ECHELON

I/O Model Reference for Smart Transceivers and Neuron Chips

Develop I/O interfaces for Series 6000, 5000, and 3100 chips and Smart Transceivers.

078-0392-01C

Echelon Corporation
www.echelon.com

Welcome

For the various input/ouput (I/O) pins on an Echelon® Smart Transceiver, an Echelon Neuron® 5000 or Neuron 6000 Processor, or Series 3100 Neuron Chip, Echelon's Neuron C programming language provides a set of *I/O models* that allow a program to define *I/O objects*. An I/O model is an abstract definition (including the relevant firmware driver) of hardware I/O for a Neuron Chip or Smart Transceiver; an I/O object is software instance of the specific I/O model. Together, they provide programmable access to one or more I/O pins in a specified configuration and for a specified input or output waveform definition.

This document describes the many different I/O models that are available for use with the Neuron Chips and Smart Transceivers. With only a few exceptions, an I/O model can be used with any Series 3100 device, Series 5000 or Series 6000 device. Where applicable, this document identifies differences in the I/O models that are specific to a particular device type.

The information in this document supersedes the equivalent information for Series 3100 devices contained in the *FT 3120 / FT 3150 Smart Transceiver Data Book*, *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book*, *Neuron C Programmer's Guide*, and *Neuron C Reference Guide*.

Audience

This document assumes that you have a good understanding of general Neuron C language programming concepts and techniques. It also assumes that you are familiar with the device requirements for Neuron Chips and Smart Transceivers.

Related Documentation

The following manuals are available from the Echelon Web site (www.echelon.com) and provide additional information that can help you develop applications for Neuron Chip or Smart Transceiver devices:

- *FT 3120 / FT 3150 Smart Transceiver Data Book* (005-0139-01D). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 3120® and FT 3150® Smart Transceivers.
- *Introduction to the LONWORKS Platform* (078-0391-01A). This manual provides an introduction to the ISO/IEC 14908 (ANSI/CEA-709.1 and EN14908) Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.
- *LONMARK® Application Layer Interoperability Guidelines.* This manual describes design guidelines for developing applications for open interoperable LONWORKS devices, and is available from the LONMARK Web site, www.lonmark.org.
- *NodeBuilder® FX User's Guide* (078-0516-01This manual describes how to develop a LONWORKS device using the NodeBuilder tool.

- *Neuron C Programmer's Guide* (078-0002-01I). This manual describes how to write programs using the Neuron C Version 2.2 programming language.
- *Neuron C Reference Guide* (078-0140-01G). This manual provides reference information for writing programs using the Neuron C Version 2.2 programming language.
- *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book* (005-0193-01A). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the PL 3120, PL 3150, and PL 3170™ Smart Transceivers.
- *Series 5000 Chip Data Book* (005-0199-01A). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the Neuron 5000 Chips and FT 5000 Smart Transceivers.
- *Series 6000 Chip Data Book (*005-0230-01*).* This manual provides the detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the Neuron 6000 or FT 6000 Smart Transceivers.

All of the Echelon documentation is available in Adobe® PDF format. To view the PDF files, you must have a current version of the Adobe Reader®, which you can download from Adobe at: www.adobe.com/products/acrobat/readstep2.html.

Table of Contents

1

Introduction

This chapter provides an overview of the available I/O models for Series 3100 devices, Series 5000, and Series 6000 devices. It includes considerations for hardware, programming, and timing.

Overview

Echelon's Neuron Chips and Smart Transceivers connect to application-specific external hardware through 11 or 12 I/O pins, named IO0-IO11. You can configure these pins to provide flexible input and output (I/O) functions with minimal external circuitry. These functions are described as *I/O models*.

The Neuron C programming language allows the application programmer to declare *I/O objects* that use one or more I/O pins. An I/O object is a software instance of an I/O model, and provides programmable access to an I/O driver for a specified on-chip I/O hardware configuration and a specified input or output waveform definition. Programs can then refer to most of these objects through **io_in()** and **io_out()** system calls to perform the actual input or output function during execution of the program. Because events are associated with changes in input values, the task scheduler can execute associated application code when these changes occur.

There are many different I/O models available for use with the Neuron Chips and Smart Transceivers. Most I/O models are available in system images by default. If an I/O model is required by an application, but is not included in the default system image, the development tool links the appropriate models into available memory space. For FT 3120, PL 3120, and PL 3170 Smart Transceiver designs, this linkage means that internal EEPROM space must be used for the additional model. For FT 3150 or PL 3150 Smart Transceiver designs, the model is added to an external flash or EEPROM region beyond the 16 KB space reserved for the system image. For Series 5000 and Series 6000 device designs, the model is added to the application image.

Series 5000 and Series 6000 chips also support application-specific interrupts, which can trigger on either or both edges, or on either level, for any of the I/O pins, regardless of any associated I/O object. See the *Neuron C Programmer's Guide* for more information about interrupts.

Summary of the Available I/O Models

Many I/O models are available for Neuron Chips and Smart Transceivers. Certain I/O models are available only for specific chip types, but most are available to all Neuron Chips and Smart Transceivers. The I/O models are grouped into the following categories:

- *Direct I/O Models* are based on a logic level at the I/O pins; none of the Neuron Chip or Smart Transceiver hardware's timer/counters are used in conjunction with these I/O models. These models can be used in multiple, overlapping combinations within the same Neuron Chip or Smart Transceiver. Direct I/O models include the following types:

Input Model Types	*Output Model Types*
bit	**bit**
byte	**byte**
leveldetect	**nibble**
nibble	**touch**
touch	

- *Timer/Counter I/O Models* use a timer/counter circuit in the Neuron Chip or Smart Transceiver. Each Neuron Chip and each Smart Transceiver has two timer/counter circuits: One whose input can be multiplexed, and one with a dedicated input. Timer/counter I/O models include the following types:

Input Model Types	*Output Model Types*
dualslope	**edgedivide**
edgelog	**frequency**
infrared	**infrared_pattern**
ontime	**oneshot**
period	**pulsecount**
pulsecount	**pulsewidth**
quadrature	**stretchedtriac**
totalcount	**triac**
	triggeredcount

- *Serial I/O Models* are used for transferring data serially over a pin or a set of pins. The **neurowire**, **i2c**, **magcard**, **magcard_bitstream**, **magtrack1**, and **serial** I/O models are mutually exclusive within a single Neuron Chip or Smart Transceiver. Both the input and output versions of a serial I/O model can coexist within a single Neuron Chip or Smart Transceiver. Serial I/O models include the following types:

Serial Input Model Types	*Serial Output Model Types*
bitshift	**bitshift**
magcard	**serial**
magcard_bitstream	
magtrack1	
serial	
wiegand	

Serial Input/Output Model Types
i2c
neurowire
sci
spi

- *Parallel I/O Models* are used for high-speed bidirectional I/O. I/O models within this group use all of the Neuron Chip or Smart Transceiver I/O pins. The parallel I/O models include the following types:

Parallel Input/Output Model Types
muxbus
parallel

Table 1 through 5 list the available I/O models within each category. **Figure 1** summarizes the pin configuration for each of the I/O models. A single device can use various I/O models of different types simultaneously.

