArray);
```

Arguments

Argument	Description
sectorOffsetArray	Pointer to an array to store the sector offsets in bytes.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully retrieved flash sector offsets
NAFLASH_UNKNOWN_FLASH	Unknown flash part

NAFlashSectorSizes

Retrieves the physical sector sizes for the flash memory configuration. This function must be called after `NAFlashSectors` so an appropriately sized array is passed to the function to store the sector size data.

The development board allows configuring flash memory in 16-bit or 32-bit mode. In 16-bit mode, the sector sizes are the size of one flash part. In 32-bit mode, the sector sizes are twice the size of one flash part (because two flash parts are accessed per I/O cycle).

Format

```
int NAFlashSectorSizes (unsigned long *sectorSizeArray);
```

Arguments

Argument	Description
sectorSizeArray	Pointer to an array to store the sector sizes for flash memory.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully retrieved flash sector sizes
NAFLASH_UNKNOWN_FLASH	Unknown flash part

NAFlashWrite

Writes a data buffer to flash memory, starting from a specified sector number and offset location. This allows writing one or more consecutive sectors (including the entire flash memory), partials sectors, or a combination.

Format

```
int NAFlashWrite (unsigned long sectorNumber,
        unsigned long sectorOffset,
        unsigned long bytesToWrite, char *buffer,
        char options);
```

Arguments

Argument	Description
sectorNumber	0-index-based flash sector to write.
sectorOffset	0-index-based flash sector offset to write.
bytesToRead	Number of bytes to write.
buffer	Pointer to the buffer with the new data.
options	One of the following: ■ ERASE_AS_NEEDED — For a full sector write, erases the sector (if necessary) and writes the new data ■ For a partial sector write, updates the new data but does not destroy the rest of the sector data ■ ALWAYS_ERASE — For a full sector write, erases the sector before writing new data ■ For a partial sector write, updates the new data and erases the rest of the sector ■ DO_NOT_ERASE — Writes new data without first erasing the sector If the write operation changes a bit from 0 to 1, the function returns NAFLASH_WRITE_FAILED.

Return values

Return value	Description
NAFLASH_SUCCESS	Successfully wrote the new data
NAFLASH_UNKNOWN_FLASH	Unknown flash part
NAFLASH_WRITE_FAILED	Write operation failed
NAFLASH_VERIFY_FAILED	Verify operation failed
NAFLASH_ERASE_FAILED	Erase operation failed
NAFLASH_SECTOROFFSET_INVALID	Invalid sector offset
NAFLASH_MEMORY_OUTOFBOUND	Write operation exceeded flash memory boundaries
NAFLASH_WRITEOPTION_INVALID	Invalid write option specified

NAflash_write

Reprograms logical sectors of flash memory.

Note: Deprecated function. See `NAFlashWrite` and other flash API functions.

Format

```
int NAflash_write (unsigned long sectorNumber,
   unsigned short *sectorDatap)
```

Arguments

Argument	Description
sectorNumber	Specifies which 32Kb logical sectors to program: ■ 16-bit mode: from 0–31 ■ 32-bit mode: 0–63
sectorDatap	Pointer to the new flash data.

Return values

Return value	Description
0	Successfully programmed flash
-1	Unknown flash part
-2	Erase failed
-3	Write failed
-4	Verify failed
-5	Invalid sector number
-6	Memory out of bounds

NAprogram_flash

Reprograms flash memory with the data specified.

Note: Deprecated function. See `NAFlashWrite` and other flash API functions.

Format

```
int NAprogram_flash (char *flashdata)
```

Arguments

Argument	Description
flashdata	Pointer to the data representing the new contents of flash memory.

Return values

Return value	Description
0	Successfully programmed flash
-1	Unknown flash part
-2	Erase failed
-3	Write failed
-4	Verify failed
-5	Invalid sector number
-6	Memory out of bounds

Serial Number API

CHAPTER 8

Overview

The serial number API lets you:

- Convert the development board serial number into an Etherntet address
- Read the serial number from memory
- Write the serial number into memory

Include file

Using the serial number API requires the following header file:

```
narmsrln.h
```

Summary of serial number API functions

Function	Description
NASerialnum_to_mac	Converts a 6-digit serial number represented as a string of alphanumeric digits into a 6-byte Ethernet address.
NAResolveSerial	Reads the six-digit serial number from NVRAM and stores it in a character array as a NULL-terminated string.
NASaveSerial	Writes a six-digit serial number into NVRAM.

Serial Number API functions

The following pages describe the serial number API functions.

NASerialnum_to_mac

Converts a 6-digit serial number represented as a string of alphanumeric digits into a 6-byte Ethernet address from the range of MAC addresses allocated by the IEEE.

Format

```
int NASerialnum_to_mac (char *srlnp, char *mac_addrp);
```

Arguments

Argument	Description
srlnp	Pointer to the six-digit serial number.
mac	Pointer to the buffer to be loaded with the MAC address.

NAResolveSerial

Reads the six-digit serial number from NVRAM and stores it in a character array as a NULL-terminated string.

Format

```
void NAResolveSerial (char *srln);
```

Arguments

Argument	Description
srln	Pointer to the buffer for the six-digit serial number.

Return values

none

NASaveSerial

Writes a six-digit serial number into NVRAM.

Format

```
void NASaveSerial (char *srln);
```

Arguments

Argument	Description
srln	Pointer to the buffer for the six-digit serial number.

Return values

none

Cache API

CHAPTER 9

Overview

The cache API lets you control the cache inside the NetSilicon chip. Cache is not available on all NetSilicon chips. See the NetSilicon hardware documentation for the chip you are using.

What cache does

Cache provides a small amount of RAM inside the chip. Because this RAM is inside the chip, accesses to it are much faster than accesses to external ROM or RAM.

The cache controller maintains a copy of some sections of external ROM or RAM in cache RAM. When the NET+ARM processor attempts to access external ROM or RAM that is cached, the cache controller directs the access to the copy stored in cache RAM.

Cache sets and cache lines

Cache RAM is divided into *cache sets*, which are divided into *cache lines*.

Each *cache set* can have 256, 1024, or 2048 cache lines, depending on the chip version.

Each *cache line* corresponds to a word in external memory and holds up to 4 bytes of data. Each line in a cache set stores both the data word being cached and the data's address in external memory.

How cache is used

When the processor accesses a word in external memory, the cache controller examines the address of the word being accessed. The upper bits of the address are examined to determine whether the word is in a section of memory that is being cached. If the section of memory is being cached, the lower bits of the address are used to determine which cache line in a cache set the word corresponds to. Then the entire address is compared against the address stored in the address field of the cache line to determine if the word is already in cache. If more than one cache set is assigned to the region of memory in question, the operation is done in parallel for all the cache sets in question.

For example, the NET+40 chip has four cache sets, each with 256 cache lines. This means that each cache set can hold up to 1024 bytes, for a total of 4096 bytes of cached RAM. The NET+40 cache controller uses bits 24-31 of the address to determine whether an address is within a section of memory being cached, and bits 2-9 are used to determine which cache line is used in each cache set.

Suppose the NET+40 cache controller was configured to use all four cache sets to cache memory from `0x08000000-0x0bffffff`, and the processor attempts to read from address `0x08001234`. The cache controller would then determine that the word in question:

- Is in the range being cached
- Corresponds to line `0x8d` in the four cache sets
- Is was already in cache

If a read or instruction fetch is being performed, and the data word is already in cache, the cache controller delivers the word to the processor without accessing external memory. If a write operation is being performed, cache RAM is updated with the new data. The cache controller can be configured to either update memory immediately during a write operation or delay the update of external RAM until later. This may be desirable if the data is likely to change soon.

"Dirty" cache

When external RAM is not updated, the cache line is marked as *dirty* to indicate that it contains data that has not yet been written to external RAM. Whenever the cache controller reuses a cache line to cache a word in a different location in memory, the current contents are written to external RAM if the cache line is marked as dirty.

Cache controller round-robin algortihm

When more than one cache set is used to cache a region of memory, the cache controller uses a round-robin algorithm to determine which cache set should be used when the cache lines in all the cache sets are already in use.

For example, suppose cache sets 1 through 4 are used to cache memory from `0x08000000–0x0bffffff`, and that the processor reads from addresses `0x8010004`, `0x8020004`, `0x8010004`, `0x8040004`, `0x8000004`, `0x8010004`, and `0x8050004`. Because bits 0-9 are the same for all the addresses, they all reference the same cache line.

The following table lists the actions the cache controller takes on each read:

Memory access	Results
0x8010004	Read word from that address in external memory and store in cache set 1.
0x8020004	Read word from that address in external memoryand store in cache set 2.
0x8010004	Fetch word stored in cache set 1.
0x8040004	Read word from that address in external memory and store in cache set 3.
0x8000004	Read word from that address in external memory and store in cache set 4.
0x8010004	Fetch word stored in cache set 1.
0x8050004	Read word from at that address in external memoryand store in cache set 1.

When the seventh read is done, all the cache lines in the four cache sets are already in use. So the algorithm uses the cache line in cache set 1, even though it was accessed more frequently and more recently than the other cache sets.

Cache control registers

Cache is configured through one or more *cache control registers* (CCRs). Each CCR sets up a region of memory that is cached and controls which cache sets are used for that region of memory.

The CCR also controls the operational mode for the cache region. The cache API allows applications to program CCRs through the `NACacheEnable` function. A region can be set up for instruction cache (read-only), or data cache (read/write).

Restrictions

Be aware of these restrictions:

- The NetSilicon chip caches accesses by the NET+ARM processor only. Accesses by other devices in the chip, such as the DMA controller, are not cached.
- The chip does not cache access to memory made by devices external to the chip.
- Consult the NetSilicon hardware documentation for restrictions on cache pertaining to the chip you are using.
- ICE debuggers cannot be used when cache is enabled.

Memory map and recommended usage

NetSilicon recommends that you use the default memory map set up by the BSP. This memory map locates:

- RAM in the 32 Mb region from address `0x00000000` to `0x01ffffff`
- ROM in the 2 Mb region from `0x02000000` to `0x021fffff`
- NVRAM in the region from `0x03000000` to `0x03001fff`

ROM is limited to 2 Mb, and NVRAM is limited to 8Kb. The because that is how much is present on the development board. These address ranges can be expanded on boards with more physical memory.

When the memory map is set up, the chip select registers are programmed so that the memory map repeats every 32 Mb. So, for example, the same word of RAM appears at addresses `0x00000000`, `0x04000000`, `0x08000000`, and 0x0c000000. The BSP then programs the cache controller to enable instruction cache in the range from `0x08000000` to `0x0bffffff`, and data cache (if supported) from `0x04000000` to `0x07ffffff`.

Address range	Memory type	Cache mode
0x0e000000 – 0x0e1fffff	ROM	Not cached
0x0c000000 – 0x0dffffff	RAM	Not cached
0x0a000000 – 0x0a1fffff	ROM	Instruction cache
0x08000000 – 0xbfffffff	RAM	Instruction cache
0x06000000 – 0x061fffff	ROM	Data cache
0x04000000 – 0x05ffffff	RAM	Data cache
0x02000000 – 0x021fffff	ROM	Not cached
0x00000000 – 0x01ffffff	RAM	Not cached

All versions of the NetSilicon chip allow memory to be duplicated as described. So, the same memory map works with NetSilicon chips that do not support cache. Using the standard memory map allows you to write applications that take advantage of cache when it is present (on a NET+40 chip, for example), but that also run on chips that do not support cache (a NET+15 chip, for example).

The read/write data region is located in non-cache memory. This means that applications built with the BSP defaults will be linked to use instruction cache, but not data cache.

When you use data cache, make sure memory that will be used for DMA transfers is not cached. NET+OS does not provide APIs to allocate non-cached memory, and there is no way to directly control where memory allocated for network buffers is allocated from. Because the Ethernet driver uses DMA, applications must be linked to locate the read/write data region in non-cache memory.

To cache a data buffer, application code should allocate the buffer from non-cached memory as normal. Then adjust the pointer to it to refer to the buffer in the cached address space.

For example, if the standard memory map is used, then a buffer allocated at 0x00200000 is in non-cache memory. It can also be referenced at 0x04200000. References to it at the higher address will be cached.

The application must obey these rules for using cache:

- Cached memory must never be accessed by DMA or any other external device. Only the NET+ARM core processor (CPU) can access memory that is being cached.
- Buffers must not be referenced by the application in both the cached and non-cached data space.

  For example, if the application writes to address `0x04200000` and then tries to read from address `0x00200000`, it may not necessarily read the same value it wrote.
- Thread stacks are good candidates for data cache.

The BSP automatically enables cache on processors that support it during initialization. This is done by the `InitBoard` function (in the `board.c` file) by calling `NACacheSetDefaults`. Cache is set up by `NACacheSetDefaults` to use the standard memory map.

Startup code

The chip select registers are programmed to set up the memory map by the startup code located in the `init.s` file. The instruction and data cache address ranges are controlled by constants defined in the `nacache.h` file.

Cache data types

The cache API uses two structures that are defined in the `nacache.h` file:

- `naCacheInfoType`
- `naCacheDescriptorType`

naCacheInfoType structure

The naCacheInfoType structure is used by NAIdentifyCache to return information about the type of cache supported.

```
typedef struct
{
    unsigned long addressBits;
    int addressMaskShift;
    int numberOfCCRs;
    int cacheSets;
    int numberOfCacheSets;
    unsigned setSize;
    unsigned flags;
    long unsigned cacheRamAddress;
    unsigned long size;
int debugSupport;
} naCacheInfoType;
```

This table describes the fields:

Field	Description
addressBits	A bit field that indicates which address bits are used by the cache controller to determine whether or not an address is in a section of memory that has been cached. Each bit set in this field indicates that the corresponding address bit is used.
addressMaskShift	Used internally.
numberOfCCRs	Determines the number of cache control registers on the chip.
cacheSets	Bit values for cache sets to use when calling NACacheEnable.
numberOfCacheSets	Number of available cache sets.
setSize	Reports the size of a cache set in bytes.

Field	Description
flags	Reports the type of cache operations supported. Possible values are: ■ INSTRUCTION_CACHE_SUPPORTED, DATA_CACHE_SUPPORTED ■ (INSTRUCTION_CACHE_SUPPORTED | DATA_CACHE_SUPPORTED)
cacheRamAddress	Indicates the address of cache RAM. This RAM is available for read/write use if cache is disabled.
size	Reports the size of cache RAM in bytes.
debugSupport	A bit field that indicates the types of debuggers that can be used when cache is turned on. Possible values are: ■ ICE_DEBUGGER_SUPPORTED, SW_DEBUGGER_SUPPORTED ■ (ICE_DEBUGGER_SUPPORTED | SW_DEBUGGER_SUPPORTED)

naCacheDescriptorType structure

The naCacheDescriptorType structures represent the CCR registers on the chip. These structures tell the NACacheEnable function how the CCR registers should be set up.

For example:

```
typedef struct
{
    unsigned long addressBase;
    unsigned long addressMask;
    unsigned flags;
    unsigned cacheSets;
} naCacheDescriptorType;
```

This table describes the fields:

Field	Description
addressBase	Indicates where a cached region of memory should start.
addressMask	Indicates the bits the cache controller should examine when determining whether an address is cached.
flags	Indicates the operational mode for the cache region: ■ ENABLE_CACHE — Enables cache region ■ DONT_CACHE_USER — User mode accesses not cached ■ COPY_BACK_MODE — Causes writes to cache to be immediately written to RAM ■ WRITE_PROTECT — Causes writes to cause data abort ■ FORCE_32BIT_PREFETCHES — External memory is known to be 32-bit
cacheSets	Selects the cache sets for the region.

Include file

Using the cache API requires the following header file:

```
nacache.h
```

Summary of cache API functions

Function	Description
NACacheIdentify	Returns a pointer to a structure containing information about the type of cache is supported by the chip.
NACacheEnable	Configures the cache controller on the chip and enables cache.
NACacheDisable	Disables cache and flushes it.
NACacheSetDefaults	Configures the cache to default values.
NACacheRestore	Re-enables cache. Cache is restored to the settings set by the last call to NACacheEnable.

Cache API functions

The following pages describe the cache API functions.

NACacheIdentify

Returns a pointer to a structure containing information about the type of cache is supported by the chip.

This structure should be treated as read-only by the application.

Format

```
naCacheInfoType *NACacheIdentify (void);
```

Arguments

none

Return values

Pointer to a read-only `naCacheInfoType` structure that describes the system's cache.

NACacheEnable

Configures the cache controller on the chip and enables cache.

The function is passed to an array of `naCacheDescriptorType` structures. Each structure contains the configuration settings for one CCR on the chip. There must be one structure for each CCR.

ICE debuggers cannot be used when cache is enabled. By convention, the valid bit in the base address register for chip select 0 (`CS0`) is set to 0 (zero) when an ICE is in use. This function tests this bit and exits without enabling cache if it is set to 0

The function first disables and flushes cache. It then configures cache according to the values passed to it and enables it.

Format

```
int NACacheEnable (naCacheDescriptorType *descriptor,
         int numberDescriptors);
```

Arguments

Argument	Description
descriptor	Pointer to an array of one or more `naCacheDescriptorType` structures.
numberDescriptors	Number of `naCacheDescriptorType` structures in the array pointed to by *descriptor*.

Return values

SUCCESS
WRONG_NUMBER_OF_DESCRIPTORS
INVALID_DESCRIPTOR
CACHE_NOT_SUPPORTED
ICE_DETECTED
SW_DEBUGGER_DETECTED

NACacheDisable

Disables and flushes cache.

Format

```
void NACacheDisable (void);
```

Arguments

none

Return values

none

NACacheSetDefaults

Configures the cache to default values.

This function is called automatically during startup from the `InitBoard` function in the `board.c` file.

Format

```
int NACacheSetDefaults (void);
```

Arguments

none

Return values

```
SUCCESS
CACHE_NOT_SUPPORTED
ICE_DETECTED
```

NACacheRestore

Re-enables cache after `NACacheDisable` has been called.

Cache is restored to the settings set by the last call to `NACacheEnable`.

Format

```
int NACacheRestore (void);
```

Arguments

none

Return values

`SUCCESS`

Interrupt Service Routines (ISR) API

CHAPTER 10

Overview

The interrupt service routines (ISR) API allows applications to install and remove ISRs.

There is no additional memory usage when you call the ISR API.

Writing ISRs under NET + OS

The NetSilicon chip supports 32 interrupt levels. The levels are assigned by hardware to specific devices internal to the chip, and to pins on the chip. When a device signals an interrupt, the hardware sets bits in the interrupt controller registers to indicate which device is signaling the event.

If the device's interrupt level is not masked off, the hardware generates an IRQ exception. This causes the NET+OS interrupt driver to be executed. The interrupt driver determines which device is signaling the interrupt condition and calls the ISR that is registered to it. The ISR processes the interrupt and then returns. At this point, the interrupt driver checks for more pending interrupts. If any are found, their ISRs are called as well. When all pending interrupts have been processed, the NET+OS interrupt driver returns control to the application.

Developing ISRs for NET+OS is straightforward; ISRs are coded as C routines and should be defined like this:

```
int applicationIsr(void *isrParameter)
```

Application ISRs should always return `0x0`.

The *`isrParameter`* parameter is set to the value passed to `NAInstallIsr` when the ISR was installed.

ISRs are installed by calling `NAInstallIsr` which takes a function pointer, interrupt level, and pointer as parameters. It copies the function pointer into the interrupt driver's table of ISRs and then enables interrupts for the specified interrupt level. Once this is done, the ISR will be called whenever the interrupt occurs. The pointer passed to `NAInstallIsr` also is stored in an internal table and passed to the ISR by the driver when it is invoked. The parameter can be used to pass context information when the same ISR is used to service multiple interrupts.

ISRs are uninstalled by calling `NAUninstallIsr`. This function accepts an interrupt level as a parameter. It disables interrupts from the device and removes the function pointer from the ISR table.

Use the `tx_interrupt_control` kernel function to enable and disable all interrupts on the chip. The functions `NAEnableIsr` and `NADisableIsr` can be used to selectively disable interrupts from a single device.

When an ISR is called, it should service the device generating the interrupt and return as quickly as possible. The ISR does not need to do anything with the chip interrupt controller because this is taken care of by the NET+OS interrupt driver. However, it does need to acknowledge the interrupt on the device so the device stops generating the interrupt. This is normally done by either reading a register on the device or setting a particular bit in a register on the device.

Most RTOS APIs are not available to ISRs. The *NET+OS Kernel User's Guide* has a list of which kernel services can be called safely from an ISR. ISRs must not call any kernel API not in that list, or any other function that uses a kernel function that is not in that list.

Include file

Using the ISR API requires the following header file:

```
na_isr.h
```

Summary of ISR API functions

Function	Description
NADisableIsr	Disables a specific interrupt bit.
NAEnableIsr	Enables a specific interrupt bit.
NAInstallIsr	Installs an application ISR.
NAUninstallIsr	Uninstalls (removes) an application ISR.

ISR API functions

The following pages describe the ISR API functions.

NADisableIsr

Disables interrupts from a specific device.

The corresponding bit in the chip interrupt enable register (IER) is reset to 0. This causes interrupts from the associated device to be ignored.

Format

```
int NADisableIsr (int bitnum);
```

Arguments

Argument	Description
bitnum	Interrupt bit to mask (0–31).

Return values

SUCCESS
NA_INVALID_LEVEL

NAEnableIsr

Enables interrupts from a specific device.

The corresponding bit in the chip interrupt enable register (IER) is set. This causes interrupts from the associated device to be acted upon.

Format

```
int NAEnableIsr (int bitnum);
```

Arguments

Argument	Description
bitnum	Interrupt bit to 3mask (0–31).

Return values

```
SUCCESS
NA_INVALID_LEVEL
```

NAInstallIsr

Installs an application ISR for the specified interrupt bit and enables interrupts from the device. Because this function enables interrupt from the device, the ISR must be completely set up and ready to process interrupts before `NAInstallIsr` is called.

Format

```
int NAInstallIsr (int bitnum, NA_ISR_HANDLER isrHandler,
         void *isrParameter);
```

Arguments

Argument	Description
bitnum	Interrupt bit of the ISR to install.
IsrHandler	Pointer to application ISR.
IsrParameter	Parameter to be passed to ISR.

Return values

SUCCESS
NA_INTERRUPT_IN_USE
NA_INVALID_LEVEL

NAUninstallIsr

Uninstalls (removes) an application ISR for the specified interrupt bit, and disables interrupts from the device.

Format

```
int NAUninstallIsr (int bitnum);
```

Arguments

Argument	Description
bitnum	Interrupt bit of the ISR to remove

Return values

```
SUCCESS
NA_INVALID_LEVEL
```

System Clock and Timer Support API

CHAPTER 11

Overview

The NET+Works system clock is based on a choice between an external crystal and an external oscillator. You make your selection with the external PLLTST* signal. Whatever source is selected, its frequency greatly affects the timing.

To assist in porting to new crystals or oscillators, the BSP includes a system clock API. This API consists of four compiler definitions and two functions which are used throughout the BSP. All new designs must review and update the definitions for proper operation of the system timers and baud rate generators.

Include files

Using the system clock API requires the following files:

- `sysClock.c` and `sysClock.h`
- `bsptimer.c` and `bspconf.h`

Memory usage

No additional memory is allocated when using the system clock and timer support routines.

Summary of system clock and timer support functions

API	Description
`NAgetSysClkFreq`	Returns the value of the internal system clock signal (`SYSCLK`)
`NAgetXtalFreq`	Returns the value of the internal auxiliary clock signal (`XTAL`), which is the primary input to the serial bit rate generator

System clock and timer API functions

The following pages describe the system clock and timer API functions.

NAgetSysClkClkFreq

Returns the value of the internal system clock signal (SYSCLK).

The SYSCLK value depends on on the compiler directives and PLLCNT value from the PLL control register:

- If PLLTST_SELECT is set to SELECT_THE_CRYSTAL_OSCILLATOR_INPUT, the frequency of the system clock is based on this formula:

  ```
  SYSCLK frequency = (FCrystOsc/5) * (PLLCNT <3 ? 6: (PLLCNT +3))
  ```

  where *FCrystOsc* is the crystal oscillator frequency, and *PLLCNT* is the value in the PLL control register.

- If PLLTST_SELECT is set to SELECT_THE_XTAL1_INPOUT_INPUT, the frequency of the system clock is XTAL1_FREQUENCY.

Format

```
unsigned int NAgetSysClkFreq (void)
```

Arguments

none

Return values

Frequency of the SYSCLK signal.

NAgetXtalClkFreq

Returns the value of the internal auxiliary clock signal (XTAL).

XTAL is the primary input to the serial bit-rate generator. The XTAL frequency is based on the system clock (SYSCLK) according to this formula:

```
XTAL frequency = (Fsysclk / (PLLCNT < 3 ? 6 : (PLLCNT + 3))
```

where *Fsysclk* is the value of SYSCLK (see the description of the NAgetSysClkGFreq function).

Format

```
unsigned int NAgetXtalClkFreq (void)
```

Arguments

none

Return values

Frequency (in Hz) of the XTAL signal.

System clock and timer compiler directives

This table summarizes the system clock and timer compiler directives. A detailed description of each follows the table.

Compiler directive	Description
PLLTST_SELECT	Determines the source to be used for the system clock.
XTAL1_FREQUENCY	Indicates the frequency of the TTL clock input applied to the XTAL1 pin.
CRYSTAL_OSCILLATOR_FREQUENCY	Indicates the frequency of the crystal oscillator.
PLL_CONTROL_REGISTER_N_VALUE	Indicates the *n* factor used in the divide-by circuits of the chip clock generator.

PLLTST_SELECT

Determines the clock source to be used. This is the address input to the SYSCLK signal multiplexer.

The SYSCLK has two possible sources:

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

PLLTST_SELECT should be set to either of these:

- SELECT_THE_XTAL1_INPUT
- SELECT_THE_CRYSTAL_OSCILLATOR_INPUT

In the current implementation, this directive is defined as of the BSP initialization directory.

This function uses the usesInternalOscillator data member of external XTAL or crystal oscillator.

XTAL1_FREQUENCY

Indicates the frequency of the TTL clock input applied to the XTAL1 pin.

If PLLTST_SELECT is set to SELECT_THE_XTAL1_INPUT, this signal generates the internal SYSCLK signal.

Otherwise, the value is ignored.

CRYSTAL_OSCILLATOR_FREQUENCY

Indicates the frequency of the crystal oscillator.

If PLLTST_SELECT is set as SELECT_THE_CRYSTAL_OSCILLATOR_INPUT, this used to generate an input to the phase lock loop (PLL) and along with the value in the PLL control register.

Otherwise, the value is ignored.

PLL_CONTROL_REGISTER_N_VALUE

This compiler directive indicates the *n* factor used in the divide-by circuits of the chip clock generation section.

This value is stored in the `PLLCNT` field within the PLL control register. This factor multiples or divides clock sources as described in the NetSilicon hardware documentation for the chip you are using.

Legal values are 0 to 15; suggested values are based on the chip type and revision.

The current implementation defines this directive as `getPLLValueBasedOnChipType`, which is in the `settings.c` file. This function returns the `PLLCount` data member of the `NetarmInitData` table, based on chip type, to determine the desired clock speed.

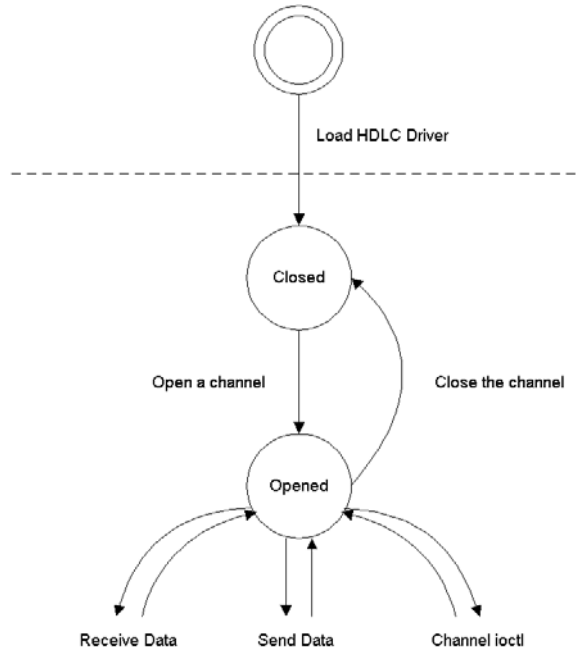
HDLC Driver API

CHAPTER 12

Overview

The high-level data link control (HDLC) driver API provides access to and control over the HDLC hardware on the NetSilicon chip.

This is the HDLC driver state diagram:



Initializing the HDLC driver

To enable the HDLC driver, define `APP_ENABLE_HDLC` in `appconf.h`.

The BSP calls `naHdlcLoad` to initialize the driver. This function allocates memory for frame buffers shared by all HDLC channels.

The driver uses three types of frames:

- **Large.** Should be big enough to hold the largest frame the application needs to receive, up to 32,768 bytes.
- **Small.**
- **Empty.** Empty frames do not have a buffer allocated with the frame; they transmit out of the user buffer.

If the specified large or small frame size is not a multiple of 4 bytes, the nearest larger multiple of 4 bytes is used instead.

The total size of allocated memory is:

```
(NAHDLC_LARGE_FRAME_SIZE * NAHDLC_NUM_LARGE_FRAMES) +
(NAHDLC_SMALL_FRAME_SIZE * NAHDLC_NUM_SMALL_FRAMES) +
40 * (NAHDLC_NUM_LARGE_FRAMES + NAHDLC_NUM_SMALL_FRAMES +
NAHDLC_NUM_EMPTY_FRAMES)
```

Opening the HDLC channel

Before starting to send and receive data, the user application should call `naHdlcOpen` to open an HDLC channel. This function:

- Allocates memory for an HDLC channel, including receive DMA buffers of large frame size each
- Initializes DMA buffer descriptors
- Enables DMA channels
- Initializes the serial channel controller
- Installs receive and transmit interrupts

The total size of allocated memory for every channel is under the value of:

```
NAHDLC_LARGE_FRAME_SIZE * HDLC_RX_DESCRIPTORS + 64
(HDLC_RX_DESCRIPTORS + HDLC_TX_DESCRIPTORS) + 300 BYTES
```

The `HDLC_RX_DESCRIPTORS` and `HDLC_TX_DESCRIPTORS` values are BSP configurable, defined in `hdlcdvr.h`.

Closing the HDLC channel

Call `naHdlcClose` to close the HDLC channel. This function disables receive and transmit interrupts, disables DMA channels, disables the serial channel, frees all HDLC frames held by the driver for this channel, and frees memory allocated for this channel.

HDLC frame structure

This structure defines the HDLC frame:

```
typedef struct naHdlcFrame

  char    *header;/*header pointer*/
  unsigned inthlen;/* header length*/

  char    *data; /*data pointer*/
  unsigned intdlen;/* data length*/

  unsigned longflags;/* frame flags*/

  naHdlcFrame;
```

Two pointers — *`header`* and *`data`* — are provided to eliminate the need to copy transmit frame data after the header into the continuous memory space. The terms *header* and *data* do not specifically mean HDLC header and an information field; you can break a frame in any meaningful way. The *`header`* and *`data`* pointers can, but are not required to, point somewhere in the frame buffer space.

In the valid HDLC frame, one of these situations exists:

- *`data`* points to frame data,
  `dlen` is equal to frame length,
  `header` = `NULL`,
  `hlen` = 0
- *`header`* points to frame data,
  `hlen` is equal to frame length,
  `data` = `NULL`,
  `dlen` = 0

- *header* points to frame header,
  hlen is equal to header length,
  data points to frame information field,
  dlen is information field length.

Beginning and ending frame flag sequences are never placed in a frame buffer, and the CRC is not placed in the frame buffer by default. The driver can be configured to place the CRC in a received frame buffer by means of the NAHDLC_RX_CRC.

The flags field passes additional information between an application and the driver as shown here:

1. The driver sets the NAHDLC_FLG_LOCKED flag to indicate its ownership of the frame.
2. The driver sets the NAHDLC_FLG_SENDER flag, if a transmit error occurred.
3. An application can set the NAHDLC_FLG_KEEP flag if the driver should not free a transmitted frame.

Allocating and freeing HDLC frames

To allocate a frame, call `pframe = naHdlcFrameAlloc(size)`.

The *size* argument can be any number less than or equal to the large frame size. Size 0 allocates empty frames. The actual frame buffer size is equal to the large frame size, the small frame size, or 0.

An allocated frame has these attributes:

- *header* = NULL
- *hlen* = 0
- *flags* = 0
- *data* points to the frame buffer
- *dlen* is equal to the frame buffer size

To free the frame, call `naHdlcFrameFree(pframe)`.

Sending HDLC frames

To send a frame, call naHdlcFrameSend:

1. Allocate a frame.
2. Copy data to the frame buffer, if needed.
3. Set `pframe` → `header` and `pframe` → `hlen` and/or `pframe` → `data` and `pframe` → `dlen`.
4. To queue the frame to send, call:

```
naHdlcFrameSend(channel, pframe, urgent, release)
```

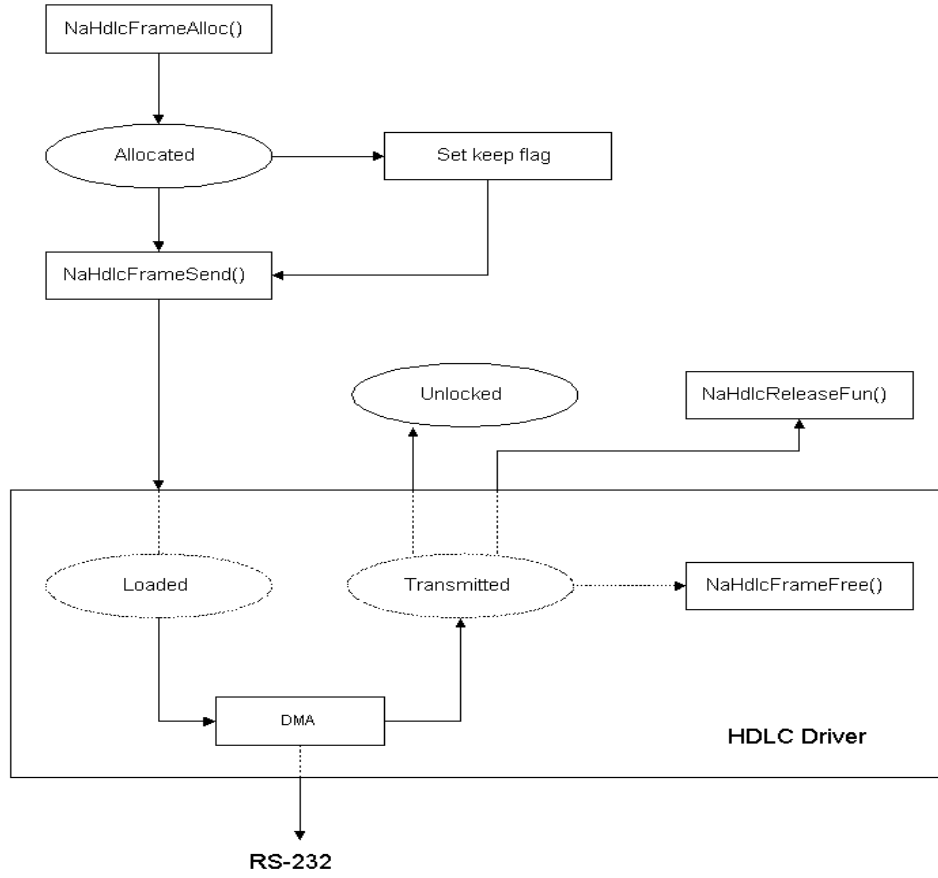
The user application controls whether the transmitted frame is released to the application or freed by the driver.

The user application can define a `naHdlcReleaseFun` callback to process transmitted frames. If `naHdlcFrameSend` is called with a non-zero release argument, the driver calls `release`from the transmit ISR. This callback should not wait in any system resources. The only HDLC API function it can call is `naHdlcFrameFree`.

The other way to affect the driver's frame release mechanism explicitly is to set the `NAHDLC_FLG_KEEP` frame flag. In this case, an application should poll the frame's *flags* field until the driver clears the `NAHDLC_FLG_LOCKD` flag. If the release callback is not used and the `NAHDLC_FLG_KEEP` flag is not set, the driver frees the transmitted frame.

To avoid a situation in which all frames are tied up in the driver's send queue, you can set the maximum send queue length with `NAHDLC_MAX_SENDQ` .

This diagram illustrates the frame send process:



Receiving HDLC frames

The receive DMA channel is configured to interrupt the CPU when an HDLC frame is received.

The `naHdlcAcceptFun` and `naHdlcReceivedFun` callback routines register two callbacks that are called from the receive ISR.

`naHdlcAcceptFun` defines a function that filters received frames. This callback returns `TRUE` to accept a frame and `FALSE` to reject it. The driver discards the rejected frame without allocating a frame structure. This callback also can be used to signal the user application to read received frames.

`naHdlcReceivedFun` defines a function that processes a frame completely in the receive ISR. This callback transfers ownership of the frame to the user application. If the `naHdlcReceivedFun callback` is not registered, the driver queues received frames.

The `naHdlcAccept Fun` and `naHdlcReceivedFun` callback routines must not wait on any system resources. The only HDLC API function they can call is `naHdlcFrameFree`.

The HDLC driver processes received frames in this way:

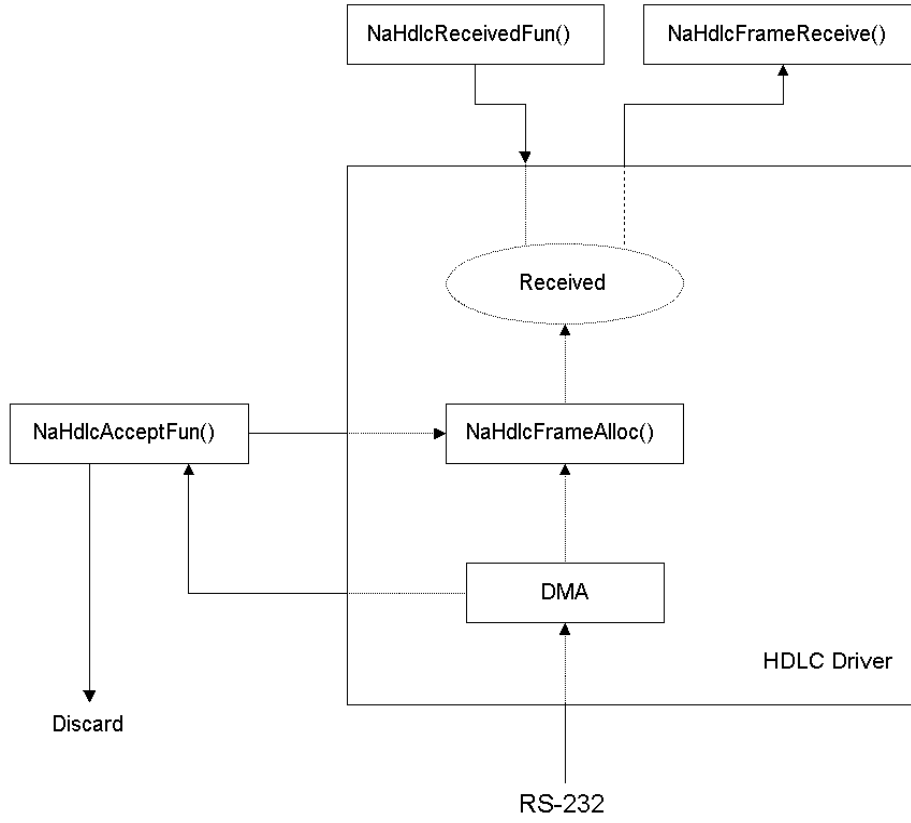
1. The driver calls `naHdlcAcceptFun` if this callback routine has been registered. If the callback returns `FALSE`, the driver discards the frame.
2. The driver allocates a frame structure of the size to fit the received frame. If the driver cannot allocate the frame structure, it discards the frame.
3. If a small frame has been allocated, the driver copies received frame data to the buffer of the allocated frame. If a large frame has been allocated, the driver swaps the DMA buffer with the allocated frame buffer.
4. The driver sets *`data`* to point to the received frame buffer and *`dlen`* to the actual frame size; *`header`*, *`hlen`*, and *`flags`* are equal to 0.
5. The driver calls `naHdlcReceivedFun` if this callback routine has been registered. Otherwise, the driver queues the received frame.

To retrieve a frame queued by the driver, the user application should call:

```
naHdlcFrame *naHdlcFrameRecv(int channel);
```

An application should free all received frames by calling `naHdlcFrameFree`.

This figure illustrates the frame receive process:



Include file

Using the HDLC driver API requires the following header file:

```
hdlcapi.h
```

Summary of HDLC driver ioctl calls

Name	Description	Argument	Default
NAHDLC_FRAME_LEN	Set maximum frame length	Maximum receive frame length	Large frame size
NAHDLC_MAX_SENDQ	Set maximum size send queue size	Maximum frames queues to send	50
NAHDLC_LL	Enable local loopback	TRUE - enable FALSE - disable	FALSE
NAHDLC_HW_HANDSHAKING	Enable hardware CTS/ RTS handshaking	TRUE – enable FALSE – disable	FALSE
NAHDLC_RX_CLOCK_INT	Set receive clock source internal	TRUE – internal FALSE –external	FALSE
NAHDLC_TX_CLOCK_EXT	Set transmit clock source external	TRUE – external FALSE – internal	FALSE
NAHDLC_RX_CLOCK_INVERT	Invert receive clock	TRUE – invert FALSE – normal	FALSE
NAHDLC_TX_CLOCK_INVERT	Invert transmit clock	TRUE – invert FALSE – normal	FALSE
NAHDLC_BAUD	Set baud rate	Baud rate	128000
NAHDLC_NUM_FLAGS	Number of additional flags between frames	0 –15	0
NAHDLC_CRC_32	Set 32-bit CRC	TRUE – 32-bit FALSE – 16-bit	16-bit CRC
NAHDLC_CRC_DISABLED	Disable CRC	TRUE – disable FALSE – enable	FALSE
NAHDLC_RX_CRC	Place receive CRC in the buffer	TRUE, FALSE	FALSE
NAHDLC_RCV_BAD_CRC	Accept receive frames with bad CRC	TRUE, FALSE	FALSE
NAHDLC_NOTX_CRC	Do not send CRC on transmit	FALSE – send CRC TRUE – no CRC	FALSE

Name	Description	Argument	Default
NAHDLC_IDLE_FLAGS	Send idle between frames	TRUE - idle (FF) FALSE - flags (7E)	FALSE
NAHDLC_ADDR_16	16 bits address match	TRUE – 16-bit FALSE – 8-bit	8-bit
NAHDLC_MATCH_1	Match address 1	Address to match 0 – don't match	0
NAHDLC_MATCH_2	Match address 2	Address to match 0 – don't match	0
NAHDLC_MATCH_3	Match address 3	Address to match 0 – don't match	0
NAHDLC_MATCH_4	Match address 4	Address to match 0 – don't match	0
NAHDLC_MASK_1	Mask address 1	Address mask	0
NAHDLC_MASK_2	Mask address 2	Address mask	0
NAHDLC_MASK_3	Mask address 3	Address mask	0
NAHDLC_MASK_4	Mask address 4	Address mask	0
NAHDLC_ACCEPT_FUN	Register the accept callback	Pointer to a naHdlcAcceptFun callback routine	NULL
NAHDLC_RECEIVED_FUN	Register the received callback	Pointer to a naHdlcReceivedFun callback routine	NULL
NAHDLC_SET_RTS	Activate RTS	TRUE / FALSE	
NAHDLC_SET_DTR	Activate DTR	TRUE / FALSE	
NAHDLC_GET_CTS	Get CTS	Filled in with 1 or 0 on return	
NAHDLC_GET_DSR	Get DSR	Filled in with 1 or 0 on return	
NAHDLC_GET_DCD	Get DCD	Filled in with 1 or 0 on return	

Descriptions of HDLC driver ioctl calls

Setting maximum frame length for received frames

NAHDLC_FRAME_LEN changes the maximum received frame size. The frame size accounts for the CRC only if the driver is configured to place the received CRC in the frame buffer. The maximum frame size value possible is large frame size.

Frames larger than the large frame size can be transmitted out of the user buffer.

Setting maximum send queue

NAHDLC_MAX_SENDQ changes the maximum number of queued transmit frames.

Setting hardware CTS/RTS handshaking

NAHDLC_HW_HANDSHAKING with *arg* = TRUE enables support for CTS and RTS handshaking. If hardware handshaking is enabled and CTS goes down, the currently transmitted frame is aborted.

The driver sets the DSR and RTS signals active whenever it disables or enables hardware flow control.

Setting local loopback

NAHDLC_LL with *arg* = TRUE enables local loopback mode, directly connecting the serial channel to its transmitter. When local loopback is enabled, the HDLC driver uses an internal clock source for both receive and transmit. When NAHDLC_LL is called to disable the local loopback mode, the driver returns to the clock settings used before local loopback mode was enabled.

Receive and transmit clock configuration

NAHDLC_TX_CLOCK_EXT configures the clock source for transmit data. If *arg* is 0, the chip provides the clock for the data it transmits (terminal transmit clock — TTC); otherwise, the chip uses the DCE's clock (transmit clock — TC).

Because DTE always uses the DCE's clock for the data it receives, the NAHDLC_RX_CLOCK_INT normally should not be used. Some tests, however, may require the chip to provide the clock for the data it receives.

Transmit and receive clocks can be inverted with NAHDLC_TX_CLOCK_INVERT and NAHDLC_RX_CLOCK_INVERT.

Baud rate

NAHDLC_BAUD sets the baud rate for the internal clock source. The baud rate can be set to any number that is:

FXTAL / 2 /2048 ≤ baud ≤ FSYSCLK / 10

FXTAL is the XTAL output frequency and FSYSCLK is the system clock frequency. The driver uses SYSCLK output if FSYSCLK / 2 baud ≤ 2048; otherwise, it uses output.

If the chip cannot generate the exact baud rate, it generates the nearest possible value to the one requested.

CRC settings

A user application can use one of these ioctl calls to change CRC settings:

Use	To
NAHDLC_CRC_32	Use a 32-bit CRC instead of a 16-bit CRC (only with the NET+50 and later chips).
NHDLC_CRC_DISABLED	Disable the CRC completely (for example, for protocols that resend received frames to speed up the process by eliminating the CRC calculations).
NAHDLC_RX_CRC	Place the CRC of a received frame in a buffer.
NAHDLC_RCV_BAD_CRC	Accept received frames with bad CRC.
NAHDLC_NOTX_CRC	Disable sending the CRC with transmit frames.

Changing the number of flags between frames

NAHDLC_NUM_FLAGS configures the number of additional flags inserted between transmit frames, from 0 to 15.

Sending idle/flags between frames

NAHDLC_IDLE_FLAGS controls whether the driver sends idle (0xFF) or flags (0x7E) when there are no frames to send.

Configuring address match

When address match is enabled, the HDLC controller receives only those frames that have one of the preset match addresses. A user application can set up to four 8-bit addresses or two 16-bit addresses to match. Some bits can be excluded from the address comparison by setting them to 1 in the corresponding address mask.

NAHDLC_ADDR_16 controls whether the HDLC controller uses 16- or 8-bit match:

- To specify an 8-bit match address, use NAHDLC_MATCH_1, NAHDLC_MATCH_2, NAHDLC_MATCH_3, or NAHDLC_MATCH_4 with an 8-bit address value.

  Use NAHDLC_MASK_1, NAHDLC_MASK_2, NAHDLC_MASK_3, or NAHDLC_MASK_4 to specify a mask for the corresponding address.
- To specify a 16-bit match address, use NAHDLC_MATCH_1 or NAHDLC_MATCH_3 with a 16-bit address value.

  Use NAHDLC_MASK_1 or NAHDLC_MASK_3 to specify a mask for the corresponding address.
- To clear the address match, use NAHDLC_MATCH_*n* with 0 argument.

Registering callbacks to accept frames and receive frames

NAHDLC_ACCEPT_FUN registers the naHdlcAcceptFun callback.

NAHDLC_RECEIVE_FUN registers the naHdlcReceivedFun callback.

Both ioctl calls remove the corresponding callback when called with the NULL argument.

Getting and setting the line status

NAHDLC_SET_RTS and NAHDLC_SET_DTR set RTS / DTR active with *arg* = TRUE, inactive with *arg* = FALSE.

NAHDLC_GET_CTS, NAHDLC_GET_DSR, and NAHDLC_GET_DCD fill in the integer variable pointed to by *arg* with 1 if CTS / DSR / DCD is active; otherwise, the integer variable is 0.

Summary of HDLC driver API functions

Function	Description
naHdlcLoad	Allocates frames shared by all HDLC channels.
naHdlcOpen	Opens an HDLC channel.
naHdlcClose	Closes an HDLC channel.
naHdlcFrameAlloc	Allocates an HDLC frame.
naHdlcFrameFree	Frees the HDLC frame.
naHdlcFrameSend	Sends an HDLC frame.
naHdlcFrameRecv	Receives an HDLC frame.
naHdlcIoctl	Performs an HDLC channel ioclt.
naHdlcAcceptFun	Filters frames out.
naHdlcReceivedFun	Transfers ownership of the frame to the user application. Used to process frames completely in the receive ISR.
naHdlcReleaseFun	User routine called from the transmit ISR to return the frame to the calling application.
naHdlcGetStats	Returns channel statistics structure.

HDLC driver API functions

The following pages describe the HDLC driver API functions.

naHdlcLoad

Allocates frames shared by all HDLC channels.

Format