Table 1. Summary of the Direct I/O Models

I/O Model	Applicable I/O Pins	Total Pins per Object	Input/Output Value
Bit Input[1]	IO0 – IO11	1	0, 1 binary data
Bit Output[1]	IO0 – IO11	1	0, 1 binary data
Byte Input	IO0 – IO7	8	0 – 255 binary data
Byte Output	IO0 – IO7	8	0 – 255 binary data
Leveldetect Input	IO0 – IO7	1	Logic 0 level detected
Nibble Input	Any adjacent four in IO0 – IO7	4	0 – 15 binary data
Nibble Output	Any adjacent four in IO0 – IO7	4	0 – 15 binary data
Touch I/O	IO0 – IO7	1	Up to 2048 bits of input or output bits

Notes:

1. The IO11 pin for this I/O model is available only for the following device types: PL 3120-E4, PL 3150, PL 3170, FT 5000, Neuron 5000, FT 6000 and Neuron 6000.

See Chapter 2, *Direct I/O Models*, for more information about the direct I/O models.

Table 2. Summary of the Parallel I/O Models

I/O Model	Applicable I/O Pins	Total Pins per Object	Input/Output Value
Muxbus I/O	IO0 – IO10	11	Parallel bidirectional port using multiplexed addressing
Parallel I/O[1]	IO0 – IO11	12	Parallel bidirectional handshaking port

Notes:

1. The IO11 pin for this I/O model is available only for the following device types: FT 5000, Neuron 5000, FT 6000 and Neuron 6000.

See Chapter 3, *Parallel I/O Models*, for more information about the parallel I/O models.

Table 3. Summary of the Serial I/O Models

I/O Model	Applicable I/O Pins	Total Pins per Object	Input/Output Value
Bitshift Input	Any adjacent pair (except IO7 + IO8 & IO10 + IO11)	2	Up to 16 bits of clocked data
Bitshift Output	Any adjacent pair (except IO7 + IO8 & IO10 + IO11)	2	Up to 16 bits of clocked data
I^2C	IO8 + IO9 or IO0 + IO1	2	Up to 255 bytes of bidirectional serial data
Magcard Bitstream	IO8 + IO9 + (one of IO0 – IO7)	2 or 3	Unprocessed serial data stream from a magnetic card reader
Magcard Input	IO8 + IO9 + (one of IO0 – IO7)	2 or 3	Encoded ISO7811 track 2 data stream from a magnetic card reader
Magtrack1	IO8 + IO9 + (one of IO0 – IO7)	2 or 3	Encoded ISO3554 track 1 data stream from a magnetic card reader
Neurowire I/O	IO8 + IO9 + IO10 + (one of IO0 – IO7)	4	Up to 256 bits of bidirectional serial data
SCI (UART)[1]	IO8 + IO10	2	Up to 255 bytes input and 255 bytes output
Serial Input	IO8	1	8-bit characters
Serial Output	IO10	1	8-bit characters
SPI	IO8 + IO9 + IO10 + (IO7)	3 or 4	Up to 255 byes of bidirectional data
Wiegand Input	Any adjacent pair in IO0 – IO7	2	Encoded data stream from Wiegand card reader

Notes:

1. The SCI (UART) model is available only for the following device types: PL 3120-E4, PL 3150, PL 3170, FT 5000, Neuron 5000, FT 6000 and Neuron 6000.

See Chapter 4, *Serial I/O Models*, for more information about the serial I/O models.

Table 4. Summary of the Timer/Counter Input Models

I/O Model	Applicable I/O Pins	Total Pins per Object	Input/Output Value
Dualslope Input	IO0, IO1 + (one of IO4 – IO7)	2	Comparator output of the dualslope converter logic
Edgelog Input	IO4	1	A stream of input transitions
Infrared Input	IO4 – IO7	1	Encoded data stream from an infrared demodulator
Ontime Input	IO4 – IO7	1	Pulse width of 0.2 µs – 1.678 s
Period Input	IO4 – IO7	1	Signal period of 0.2 µs – 1.678 s
Pulsecount Input	IO4 – IO7	1	0 – 65,535 input edges during 0.839 s
Quadrature Input	IO4 + IO5, IO6 + IO7	2	± 16,383 binary Gray code transitions
Totalcount Input	IO4 – IO7	1	0 – 65,535 input edges

See Chapter 5, *Timer/Counter Input Models*, for more information about the timer/counter input models.

Table 5. Summary of the Timer/Counter Output Models

I/O Model	Applicable I/O Pins	Total Pins per Object	Input/Output Value
Edgedivide Output	IO0, IO1 + (one of IO4 – IO7)	2	Output frequency is the input frequency divided by a user-specified number
Frequency Output	IO0, IO1	1	Square wave of 0.3 Hz to 2.5 MHz
Infrared Pattern Output	IO0, IO1	1	Series of timed repeating square wave output signals
Oneshot Output	IO0, IO1	1	Pulse of duration 0.2 µs to 1.678 s
Pulsecount Output	IO0, IO1	1	0 – 65,535 pulses

I/O Model	Applicable I/O Pins	Total Pins per Object	Input/Output Value
Pulsewidth Output	IO0, IO1	1	0 – 100% duty cycle pulse train
Stretched Triac Output[1]	IO0, IO1 + (one of IO4 – IO7)	2	Delay of output pulse with respect to input edge
Triac Output[2]	IO0, IO1 + (one of IO4 – IO7)	2	Delay of output pulse with respect to input edge
Triggered-Count Output	IO0, IO1 + (one of IO4 – IO7)	2	Output pulse controlled by counting input edges

Notes:

1. The Stretched Triac Output model is available only for the following device types: FT 5000, Neuron 5000, FT 6000, and Neuron 6000.
2. Dual-edge triggering is not available for the following device types: Neuron 3150, FT 3150, or PL 3150.