```
int naHdlcLoad (int large_frame_size, int small_frame_size,
        int num_large_frames, int num_small_frames,
        int empty_frames);
```

Arguments

Argument	Description
large_frame_size	Size of the large frame buffer: > 0 and < 32768.
small_frame_size	Size of the small frame buffer: greater than *large_frame_size*.
num_large_frames	Number of large frames.
num_small_frames	Number of small frames.
num_empty_frames	Number of empty frames.

Return values

Return value	Description
NAHDLC_SUCCESS	Successfully allocated the frames
NAHDLC_ERR_INVAL	Bad arguments
NAHDLC_ERR_EXISTS	Frames already allocated
NAHDLC_ERR_NOMEM	Not enough memory

naHdlcOpen

Opens an HDLC channel.

Format

```
int naHdlcOpen (int channel);
```

Arguments

Argument	Description
channel	Channel number.

Return values

Return value	Description
NAHDLC_SUCCESS	Successfully opened the channel
NAHDLC_ERR_INVAL	Bad channel number
NAHDLC_ERR_NODE	Channel not initialized
NAHDLC_ERR_EXISTS	Channel already open

naHdlcClose

Closes an HDLC channel.

Format

```
int naHdlcClose (int channel);
```

Arguments

Argument	Description
channel	Channel number.

Return values

Return value	Description
NAHDLC_SUCCESS	Successfully closed the channel
NAHDLC_ERR_INVAL	Bad channel number
NAHDLC_ERR_NODEV	Channel not open

naHdlcFrameAlloc

Allocates an HDLC frame.

Format

```
naHdlcFrame *naHdlcFrameAlloc (int size);
```

Arguments

Argument	Description
size	Frame size.

Return values

Return value	Description
	Pointer to an allocated frame
NULL	Could not allocate a frame

naHdlcFrameFree

Frees the HDLC frame.

Format

```
void naHdlcFrameFree (naHdlcFrame *pframe);
```

Arguments

Argument	Description
pframe	Pointer to the frame to free.

Return values

none

naHdlcFrameSend

Sends an HDLC frame.

Format

```
int naHdlcFrameSend (int channel, naHdlcFrame *pframe,
  int urgent, naHdlcSentFun send_done);
```

Arguments

Argument	Description
channel	Channel number.
pframe	Pointer to the frame to send.
urgent	One of the following: ■ TRUE — urgent delivery ■ FALSE — normal delivery

Return values

Return value	Description
NAHDLC_SUCCESS	Successfully sent the frame
NAHDLC_ERR_INVAL	Bad channel number or null frame pointer
NAHDLC_ERR_NODEV	Bhannel not open
NAHDLC_ERR_FRAMEERR	Frame not valid
NAHDLC_ERR_AGAIN	Channel send queue is full

naHdlcFrameRecv

Receives an HDLC frame.

Format

```
naHdlcFrame *naHdlcFrameRecv (int channel);
```

Arguments

Argument	Description
channel	Channel number.

Return values

Return value	Description
NULL	No frame received

naHdlcIoctl

Performs an HDLC channel `ioctl` call.

Format

```
int naHdlcIoctl (int channel, int request, void *arg);
```

Argument

Arguments	Description
channel	Channel number.
request	`ioctl` request.
arg	`ioctl` argument.

Return values

Return value	Description
NAHDLC_SUCCESS	Success
NAHDLC_ERR_INVAL	Bad channel number or null configuration pointer
NAHDLC_ERR_NODEV	Channel not open
NAHDLC_ERR_SIZE	Maximum frame size is too large
NAHDLC_ERR_NOTSUP	Configuration option is not supported

naHdlcAcceptFun

Callback routine to filter frames out.

This callback, if registered, is called by HDLC driver from the receive ISR. To register this callback, use `naHdlcAcceptFun`.

Format

```
typedef int (* naHdlcAcceptFun)(int channel, char *buffer,
         int len);
```

Argument

Arguments	Description
channel	Channel number.
buffer	Pointer to the received frame buffer.
len	Received frame length.

Return values

Return value	Description
TRUE	Frame accepted
FALSE	Frame rejected

naHdlcReceivedFun

Callback routine to process frames completely in the receive ISR. In this case, the driver does not queue received frames, so there is no need to call `naHdlcFrameRecv`.

This callback, if registered, is called by the HDLC driver from the receive ISR. This call transfers the frame's ownership to the user application. To register this callback routine, use `naHdlcReceivedFun`.

Format

```
typedef void (*naHdlcReceivedFun)(int channel,
        naHdlFrame *pframe);
```

Arguments

Argument	Description
channel	Channel number.
pframe	Pointer to the received frame.

Return values

none

naHdlcReleaseFun

If this routine is passed as an argument in `naHdlcFrameSend`, it is called from the transmit ISR. It returns the transmitted frame back to the user application.

Format

```
void (*naHdlcReleaseFun) (int channel, naHdlcFrame *pframe);
```

Arguments

Argument	Description
channel	Channel number.
pframe	Pointer to the transmitted frame.

Return values

none

naHdlcGetStats

Returns a pointer to channel statistics structure.

Format

```
naHdlcStatistics *naHdlcGetStats (int channel);
```

Arguments

Argument	Description
channel	Channel number.

Return values

Return value	Description
	Pointer to channel statistics structure
NULL	Bad channel number or channel not open

DMA Driver API

CHAPTER 13

Overview

The direct memory access (DMA) controller supports 10 channels. (Some NetSilicon chips may not use all 10. Consult the hardware documentation for the chip you are using.) Channels 3 and 4 can be used to connect to the peripherals attached to the BBus. The DMA driver provides functions for controlling the DMA hardware for channels 3 and 4 which also can be used for internal peripherals.

This following table shows the assignment of DMA channels to internal peripherals:

Peripheral	DMA channel	Usage
Ethernet 1	Channel 1	Receive
	Channel 2	Transmit
ENI FIFO 1	Channel 3	Receive unavailable if Parallel 1 is opened or if used for external peripherals
	Channel 4	Transmit unavailable if Parallel 2 is opened or if used for external peripherals
Parallel 1	Channel 3	Receive/transmit unavailable if ENI FIFO is opened or if used for external peripherals
Parallel 2	Channel 4	Receive/transmit unavailable if ENI FIFO is opened or if used for external peripherals
Parallel 3	Channel 5	Receive/transmit unavailable if ENI FIFO is opened
Parallel 4	Channel 6	Receive/transit unavailable if ENI FIFO is opened
Serial 1	Channel 7	Receive
	Channel 8	Transmit
Serial 2	Channel 9	Receive
	Channel 10	Transmit

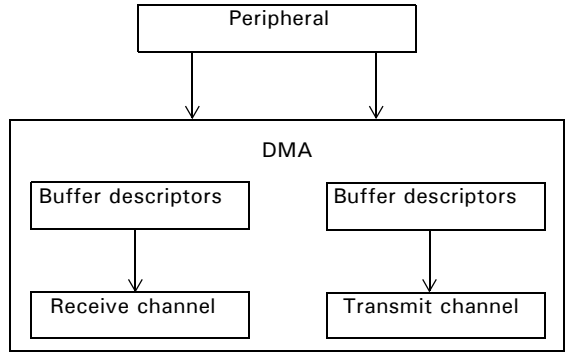
Modes of operation

DMA channels support two modes of operation:

- Fly-by mode — Data is directly transferred between memory and the peripheral.
- Memory-to-memory — Data is not directly transferred but is buffered in between transfers. The data is copied from the source memory location into a temporary area in the DMA channel and then written to the destination memory location.

Buffer descriptors

Data is moved using *buffer descriptors*. Each buffer descriptor requires two 32-bit words for fly-by mode or four 32-bit words for memory-to-memory mode. Each DMA channel can address up to 128 fly-by buffer descriptors or 64 memory-to-memory buffer descriptors.



Each channel contains a *buffer descriptor pointer*, which provides the address of the first byte descriptor. Loading the source address of the data and manipulating the control flags in the descriptor buffer activates the channel.

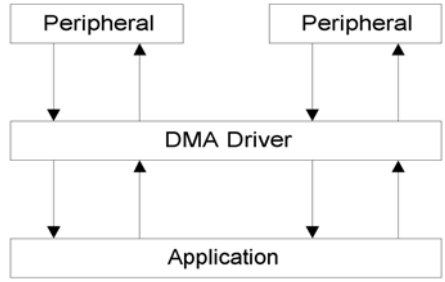
For details about the hardware specifics of the DMA, see the NetSilicon hardware documentation for the chip you are using.

API to support DMA hardware

The DMA driver provides easy access to the DMA hardware without dealing with specific hardware details. For example, the DMA hardware requires configuration and manipulation of the internal registers. Each channel is reserved to direct data to or from a particular peripheral.

Two types of buffer descriptors support data transfer: one requires both the source and destination address, and the other requires only a source address.

Although you would need to deal with some of these issues when programming the DMA hardware directly, the DMA driver API hides much of the detail.



How applications use the DMA driver

The DMA driver API comprises seven functions which can be used by the application layer to interact with the DMA hardware. The DMA driver can be initialized during system startup.

A channel for a peripheral can be opened either to receive incoming data or to transmit outgoing data. When the channel is opened, data movement in the channel can be enabled or disabled. If the channel is enabled, buffers can be loaded into the DMA system for data transmission or data collection. When the data system is finished with the buffers, it is unloaded for additional processing. When the channel is no longer needed, the application can close it.

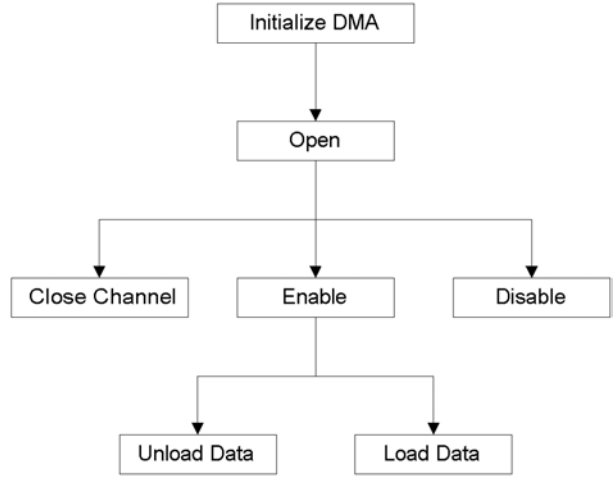


Figure 5: DMA driver API functions

This figure illustrates the DMA state diagram:

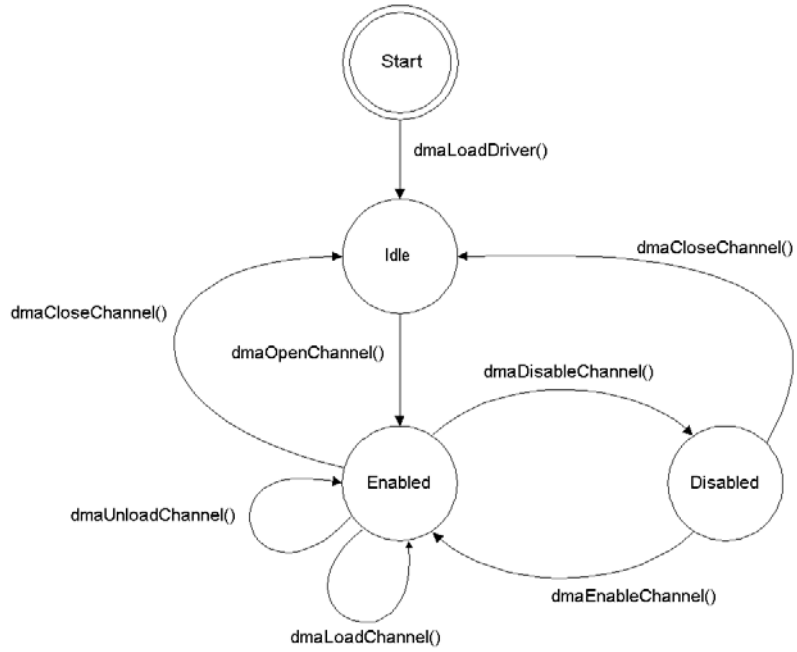


Figure 6: DMA state diagram

Initializing the DMA driver

To start the DMA driver, the application must call `dmaLoadDriver`. This function creates a default state for each DMA channel. This function should be called during system initialization and before any peripherals are used. It needs to be called only once.

Opening a channel

After the DMA driver is initialized, either DMA channel 3 or 4 can be opened. To access a channel for a peripheral, call `dmaOpenChannel` with either of the following channel types: `DMA_FIFO_1_RX` or `DMA_FIFO_1_TX`.

Configuring a channel

After a peripheral has been selected, each channel must be configured. Call `dmaOpenChannel` to pass several configuration parameters:

- Channel type: `DMA_FIFO_1_RX` or `DMA_FIFO_1_TX`
- The data transfer mode: Fly-by or memory-to-memory
- The burst transfer size: 8-bit of data, 16-bit of data, or no burst transfer
- Whether the request is from an internal or external source
- Whether the source or destination address is to be incremented
- The data operand size: 8-bit, 16-bit, or 32-bit
- The number of buffer descriptors to use
- A buffer release callback routine, if needed

After the call to open a channel, the DMA driver allocates the buffer descriptors and configures the registers with the values specified in the configuration. The interrupt handlers are then installed. The registers are set for an interrupt to occur. At this point, the channel is ready to be used.

Closing a channel

When an application no longer needs the DMA channel, it should release the channel by calling `dmaCloseChannel`. If pending requests are in the transmit channel, the channel cannot be closed. Any empty buffers in the receive channel are released back to the application. The interrupt for the channel is disabled. Any memory allocated is freed.

Disabling a channel

The application may need to stop the DMA channel from sending outgoing data through a transmit channel or collecting incoming data from a receive channel. Calling `dmaDisableChannel` immediately stops the activity in the channel, which may cause data in the channel to be lost. At this point, the DMA channel is in an unknown state and does not know which buffer descriptor to process next. All requests are automatically released back to the driver.

Enabling a channel

If a DMA channel has been disabled, it can be enabled again by calling `dmaEnableChannel`. However, activity in the channel does not resume; intead, the DMA driver is reset to start at the first buffer descriptor.

Defining a request

Once a channel to a peripheral has been opened, it is ready to receive or transmit data, depending on the channel selected. Data is passed between the application and DMA driver by using a *request*. A request can contain one or more *messages*. A message contains several fields, inlcuding:

- `src_addr`
- `dst_addr`
- `length`
- `status`
- `error_value`

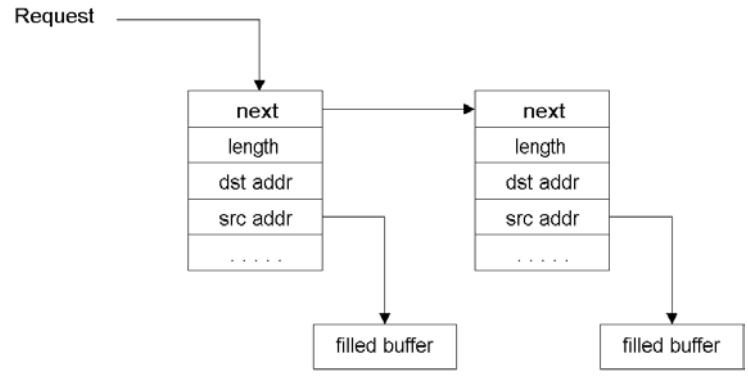
The use of these fields is based on the channel selected:

- **Transmit channel.** The *source address* is the location from which outgoing data is to be read. This address stores the buffers provided by the application that needs to be transmitted.
- The *destination address* contains the location to which data is being written.

  This address is not used in fly-by mode because the channel is attached to the address of an assigned peripheral.

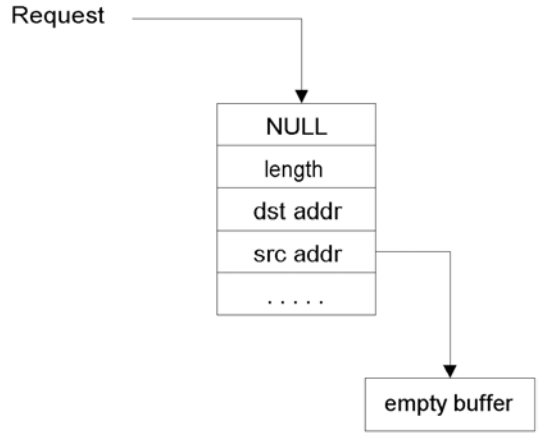
  The size of the buffer to be transmitted is stored in the `length` field.

  The `status` and `error_value` fields contain the result for each message transmitted out the DMA channel. Multiple messages can be chained using the `next` field.



- **Receive channels.** The *source address* (`src_addr`) is the location from which incoming data is to be read. This address is not used in fly-by mode since the channel is attached to the address of an assigned peripheral.

  The *destination address* (`dst_addr`) contains the location where incoming data is to be stored. This address contains the address of an empty buffer provided by the application. The buffers must be on a 32-bit boundary; this restriction is a requirement of the hardware. The size of the empty buffer provided by the application is stored in the `length` field. The `status` and `error_value` fields contain the result for each message received from the DMA channel. Chain messages are not allowed for the received channel.



After a request is submitted, the application no longer owns the message in the request. This allows the DMA driver to avoid performing buffer copies. The application should not modify the buffer until it is released back to the application by the driver.

Sending a request

Before data can be moved to a peripheral, a channel reserved for outgoing data must be opened. The buffers are loaded into the transmit channel by calling `dmaLoadChannel`. These buffers are submitted to the transmit channel as a request message.

A request can be accepted only if space is available in the channel; otherwise, a busy status is returned. This means no request is queued if the channel is busy.

Each request can have one or more buffers set in the `source` address field. In memory-to-memory mode, the `destination` address field must be defined to inform the DMA where to write the data. The application should not modify any of the buffers until the DMA driver releases it. All buffers are released when the transmit channel has delivered the request.

Supplying a request

Before data can be received from a peripheral, a channel reserved for incoming data must be opened. The application must supply empty buffers to the DMA driver to store incoming data. The buffers are loaded into the receive channel by calling `dmaLoadChannel`.

These buffers are submitted to the receive channel as a request message. A request can be accepted only if space is available in the channel; otherwise, a busy status is returned. This means that no request is queued if the channel is busy. It is important for an application to replenish the request into the receive channel in order for the DMA driver to continually collect data. If no request is available when required, some data will be lost.

Each request can have one or more buffers set in the `destination` address field. In memory-to-memory mode, the `source` address field must be defined in order for the DMA to know where to fetch the data. The application should not modify any buffer until it is released. All buffers are released back to the application when the receive channel fills the empty buffer with data collected from the peripheral. At this point, another buffer is required to replace the one just released.

Retrieving a request

After the DMA driver has processed a submitted request, the request is released back to the application. The DMA driver relinquishes a request by executing the release callback routine. Once the application receives the release signal, it can retrieve the request by calling `dmaUnloadChannel`.

The transmit channel uses this function to retrieve any requests that have been transmitted. Once a request is retrieved, it can be reused by the application to send another message. The receive channel uses this unload function to retrieve any request containing incoming data. Once the application has processed the incoming request, it probably should be reused for other incoming data.

Releasing a request

When an application submits a request, no buffers are copied internally in the DMA driver. Instead, the buffers owned by the application are used directly by the DMA channel. At this point, the application should not modify the data in the buffer until the buffer is released. A buffer is released only when the DMA driver is done using it or an error condition occurs. The condition of release can be determined by examining the passed status parameter.

To release a request, the DMA driver generates a signal to the application by invoking the callback release routine. When the application receives the signal, it regains ownership of the request. Once a request is retrieved and processed, it probably should be reused to transport outgoing data or to collect incoming data.

The code implemented in the release callback routine depends heavily on the application. Using semaphores or waking up a thread is one way in which an application can be notified when a release occurs. If other channels share the same callback routine, be sure the routine is re-entrant.

The release callback routine is registered when a channel is opened. This routine is not required if the application does not want an immediate release signal generated by the DMA driver during an ISR and during a `dmaDisable_channel`. The application can retrieve the request from the channel by calling `dmaUnloadChannel`.

When an application gets a release signal from a receive channel, it is an indication that one less buffer is available for the DMA driver to load incoming data. A short supply of buffers can cause data to be lost. An application that expects a high data volume should replenish released buffers quickly.

Include file

Using the DMA driver API requires the following header file:

```
dma_api.h
```

Summary of DMA driver API functions

Function	Description
`dmaReleaseType`	Calls a routine during an ISR or `dmaDisableChannel`.
`dmaLoadDriver`	Loads the DMA driver by initializing data structures.
`dmaOpenChannel`	Retrieves a processed request from the DMA channel.
`dmaCloseChannel`	Closes an open DMA channel.
`dmaDisableChannel`	Stops DMA from receiving data through the inbound channel or transmitting data through the outbound channel.
`dmaEnableChannel`	Allows DMA to receive data through the inbound channel or to transmit data through the outbound channel.
`dmaLoadChannel`	Submits a request to a DMA channel.
`dmaUnloadChannel`	Retrieves a processed request from the DMA channel.

DMA driver API functions

The following pages describe the DMA driver API functions.

dmaReleaseType

Signals the application that the DMA driver has released a request. A request is released only when incoming data is received or outgoing data has been transmitted.

The DMA driver calls this function during an ISR or `dmaDisableChannel`.

Once a request is released, the DMA driver is not allowed to modify it. If other channels share the same callback routine, make sure it is re-entrant.

This callback routine is not required since the application can poll for any released buffer by calling `dmaUnloadChannel`.

You can check the status field in the `request_msg` to know the result returned by the DMA channel when the buffer was processed.

Format

```
void (*dmaReleaseType) (int channel_id,
         dmaMessage Type *request_msg);
```

Arguments

Argument	Description
channel_id	Identifier for the DMA channel.
request_msg	Pointer to the message.

Return values

none

dmaLoadDriver

Loads the DMA driver by initializing data structures. Each channel is initialized to default values.

Format

```
int dmaLoadDriver (void);
```

Arguments

none

Return values

DMA_SUCCESS

dmaOpenChannel

Opens the receive or transmit channel for a peripheral.

Format

```
int dmaOpenChannel (int *channel_id, int channel_type,
        unsigned int option_flags, int ring_size,
        dmaReleaseType release_rtn);
```

Arguments

Argument	Description
channel_id	Returns the identifier to associate with the requested channel. The value is (*channel_type* - 1).
channel_type	Peripheral channel to open either DMA_FIFO_1_RX or DMA_FIFO_1_TX.
option_flags	One or more flags with option settings for the DMA driver. See *"Option flag settings,"* below.
ring_size	Number of buffer descriptors to allocate. The buffer descriptor type depends on the mode selected. Each DMA buffer descriptor requires two 32-bit words for fly-by mode and four 32-bit words for memory-to-memory mode. Each DMA channel can address a maximum number of 128 fly-by buffer descriptors or 64 memory-to-memory buffer descriptors.
release_rtn	Callback routine to signal the application when the DMA driver releases a request.

Option flag settings

- DMA operation mode. Defines DMA operation modes. The receive channel uses fly-by write to direct data from the FIFO to be stored into memory; the transmit channel uses Fly-by read to transfer data from memory out to the FIFO. Defaults for receive and transmit channels are fly-by write and fly-by read, respectively.

  To override the defaults with memory-to-memory, use DMA_MEM_TO_MEM_MODE.

- **Burst transfer size.** These flags specify the burst transfer size. The default is no burst.

  To override the default, use:
  - `DMA_8_BYTE_BURST`
  - `DMA_16_BYTE_BURST`

- **Channel request source.** Specifies the channel request source; only channels 3 and 4 support this field. The default is internal request.

  To override the default value with external request, use `DMA_EXTERNAL_REQ`.

- **Source address increment.** Specifies whether the source address is incremented during data transfer. This flag is not valid for fly-by mode. If the source address is a memory address, this field should be set. Incrementing should not be selected if the source address is a location with a fixed address. The FIFO register is a 32-bit register with a fixed address. By default, the source address is not incremented.

  To increment the source address, use `DMA_SRC_INCR`.

- **Destination address increment.** Specifies whether the destination address is incremented during data transfer. This flag is not valid for fly-by mode. If the destination address is a memory address, this field should be set. Incrementing should not be selected if the destination address is a location with a fixed address. The FIFO register is a 32-bit register with a fixed address. By default, the destination address is not incremented.

  To increment the destination address, use `DMA_DADDR_INC`.

- **DMA transaction data operand size.** These flags specify the size of each DMA transaction data operand used when the external channel request source is defined or with memory-to-memory mode. The default is 32-bit.

  To override the default value with 8-bit or 16-bit data size, use:
  - `DMA_8_BIT`
  - `DMA_16_BIT`

- **Signal lines.** The following flags specify the signal lines to be enabled. The input and output signal lines are used by external DMA channels. This feature is active only if DMA_EXTERNAL_REQ is configured.
  - DMA_DREQ_SIGNAL
  - DMA_DACK_SIGNAL
  - DMA_DONE_IN_SIGNAL
  - DMA_DONE_OUT_SIGNAL

Return values

DMA_SUCCESS
DMA_CHANNEL_INUSE
DMA_SYSTEM_ERROR
DMA_INVALID_RING_SIZE
DMA_INVALID_FLAG
DMA_CALLBACK_UNDEFINED
DMA_INVALID_STATE
DMA_CHANNEL_UNSUPPORTED
DMA_INVALID_CHANNEL

dmaCloseChannel

Closes an open DMA channel.

Format

```
int dmaCloseChannel (int channel_id);
```

Arguments

Argument	Description
channel_id	Identifier for the DMA channel to be closed.

Return values

DMA_SUCCESS
DMA_CHANNEL_UNOPENED
DMA_INVALID_CHANNEL
DMA_LOAD_PENDING
DMA_UNLOAD_PENDING

dmaDisableChannel

Stops the DMA system from receiving data through the inbound channel or transmitting data through the outbound channel.

Disabling the channel causes the DMA system to be in an unknown state, which cause data to be lost. Therefore, before calling this function, make sure no pending requests are in the DMA driver.

Format

```
int dmaDisableChannel (int channel_id);
```

Arguments

Argument	Description
channel_id	Identifier for the DMA channel to be disabled.

Return values

DMA_SUCCESS
DMA_CHANNEL_UNOPENED
DMA_INVALID_CHANNEL
DMA_INVALID_STATE

dmaEnableChannel

Allows the DMA system to receive data through the inbound channel or to transmit data through the outbound channel.

Re-enabling the channel causes the DMA driver to reset the buffer descriptor back to the first buffer, and it is not to resume any transmit or receive actions in DMA.

Format

```
int dmaEnableChannel (int channel_id);
```

Arguments

Argument	Description
channel_id	Identifier for theDMA channel to be enabled.

Return values

```
DMA_SUCCESS
DMA_CHANNEL_UNOPENED
DMA_INVALID_CHANNEL
DMA_INVALID_STATE
```

dmaLoadChannel

Submits a request to a DMA channel:

- If the DMA channel is used for transmitting, the request contains outgoing data.
- If the channel is in receiver mode, the request contains empty buffers for incoming data.

The buffers cannot be used in cache memory. The DMA driver reuses the buffers in the message request to avoid buffer copies. After a request is submitted, the application should not modify any of the buffers until they are released by the DMA driver.

Format

```
int dmaLoadChannel (int channel_id,
         dmaMessageType *request_msg);
```

Arguments

Argument	Description
channel_id	Identifier for the DMA channel that is opened.
request_msg	Pointer to the dmaMessageType structure (see below).

dmaMessageType structure

```
typedef struct dmaMessageType
{
    struct dmaMessageStruct *next;
    void *src_addr;
    void *dst_addr;
    lonG length;
    long status;
    long error_value;
    long reserved[4];
} dmaMessageType
```

This table describes the fields in the dmaMessageType structure:

Field	Description
next	Pointer to a chain of dmaMessageStruct or NULL. This field is not valid for receive channels.
src_addr	Source address. The address of a peripheral device must be aligned on a 32-bit boundary. ■ For transmit channels, the address points to the location from which outgoing data is to be read. ■ For receive channels, this is the address from which incoming data is to be read. This field cannot be used in fly-by mode with receive channels
dst_addr	Destination address. The address of a peripheral device must be aligned on a 32-bit boundary. ■ For ansmit channels, the address points to the location to which outgoing data is written. This field cannot be used in fly-by mode with transmit channels. ■ For receive channels, this field contains the address at which incoming data is to be stored.
length	Size of buffer; the maximum size is 32 Kb.
status	Status of DMA when the buffer is received or transmitted through the channel.
error_value	Error value returned by the DMA driver.
reserved	Reserved for internal use by the DMA driver.

Return values

```
DMA_SUCCESS
DMA_CHANNEL_UNOPENED
DMA_INVALID CHANNEL
DMA_INVALID_REQUEST
DMA_MESSAGE_OVERFLOW
DMA_SYSTEM_ERROR
DMA_CHANNEL_BUSY
DMA_INVALID_STATE
```

dmaUnloadChannel

Retrieves a processed request from the DMA channel.

For a transmit channel, a request is ready to be freed.

For a receive channel, an incoming request is waiting to be processed.

Format

```
int dmaUnloadChannel (int channel_id,
         dmaMessageType *request_msg, int wait_time);
```

Arguments

Argument	Description
channel_id	Identifier for the DMA channel that is opened.
request_msg	Pointer to the dmaMessageType structure (described in the section on dmaLoadChannel).
wait_time	Number of seconds to wait for incoming data until time expires. ■ 0 indicates no waiting ■ –1 is treated as forever

Return values

DMA_SUCCESS
DMA_CHANNEL_UNOPENED
DMA_INVALID CHANNEL
DMA_NO_REQUEST
DMA_SYSTEM_ERROR
DMA_INVALID_REQUEST

SPI Driver API

CHAPTER 14

Overview

The serial peripheral interface (SPI) driver enables applications to send and receive blocks of data via a serial port using a single read/write operation. This facilitates bi-directional data transfer.

The SPI protocol is defined in the Motorola MCC68HC11KW1 specification and in the NetSilicon hardware documentation for the chip you are using.

Include file

Using the SPI driver API requires the following header files:

- `netosio.h`
- `netos_serl.h`
- `netos_spi.h`
- `ind_io.h`
- `bool.h`

Summary of SPI driver API functions

Function	Description
`open`	Opens the serial device.
`write`	Writes a number of bytes from a specified buffer.
`read`	Reads a number of bytes into a specified buffer.
`close`	Closes the serial device.
`ioctl`	Changes the configuration of the serial port.

SPI driver API functions

The following pages describe the SPI driver API functions.

open

Opens the serial device.

Format

```
int open (const char *filename, int mode);
```

Arguments

Argument	Description
filename	One of the following: ■ /com/0 ■ /com/1
mode	How a given file will be opened: ■ O_RDONLY — for reading only ■ O_WRONLY — for writing only ■ O_RDWR — for reading or writing ■ O_DMA — for DMA mode (not recommended) ■ O_SPI_SLAVE — open as SPI slave ■ O_SPI_BLOCKSEL — block SEL signal ■ O_SPI_CLOCK_H — set the initial value of the clock signal high

O_SPI_SLAVE, O_SPI_BLOCKSEL, and O_SPI_CLOCK_HI can be set only during the first execution of the open function. Subsequent calls do not change the mode of operation.

Return values

Return value	Description
integer	File number
EINVAL	Invalid port name or invalid mode or mode combination
EBUSY	Port is already opened or another read/write or ioctl operation is in progress

write

Writes a specified number of bytes from a specified buffer.

Format