See Chapter 6, *Timer/Counter Output Models*, for more information about the timer/counter output models.

Neuron Chips and Smart Transceivers have two 16-bit timer/counters on-chip. The input to timer/counter 1, also called the *multiplexed timer/counter,* is selectable among pins IO4 – IO7, through a programmable multiplexer and its output can be connected to pin IO0. The input to timer/counter 2, also called the *dedicated timer/counter,* can be connected to pin IO4 and its output to pin IO1.

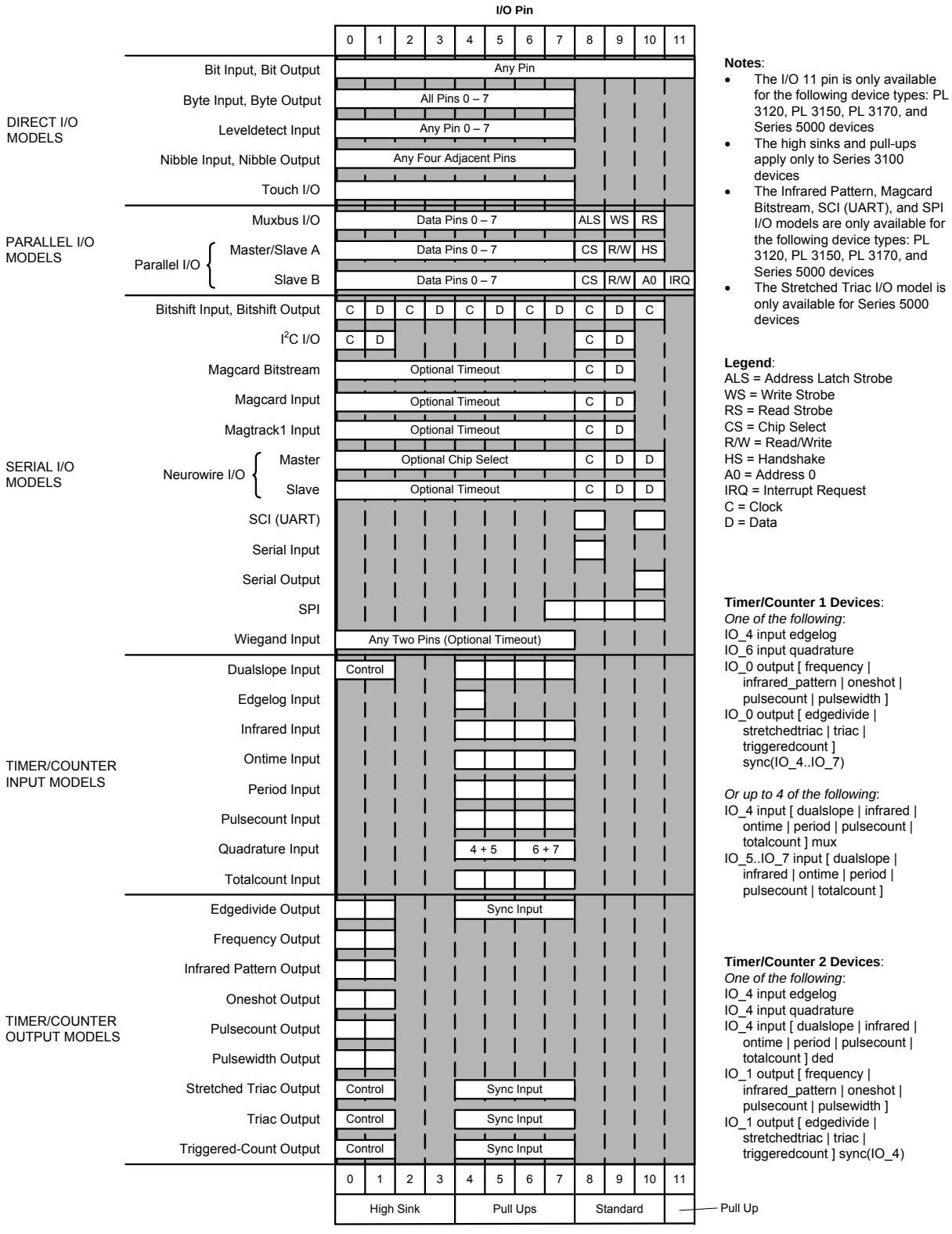
The timer/counters are implemented as a 16-bit load register writable by the CPU, a 16-bit counter, and a 16-bit latch readable by the CPU. The load register and latch are accessed a byte at a time. No I/O pins are dedicated to timer/counter functions. If, for example, timer/counter 1 is used for input signals only, then IO0 is available for other input or output functions. Timer/counter clock and enable inputs can be from external pins, or from scaled clocks derived from the system clock; the clock rates of the two timer/counters are independent of each other. External clock actions occur optionally on the rising edge, the falling edge, or both rising and falling edges of the input.

For Series 5000 and Series 6000 devices, many of the timer/counter I/O models can also trigger interrupt tasks, which can provide minimum application latency for I/O events that are related to the timer/counter models. See the *Neuron C Programmer's Guide* for more information about defining and using interrupts for Series 5000 and 6000 devices.

Multiple timer/counter input objects can be declared on different pins within a single application. By calling the **io_select()** function, the application can use the first timer/counter to implement up to four different input objects. If a timer/counter is configured to implement one of the output models, or is configured as a quadrature input object, then it can not be reassigned to another timer/counter object in the same application program.