```
int write (int fno, const void *buf, int size);
```

Arguments

Argument	Description
fno	File number returned by a successful `open` call.
buf	Pointer to the buffer that contains the data to be written.
size	Number of bytes to be written: ■ In master mode: 1–32767 ■ In slave mode: When `SIO_SPI_USE_SEL_ISR` is set: `1-32767` When `SIO_SPI_USE_SEL_ISR` is not set: `40-32767`

Return values

Return value	Description
integer	Number of bytes written
`EINVAL`	Invalid port name or invalid mode or mode combination
`EBUSY`	Port is already opened or another read/write or `ioctl` operation is in progress.
`EBADF`	Port is not opened
`EAGAIN`	Operation timed out

read

Reads a specified number of bytes into a specified buffer.

Format

```
int read (int fno, void *buf, int size);
```

Arguments

Argument	Description
fno	File number returned by a successful `open` call.
buf	Pointer to the buffer containing the data to be read.
size	Number of bytes to be read. Range 1–32767.

Return values

Return value	Description
integer	Number of bytes read
EINVAL	Invalid port name or invalid mode or mode combination
EBUSY	Port is already opened or another read/write or `ioctl` operation is in progress
EBADF	Port is not opened
EAGAIN	Operation timed out

close

Closes the serial device.

Format

```
int close (int fno);
```

Arguments

Argument	Description
fno	File number returned by a successful `open` call.

Return values

Return value	Description
`EINVAL`	Invalid port name or invalid mode or mode combination
`EBUSY`	Port is already opened or another read/write or `ioctl` operation is in progress
`EBADF`	Port is not opened

ioctl

Changes the configuration of the serial port.

Format

```
int ioctl (int fno, int request, char *argp);
```

For C++:

```
int ioctl (int fno, int request, int *argp);
int ioctl (int fno, int request, int & arg)
```

Arguments

Argument	Description
fno	File number returned by a successful `open` call.
request	One of the serial I/O commands to set or get parameters (see below).
argp	Pointer to the value to set or get.

Serial I/O commands to set or get parameters

Request	Description
SIO_BAUD_SET	Sets the baud rate of serial driver (see next table).
SIO_BAUD_GET	Gets the baud rate of serial driver.
SIO_SLAVE_GET	Gets master/slave mode of operation: ■ Master=0 ■ Slave=1
SIO_RD_ADDRESS_SET	Sets EPROM address for the next read operation: ■ Master mode: 1–32767 ■ Slave mode: 1–32767 ■ Slave mode and select ISR: 40–32767 The address is automatically increased after each read operation by the number of bytes actually read.

Request	Description
SIO_RD_ADDRESS_GET	Gets EPROM address for the next read operation: ■ Master mode: 1–32767 ■ Slave mode: 1–32767 ■ Slave mode and select ISR: 40–32767
SIO_WR_ADDRESS_SET	Sets EPROM address for the next write operation: ■ Master mode: 1–32767 ■ Slave mode: 1–32767 ■ Slave mode and select ISR: 40–32767 The address automatically increases after each write operation by the number of bytes actually written.
SIO_WR_ADDRESS_GET	Gets EPROM address for the next write operation.
SIO_SET_LSB	Sets LSB/MSB mode: ■ LSB=1 ■ MSB=0
SIO_GET_LSB	Gets LSB/MSB mode ■ LSB=1 ■ MSB=0
SIO_SET_REG	Writes into control registry.
SIO_GET_REG	Reads status of the control registry
SIO_MODE_GET	Gets opened mode: ■ SIO_SPI_Direct ■ SIO_SPI_X250200
SIO_SPI_HW_GET	Gets hardware option for the SPI operation: ■ SIO_SPI_Direct ■ SIO_SPI_X250200
SIO_SPI_HW_SET	Sets hardware option for the SPI operation: ■ SIO_SPI_Direct ■ SIO_SPI_X250200
SIO_SPI_SS_SET	Sets port for generating Slave Select signal: ■ Bits 15–8: registry number ■ Bits 7–0: bit number

Request	Description
SIO_SPI_USE_SEL_ISR	Uses interrupt on the SEL signal. Bits 15-8: Port A, PORT B, PORT C, PORT D; PORT F-H
SIO_SET_SLAVE_TIMEOUT	Sets timeout value for the SPI slave mode (ticks).
SIO_GET_SLAVE_TIMEOUT	Sets timeout value for the SPI slave mode (ticks).
SIO_SET_TXCINV	Sets TXD to be driven on rising edge of TX clock: ■ 0=high-to-low transition ■ 1=low-to-high transition
SIO_SET_RXCINV	Sets RXD to be sampled on falling edge of RX clock: ■ 0=high-to-low transition ■ 1=low-to-high transition

Serial port baud rate option

Serial baud rate options are in the form SIO_*n*_BAUD where *n* is one of the following:

1200	14400	115200
2400	19200	234000
4800	28800	307200
7200	38400	460800
9600	57600	1036800

Return values

Return value	Description
EINVAL	Invalid port name or invalid mode or mode combination
EBUSY	Port already opened or another read/write or ioctl operation in progress
EBADF	Port is not opened
EAGAIN	Operation timed out

Serial EEPROM API

CHAPTER 15

Overview

EEPROM is electrically erasable programmable ROM. The serial EEPROM driver supports a family of EEPROMs, using GPIO lines on the chip to free up a chip select and the SPI interface.

For information on the EEPROM parts supported, refer to the NetSilicon hardware documentation for the chip you are using.

The serial EEPROM API lets you read and write EEPROMs, similar to the flash memory API.

Include file

Using the serial EEPROM API requires the folloiwng header file:

```
seeprom.h
```

Summary of serial EEPROM API functions

Function	Description
NASEBlockWriteProtect	Sets the write protection level for the serial EEPROM.
NASECreateSemaphores	Creates the semaphores used to synchronize access to the serial EEPROM driver API.
NASEInit	Specifies the EEPROM part and the GPIO lines used to with it.
NASEMemset	Initializes sections of the serial EEPROM with a specified character.
NASERead	Reads data from the serial EEPROM into a buffer.
NASEWrite	Writes a data buffer to the serial EEPROM.

Serial EEPROM API functions

The following pages describe the serial EEPROM API functions.

NASEBlockWriteProtect

Sets the write protection level for the serial EEPROM. Write protection settings are non-volatile — that is, they remain in effect until changed.

Format

```
int NASEBlockWriteProtect (unsigned char protectionLevel);
```

Arguments

Argument	Description
protectionLevel	One of the following options: ■ WRITE_PROTECT_LEVEL0 (none) ■ WRITE_PROTECT_LEVEL1 (quarter) ■ WRITE_PROTECT_LEVEL2 (half) ■ WRITE_PROTECT_LEVEL3 (all)

The following table lists the address ranges protected at each level for the EEPROM parts:

Level	Parts AT25080	AT25160	AT25320	AT25640
0	none	none	none	none
1	0x0300 - 0x03FF	0x0600 - 0x07FF	0x0C00 - 0x0FFF	0x1800 - 0x01FF
2	0x0200 - 0x03FF	0x0400 - 0x07FF	0x0800 - 0x0FFF	0x1000 - 0x01FF
3	0x0000 - 0x03FF	0x0000 - 0x07FF	0x0000 - 0x0FFF	0x0000 - 0x01FF

Return values

Return value	Description
NASE_SUCCESS	Successfully set the write protection
NASE_INVALID_WRITE_PROTECT_LEVEL	Invalid write protect level specified

NASECreateSemaphores

Creates the semaphores used to synchronize access to the serial EEPROM driver API. This function should be called once after powerup.

The following functions can be semaphore protected by calling `NAFlashCreateSemaphores` from the root application thread:

- `NASEWrite`
- `NASERead`
- `NASEMemset`
- `NASEBlockWriteProtect`

Format

```
int NASECreateSemaphores();
```

Arguments

none

Return values

Return value	Description
`NASE_SUCCESS`	Successfully created the semaphores
`NASE_DUPLICATE_CALL`	Semaphores already created
`NASE_SEMAPHORE_CREATE_FAILED`	Could not create semaphores

NASEInit

Specifies the EEPROM part and the GPIO lines used with it.

This function also initializes global variables and the GPIO ports. This function should be called once after powering up the development board.

Format

```
int NASEInit (unsigned long eepromType,
     unsigned long chipSelectLine, unsigned long clockLine,
     unsigned long serialIn, unsigned long serialOut);
```

Arguments

Argument	Description
eepromType	Serial EEPROM part: ■ AT25080 (1 Kb) ■ AT25160 (2 Kb) ■ AT25320 (4 Kb) ■ AT25640 (8 Kb)
chipSelectLine	GPIO line used as the chip select line. The format is: PORT*x*_BIT*n* where *x* is the port ID (C, D, or F) and *n* is an integer from 0 to 7. For example, PORTC_BIT0.
clockLine	GPIO line used as the clock line. The format is: PORT*x*_BIT*n* where *x* is the port ID (C, D, or F) and *n* is an integer from 0 to 7. For example, PORTC_BIT1.

Argument	Description
serialIn	GPIO line used as the serial-in line (relative to the EEPROM). The format is PORT*x*_BIT*n* where *x* is the port ID (C, D, or F) and *n* is an integer from 0 to 7. For example, PORTC_BIT1.
serialOut	GPIO line used as the serial-out line (relative to the EEPROM). The format is PORT*x*_BIT*n* where *x* is the port ID (C, D, F, G, or H) and *n* is an integer from 0 to 7. For example, PORTH_BIT0.

Note: IO/lines on Ports C, D, and F can be configured for both input and output. Port G and H lines can be configured only for input.

Return values

Return value	Description
NASE_SUCCESS	Successfully initialized the GPIO ports
NASE_INVALID_PART	Invalid EEPROM part specified
NASE_INVALID_CS	Invalid chip select line specified
NASE_INVALID_CLK	Invalid clock line specified
NASE_INVALID_SI	Invalid serial-in line specified
NASE_INVALID_SO	Invalid serial-out line specified
NASE_DUPLICATE_LINES	Duplicate GPIO lines specified
NASE_DUPLICATE_CALL	GPIO ports have already been initialized

NASEMemset

Initializes sections of the serial EEPROM with the specified character.

Format

```
int NASEMemset (unsigned long offset, char data,
        unsigned long length);
```

Arguments

Argument	Description
offset	Offset from the serial EEPROM memory base.
data	Character used to initialize the serial EEPROM.
length	Number of bytes to initialize.

Return values

Return value	Description
NASE_SUCCESS	Successfully initialized the EEPROM
NASE_INVALID_OFFSET	Invalid offset specified
NASE_INVALID_LENGTH	Specified length exceeds range of available memory
NASE_WRITE_FAILED	Initialization failed
NASE_VERIFY_FAILED	Data written to EEPROM does not match data read back

NASERead

Reads data from the serial EEPROM.

Format

```
int NASERead(unsigned long offset, char *buffer,
        unsigned long length);
```

Arguments

Argument	Description
offset	Offset from the serial EEPROM memory base.
buffer	Pointer to the buffer to store data read from EEPROM.
length	Number of bytes to read.

Return values

Return value	Description
NASE_SUCCESS	Successfully read the data
NASE_INVALID_OFFSET	Invalid offset specified
NASE_INVALID_LENGTH	Specified length exceeds range of available memory

NASEWrite

Writes a data buffer to the serial EEPROM.

Format

```
int NASEWrite(unsigned long offset, char *buffer,
        unsigned long length);
```

Arguments

Argument	Description
offset	Offset from the serial EEPROM memory base.
buffer	Pointer to the data buffer to write.
length	Number of bytes to write.

Return values

Return value	Description
NASE_SUCCESS	Successfully wrote the data
NASE_INVALID_OFFSET	Invalid offset specified
NASE_INVALID_LENGTH	Specified length exceeds range of available memory
NASE_WRITE_FAILED	Write operation failed
NASE_VERIFY_FAILED	Data written to EEPROM does not match data read back

Index

A

B

C

D

E

F

H

I

J

K

L

M

N

P

R

S

X


NetSilicon
A Digi International® Company