The following guidelines for declaring I/O object types apply to the I/O models shown in **Figure 1**:

- Up to 16 I/O objects can be declared.
- Timer/counter 1 can be multiplexed for up to four input objects.
- The **neurowire**, **i2c**, **magcard**, **magcard_bitstream**, **magtrack1**, and **serial** I/O models are mutually exclusive. One or more of a single type of these I/O models can be declared in one program.
- Because the **parallel** and **muxbus** I/O models require all I/O pins for some Neuron Chips and Smart Transceivers, no other object types can be declared when either of these objects is declared. You can declare the IO11 pin as a **bit** input or output in addition to the **parallel** or **muxbus** object for the following device types: PL 3120-E4, PL 3150, or PL 3170. For Series 5000 and Series 6000 devices, you can also declare the IO11 pin as a **bit** input or output in addition to the **parallel** (master or slave A mode) or **muxbus** object; the IO11 pin serves as an IRQ pin for the **parallel** (slave B mode) object.
- Direct I/O object types (such as **bit**, **nibble**, **byte**) can be declared in any combination; see *Overlaying I/O Objects*. Timer/counter, **serial**, and **neurowire** I/O object declarations override the pin directions of any overlaying direct I/O object types.
- The **quadrature** and **dualslope** input objects cannot be multiplexed with other input objects on timer/counter 1. The **edgelog** input uses both timer/counters and is exclusive of any other timer/counter objects.
- The **bitshift** I/O objects cannot be declared on the same I/O pins as timer/counter objects. Direct I/O objects can be overlaid with **bitshift** I/O objects. Two adjacent **bitshift** I/O objects cannot share any I/O pins.



Notes:

- The I/O 11 pin is only available for the following device types: PL 3120, PL 3150, PL 3170, and Series 5000 devices
- The high sinks and pull-ups apply only to Series 3100 devices
- The Infrared Pattern, Magcard Bitstream, SCI (UART), and SPI I/O models are only available for the following device types: PL 3120, PL 3150, PL 3170, and Series 5000 devices
- The Stretched Triac I/O model is only available for Series 5000 devices

Legend:
ALS = Address Latch Strobe
WS = Write Strobe
RS = Read Strobe
CS = Chip Select
R/W = Read/Write
HS = Handshake
A0 = Address 0
IRQ = Interrupt Request
C = Clock
D = Data

Timer/Counter 1 Devices:
One of the following:
IO_4 input edgelog
IO_6 input quadrature
IO_0 output [frequency | infrared_pattern | oneshot | pulsecount | pulsewidth]
IO_0 output [edgedivide | stretchedtriac | triac | triggeredcount] sync(IO_4..IO_7)

Or up to 4 of the following:
IO_4 input [dualslope | infrared | ontime | period | pulsecount | totalcount] mux
IO_5..IO_7 input [dualslope | infrared | ontime | period | pulsecount | totalcount]

Timer/Counter 2 Devices:
One of the following:
IO_4 input edgelog
IO_4 input quadrature
IO_4 input [dualslope | infrared | ontime | period | pulsecount | totalcount] ded
IO_1 output [frequency | infrared_pattern | oneshot | pulsecount | pulsewidth]
IO_1 output [edgedivide | stretchedtriac | triac | triggeredcount] sync(IO_4)

Figure 1. Pin Configuration Summary for the I/O Models

Example: The following I/O models can be combined for a Neuron Chip or Smart Transceiver:

- 1 **parallel** I/O model (on **IO_0**..**IO10**)

  OR

- 1 **muxbus** I/O model (on **IO_0**..**IO10**)

  OR

- A combination of any or all of the other I/O models *A* through *E* shown in **Table 6**:

Table 6. Example I/O Model Combinations

A	B	C	D	E
1 to 4 timer/counter inputs (multiplexed on **IO_4**, **IO_5**, **IO_6**, **IO_7**), including **quadrature** input on **IO_6** OR 1 timer/counter output (on **IO_0**)	1 timer/counter input (on **IO_4**), including **quadrature** input on **IO_4** OR 1 timer/counter output (on **IO_1**)	1 **neurowire** I/O object (on **IO_8**, **IO_9**, **IO_10**) and 1 of **IO_0** ... **IO_7** OR 1 serial I/O object type (on **IO_8**, **IO_10**)	Any direct I/O object type on any pin (**IO_0** through **IO_10**)	A bit I/O object on **IO_11**

Hardware Considerations

For a description of the electrical characteristics of the I/O pins, refer to the appropriate Series 3100, Series 5000, or Series 6000 device data sheet. Pins that are configured as outputs can also be read as inputs, returning the value that was last written to the pin. In addition, an application program can optionally specify the initial values of digital outputs.

For Series 3100 devices, pins IO4 – IO7 and IO11 have optional pull-up current sources that act as pull-up resistors. You use a Neuron C compiler directive (**#pragma enable_io_pullups**) to enable these pull-ups. Also for Series 3100 devices, pins IO0 – IO3 have high current-sink capability (20 mA); the other pins have standard current-sink capability.

For Series 3100 FT Smart Transceivers, the I/O pull-ups are enabled during the stack initialization and built-in self-test (BIST) task (see the *FT 3120 / FT 3150 Smart Transceiver Data Book* for more information about the stack initialization and BIST task). However, for Series 3100 PL Smart Transceivers, the I/O pull-ups are not enabled during the stack initialization and BIST task.

Recommendation: For Series 3100 PL Smart Transceivers (especially for devices with energy storage power supplies), you must ensure that I/O pins that

are not used by the application are tied high or low on the PC board, or are left unconnected and configured as a bit output by the application in order to prevent unnecessary power consumption. See the *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book* for more information.

For Series 5000 and Series 6000 devices, the I/O pins do not have configurable pull-ups or high current-sink capability. If your I/O circuitry requires pull-up resistors, you must add them to the hardware design for the device. The I/O pins on a Series 5000 device have an 8 mA current source and sink capability. If your I/O circuitry has higher current requirements, you can add external driver circuitry (for example, using a Fairchild Semiconductor® 74AC245/74ACT245 Octal Bidirectional Transceiver or 74VHC245/74VHCT245 Octal Buffer/Line Driver).

In addition, the Series 5000 and Series 6000 device pins are all 3.3 V pins: the input pins are 5 V tolerant, and the output pins are CMOS compatible. Series 3100 device pins are all 5 V pins.

For Series 3100, Series 5000, and Series 6000 devices, pins IO0 – IO7 have low-level detect latches.

Because the I/O pins are controlled by system firmware, the timing for reading or writing an I/O pin includes latency that can vary by I/O model and even vary by I/O pin. All inputs are software sampled during processing for the Neuron C **when** statement. In general, the latency scales inversely with the system clock rate.

To maintain and provide consistent behavior for external events, and to prevent setup and hold metastability, all I/O pins, when configured as simple inputs, are passed through a hardware synchronization block, shown in **Figure 2**, that is sampled by the internal system clock.

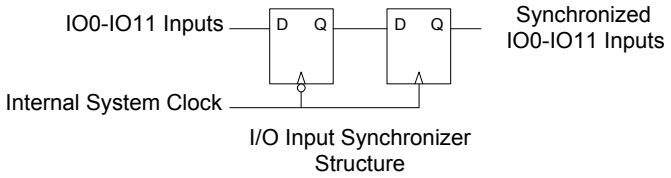


Figure 2. Synchronization Block

I/O pins used for other functions do not have this synchronization requirement.

For Series 3100 devices, the sample rate is always the input clock divided by two (for example, for a 10 MHz input clock, the sample rate is 5 MHz). For a signal to be reliably synchronized with a 10 MHz input clock, it must be at least 220 ns in duration; see **Figure 3**.

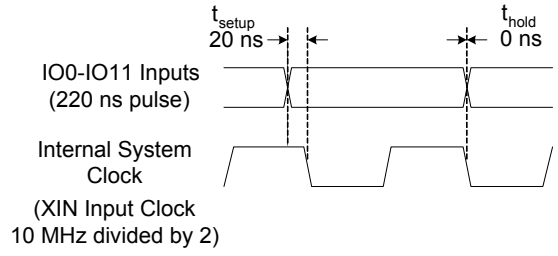


Figure 3. Synchronization of External Signals for Series 3100 Devices

For Series 5000 and Series 6000 devices, the sample rate is equivalent to the system clock rate. For a signal to be reliably synchronized with an 80 MHz system clock, it must be at least 17.5 ns in duration; see **Figure 4**.

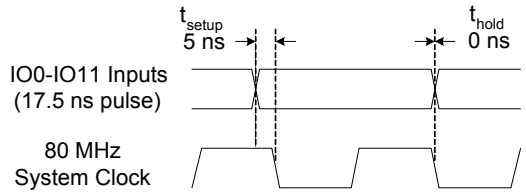


Figure 4. Synchronization of External Signals for Series 5000 Devices

Any event that lasts longer than 220 ns (for a Series 3100 device at 10 MHz) or 17.5 ns (for a Series 5000 or Series 6000 device at 80 MHz) is synchronized by hardware, but there can be latency in software sampling, which can result in a delay in detecting the event. If the state changes at a faster rate than software sampling can process, the interim changes are not detected.

The following exceptions apply to the use of the synchronization block:

- The chip select (CS~) input used in the slave B mode of the parallel I/O object recognizes rising edges asynchronously.
- The **leveldetect** input is latched by a flip-flop with a 200 ns clock (for Series 3100 devices) or a 12.5 ns clock (for Series 5000 or Series 6000 devices). The level detect transition event is latched, but there is a delay in software detection.
- The SCI (UART) and SPI objects are buffered on byte boundaries by the hardware, and are transferred to memory using an interrupt.
- Events on the I/O pins for the input timer/counter functions are accurately measured, and a value returned to a register, regardless of the state of the application or interrupt processor within the Neuron Chip or Smart Transceiver. However, the application processor can be delayed in reading the register.

I/O Timing Issues

The I/O timing for Neuron Chips and Smart Transceivers is influenced by four separate, yet overlapping, areas of the overall chip architecture:

- The scheduler
- The I/O model's firmware
- The Neuron Chip or Smart Transceiver hardware
- Interrupts

The contribution of the scheduler to the overall I/O timing is approximately uniform across all I/O objects because its contribution to the overall I/O timing is at a relatively high functional level.

The contribution of both firmware and hardware varies from one I/O model to another (for example, Bit I/O as opposed to Neurowire I/O).

The contribution of interrupts varies with the nature of the data interrupting the processor. See *SCI (UART) Input/Output* and *SPI Input/Output* for more

information. Also, for Series 5000 and Series 6000 devices, when hardware interrupt tasks run in the application (APP) processor (for the two lowest clock rates), the contribution of interrupt processing, including the application-specific interrupt tasks, directly adds to the scheduler delay. However, at higher clock rates, the contribution of interrupts is very small and approximately constant.

Scheduler-Related I/O Timing Information

As part of the Neuron system firmware, the scheduler provides an orderly and predictable means to facilitate the evaluation of user-defined events. The **when** clause, provided by the Neuron C language, is used to specify such events. For more information on the operation of the scheduler, see the *Neuron C Programmer's Guide.*

There is a finite latency associated with the operation of the scheduler. The time required for the scheduler to evaluate the same **when** clause in a particular user application program is, to a large extent, a function of the size of the user code, the total number of **when** clauses, and the state of the events associated with those **when** clauses. Therefore, it is impractical to specify a nominal value for this latency, because each application has its own distinct behavior.

The best-case latency can be viewed in several ways, each exposing a different aspect of the operation of the scheduler. A simple example consists of having an application program that consists of two **when** clauses, both of which always evaluate to TRUE, as shown below.

```
IO_0 output bit testbit;

when (TRUE) {
  io_out (testbit, 1);
}

when (TRUE) {
  io_out (testbit, 0);
}
```

Processing of **when** clauses is performed in a round-robin fashion; therefore, the Neuron C code above performs alternating activation of the IO0 pin (in this case, to isolate and extract the timing parameters associated with the scheduler). The waveform seen on pin IO0 of the device, as a result of the above code, is shown in **Figure 5**.

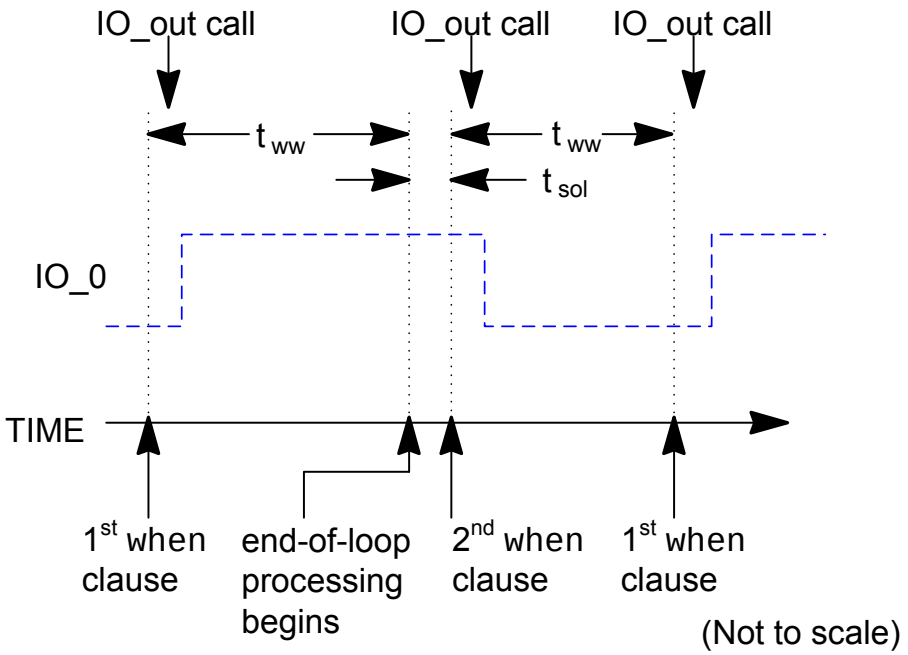


Figure 5. Scheduler Overhead Latency: when Clause to when Clause

The when-clause to when-clause latency, t_{ww}, in this case includes the execution time of one **io_out()** function (which for a Series 3100 device with a 10 MHz input clock, has approximately 65 µs latency; for a Series 5000 or Series 6000 device with an 80 MHz system clock, this latency is approximately 4 µs) and applies to an event that always evaluates to TRUE. The actual t_{ww} for a particular application depends on the actual task within the **when** statement as well as the **when** event that is evaluated.

The above example not only measures the best-case minimum latency between consecutive **when** clauses (whose events evaluate to TRUE), but also reveals the scheduler's end-of-loop overhead latency, t_{sol}. As shown in **Figure 5**, t_{ww} is the off-time period of the output waveform, and t_{sol} is the on-time of the output waveform, minus t_{ww}. The scheduler overhead latency, or the scheduler end-of-loop latency, occurs just before the execution of the last **when** clause in the program.

The latency associated with the return from the **io_out()** function is small, relative to that of the execution of the function call itself.

Note: Some I/O models suspend application processing until the task is complete because they are firmware-driven. These I/O models include: bitshift, Neurowire, parallel, software serial I/O models, I²C, magcard, magtrack, Touch I/O, and Wiegand. However, they do not suspend network communication (which is handled by the network processor and the media access processor).

Firmware and Hardware-Related I/O Timing Information

All I/O updates in a Neuron Chip or Smart Transceiver are performed by the Neuron firmware using system image function calls.

The total latency for a particular function call, from start to end, has two separate parts:

- Processing time required before the actual hardware I/O update (read or write) occurs
- The time required to finish the current function call and return to the application program

Overall accuracy is always related to the accuracy of the clock in (CLK1 or XIN) input of the Neuron Chip or Smart Transceiver. Timing diagrams are provided for all non-trivial cases to clarify the parameters given.

Programming Considerations

Before performing I/O, you must first declare the I/O objects that monitor and control the 11 or 12 Neuron Chip or Smart Transceiver I/O pins, named IO0, IO1, ..., IO11. By default, any undeclared pin is unused, and is deactivated. In the deactivated state, the pin is in a high-impedance state. The declaration syntax for I/O objects is described in detail in subsequent chapters of this manual.

Note: Unused *input* pins must have pull-up resistors. For Series 3100 devices, you can use the **enable_io_pullups** compiler directive for pins IO4 through IO7 (see the *Compiler Directives* chapter of the *Neuron C Reference Guide* for more information on this directive). For Series 3100 power line devices, this directive also enables the pull-up for the IO11 pin. You can define unused pins as *outputs* to avoid using pull-ups.

To perform I/O, you normally use the built-in I/O functions: **io_in( )**, **io_out( )**, **io_set_direction( ), io_select( )**, **io_change_init( )**, and **io_set_clock( )**. The **io_out_request( )** function is used to perform I/O with a **parallel** I/O object. See *Performing I/O: Functions and Events* for more information about these functions.

I/O objects can also be linked to Neuron C events, because changes in I/O often affect task scheduling. See *I/O Events* for a description of the **io_changes** and **io_update_occurs** events, which are the I/O-related events that are used in **when** clauses.

Timer/Counter I/O devices can also be linked to Neuron C interrupt tasks, allowing for low-latency application-specific response to certain events. The interrupt trigger is defined by the timer/counter I/O model in use.

All I/O pins IO0..IO11 can also be used to define one or two I/O interrupt tasks, allowing for low-latency application-specific response to a positive or negative level, a rising or falling edge, or any edge sampled on that I/O pin. I/O interrupts operate independently from any I/O devices that are associated with the same pins.

See the *Neuron C Programmer's Guide* for more information about application-specific interrupts.

For more detailed information on, and additional examples of using I/O, see the following LONWORKS engineering bulletins:

- *Analog-to-Digital Conversion with the Neuron Chip* engineering bulletin (part no. 005-0019-01)

- *Driving a Seven Segment Display with the Neuron Chip* engineering bulletin (part no. 005-0014-01)
- *Neuron Chip Quadrature Input Function Interface* engineering bulletin (part no. 005-0003-01)
- *Parallel I/O Interface to the Neuron Chip* engineering bulletin (part no. 005-0021-01)
- *EIA-232C Serial Interfacing with the Neuron Chip* engineering bulletin (part no. 005-0008-01)

Declaring I/O Objects in Neuron C

Declaring an I/O object in a Neuron C application performs all of the following tasks:

- Informs the compiler what type of I/O operation will be performed, and on which pin or pins. The compiler creates instructions that configure the hardware within the Neuron core as a result of this declaration. The firmware configures the hardware whenever the device or application is reset.
- Associates the name of the I/O object with an I/O model.
- Associates the I/O object with one or more I/O pins, and defines additional properties of the I/O object.

The general syntax for declaring an I/O objects in the Neuron C language is shown below.

pin direction model [options] io-object-name;

pin

One of the Neuron C keywords that name one of the twelve I/O pins, **IO_0** through **IO_11** (the IO11 pin is available only on Series 3100 power line devices and Series 5000 devices). The named pin defines the first pin for multi-pin I/O models. In general, pins can appear in a single object declaration only. However, a pin can appear in multiple declarations of the **bit**, **nibble**, and **byte** I/O object types. Also, **IO_8** can appear in multiple declarations of **neurowire master** specifying different **select** pins. In this case, it is not required that all declarations have the same direction, that is, input or output; see *Overlaying I/O Objects*.

direction

Specifies whether the object is an input or an output. Some I/O models are bidirectional, and do not require the specification of direction.

model

Specifies the I/O model for this I/O object.

options

Optional I/O parameters, dependent on the chosen *model* for the I/O object. The description of each model includes the model's available options. Except where noted, these options can be listed in any order. All options have default values that are used when you do not include the option in the object declaration.

io-object-name

A user-supplied name for the I/O object, in the ANSI C format for variable identifiers.

The description for each I/O object includes a detailed explanation of the syntax for each I/O model.

Example: A logic level needs to be measured at the IO3 input pin of the device, which is named **IO_3** in Neuron C. The pin is connected to a proximity detector, as its programmatic name indicates.

```
IO_3 input bit ioProximity;
```

Your program can now refer to the IO3 binary input through the **ioProximity** variable when using utility functions for this I/O object.

Overlaying I/O Objects

For some I/O models, you can declare more than one I/O object for the same pin. That is, you can *overlay* one I/O object on another.

Example 1: The following declarations allow a program to read four adjacent pins in one operation (with the **nibble** I/O model) or read each pin individually (with the **bit** I/O model):

```
IO_4 input nibble ioAllPoints;
IO_4 input bit ioPoint1;
IO_5 input bit ioPoint2;
IO_6 input bit ioPoint3;
IO_7 input bit ioPoint4;
```

Example 2: The following declarations enable a program to monitor (read back) the level on its own **oneshot output** object:

```
IO_1 output oneshot clock (3) ioBreakHigh;
IO_1 input bit ioBreakHighLevel;
```

With respect to overlaying, I/O models can be divided into hard pin direction I/O models and soft pin direction I/O models:

- The *soft* pin direction I/O models (**bit**, **nibble**, and **byte**) are changed by subsequent pin declarations. When multiple soft pin direction I/O objects are declared for the same pin, the last soft I/O object declared is the one that affects the initial direction of the pin at run-time.
- The *hard* pin direction I/O models (all other I/O models) are not affected by subsequent declarations.

The **io_set_direction( )** function allows the application to change the direction of any **bit**, **nibble**, or **byte** type I/O object at run time. See the *Neuron C Reference Guide* for information about the **io_set_direction( )** function.

In example 2 above, the **oneshot** model is a hard pin direction I/O model, but the **bit** model is a soft pin direction I/O model. The order of declarations is not important, and the **oneshot** object is the one that affects the direction of pin IO1 (set during initialization and after reset).

Example 3: If a program declares the following:

```
IO_2 input  bit ioPoint1;
IO_2 output bit ioPoint2;
```

The IO2 pin is an output bit I/O object (because the output is declared last). Assuming that the **io_set_direction( )** function is not called, a subsequent call to the **io_out( )** function for **ioPoint2** sets the level of this pin. A call to the **io_in( )** function for **ioPoint1** can then be used to read back the actual pin level of this output object.

Multiplexing I/O Models

Input to one of the timer/counter circuits can be multiplexed among pins **IO_4** to **IO_7** or provide output to **IO_0**. This timer/counter is called Timer/Counter 1 or the *multiplexed* timer/counter. A second timer/counter circuit derives input only from **IO_4** or provides output to **IO_1**. This second timer/counter circuit is called Timer/Counter 2 or the *dedicated* timer/counter. **Figure 6** shows a signal flow diagram for both the multiplexed and dedicated timer/counter circuits.

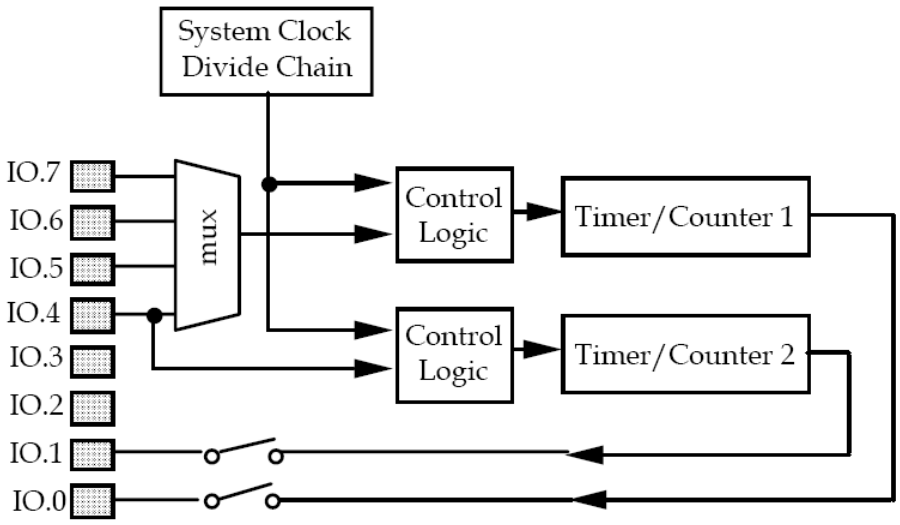


Figure 6. Flow Diagram for Timer/Counter Circuits

Performing I/O: Functions and Events

A Neuron C application program can access I/O objects in either of the following ways:

- By using an explicit **io_in( )** or **io_out()** function
- By referring to an event associated with the object in a **when** clause
- For timer/counter objects, by using the **io_select()** function

The following sections describe both methods.

General I/O Functions

After you declare the I/O objects for a Neuron C application, you can access the objects through the I/O functions that are provided by Neuron C language. **Table 7** lists these functions. You do not need to declare or link these functions

because they are included by the Neuron C compiler. The compiler enforces type checking for the parameters of these functions.

Table 7. General I/O Functions

Function	Description
io_change_init( )	Initializes the value of an input object for the **io_changes** event
io_edgelog_preload( )	Sets the timer/counter preload value for an **edgelog** I/O object
io_edgelog_single_preload( )	Sets the timer/counter preload value for an **edgelog single_tc** I/O object
io_in( )	Reads data from an I/O object
io_in_ready( )	An event function that evaluates to TRUE when a block of data is available to be read from a **parallel** I/O object
io_in_request( )	Starts an I/O input cycle for a **dualslope** I/O object
io_out( )	Writes data to an I/O object
io_out_request( )	Requests the write token for a **parallel** I/O object
io_preserve_input( )	Causes the first value obtained from a timer/counter after reset or an **io_select( )** to be considered valid
io_select( )	Selects one of the multiplexed input objects (see *Multiplexing I/O Models*)
io_set_baud( )	Changes the bit rate setting for the specified object
io_set_clock( )	Changes the **clock** setting for the specified object
io_set_direction( )	Changes the direction of I/O pins associated with any **bit**, **nibble**, or **byte** I/O objects

See the *Neuron C Reference Guide* for more information about these functions. The following sections describe the two most common functions and a common variable.

io_in( ) Function

When a program needs to retrieve signals from a peripheral device, declare an input object and use the built-in **io_in( )** function.

The syntax for the **io_in( )** function is:

return-value = **io_in (** *io-object-name* [**,** *args*] **)**

return-value

The current value read from the input device. The data type of the return value and its semantics are a function of the I/O model implemented by this I/O object.

io-object-name

The name for the I/O object, which corresponds to the *io-object-name* in the I/O object declaration.

args

Arguments that depend on the type of the I/O model. Some of these arguments can also appear in the I/O object declaration. If specified in both places, the value of the function argument overrides the declared value for the specific function call only. If the value is not specified in either the function argument or the declaration, the default value is used.

Example: The **io_in( )** function returns the value of the **ioProximity** proximity detector declared earlier:

```
detected = io_in(ioProximity);
```

See the *Neuron C Reference Guide* for object-specific rules that apply to this function.

io_out( ) Function

When a program needs to send signals to a peripheral device, declare an output object and use the built-in **io_out( )** function.

The syntax for the **io_out( )** function is:

io_out (*io-object-name***,** *output-value* [**,** *args*] **)**

io-object-name

The name for the I/O object, which corresponds to the *io-object-name* in the I/O object declaration.

output-value

The value that the function should set for the output device. The data type of the value and its semantics are a function of the I/O model implemented by this I/O object.

args

Arguments that depend on the type of the I/O model. Some of these arguments can also appear in the I/O object declaration. If specified in both places, the value of the function argument overrides the declared value for the specific function call only. If the value is not specified in either the function argument or the declaration, the default value is used.

Example 1: A lamp device could use the **io_out( )** function to turn the lamp on:

```
io_out(ioLamp, 0);
```

Example 2: A relay is attached to the IO0 pin (with appropriate driver circuitry). The declaration syntax for this simple device is:

```
#define ON 1
#define OFF 0

IO_0 output bit ioRelay;
// or IO_0 output bit ioRelay = ON;
```

The second (commented out) declaration in the example above uses an *initializer*, which tells the system that following a reset, the **ioRelay** object output value should initially be set to 1. The default initial value is 0.

Now you can control the state of **ioRelay** by using the **io_out( )** function:

```
if (flowTotal > 500) {
  io_out(ioRelay, ON);
}
```

The **io_out()** function takes a valid C expression for its argument. If the type of the expression matches the type of the output-value argument (which in turn is a function of the I/O model in use), you can also control the relay with direct logic:

```
io_out(ioRelay, flowTotal > 500);
```

input_is_new Variable

For all timer/counter input models, the built-in **input_is_new** variable is set to TRUE whenever the **io_in( )** call returns an updated value. This variable is also set for implicit calls (see *I/O Events* for information about implicit **io_in( )** calls). The data type of the **input_is_new** variable is an **unsigned short**. The frequency with which updates occur depends on the I/O model.

Note that the **input_is_new** variable is cleared after a related timer/counter interrupt executes; see the *Neuron C Programmer's Guide* for more information about timer/counter interrupts and their relation to I/O functions and I/O (timer/counter) events.

Example: This example uses one of the timer/counter I/O devices. Assume that the IO7 pin is attached to an optical flow meter that presents a number of pulses proportional to the volume of a fluid. The total volume in gallons needs to be determined. This example uses a Series 3100 Smart Transceiver with a 10 MHz input clock.

The **pulsecount** input model counts input edges and latches the count approximately every 0.8388608 (specifically, every $2^{23}/10^7$ seconds). If you were to use the **io_in( )** function for this I/O object, you would always read the *currently latched* value. If you are summing the total flow, you need to qualify this operation. Use the **input_is_new** variable, which is set to TRUE following an **io_in( )** function only if a *new* measurement is made, or in this case, every 0.8388608 seconds.

```
IO_7 input pulsecount ioFlowSensor;
// 451 pulses/gallon
long totalVolume, tempVolume;

...

{
  tempVolume = io_in(ioFlowSensor);
```

```
  if (input_is_new) {
    totalVolume += tempVolume;
  }
}

...
```

I/O Events

An alternative to using the explicit **io_in( )** function is to associate an input object with a predefined event or interrupt. The two I/O-related predefined events are:

- **io_changes**
- **io_update_occurs**

These events are used only with input objects. When they are evaluated, both the **io_update_occurs** and **io_changes** events perform an implicit **io_in( )** function call to obtain an input value for the object. Your program can access this input value by using the **input_value** variable.

You can also associate timer/counter I/O devices or individual I/O pins with application interrupts on a Series 5000 or Series 6000 device. See the *Neuron C Programmer's Guide* for more information about application interrupts.

io_changes Event

This event is TRUE when the value read from the specified input object changes state. The change can be one of three types:

- Any change (an unqualified change)
- A change (in absolute value) **by** a specified amount (or greater)
- A change **to** a specified value

The syntax for this event is:

io_changes(*io-object-name***)** [**by** *expr* | **to** *expr*]

The use of this event results in a comparison of the current value read from the input object with a reference value (except for the **to** option). The *reference value* is the value that was read the last time the change event evaluated to TRUE (and saved, at that time, by the firmware). For an **io_changes** event that does not use either the **by** option or the **to** option, a state change occurs when the current value is different from the reference value. When using the optional forms of the **io_changes** event, the *expr* expression does not need to be a constant. However, a constant expression is more efficient.

The **io_changes** event for a timer/counter input device occurs only if the device has a new value, different from the previous value. For the timer/counter devices, the **io_changes** event happens as listed in **Table 8**, depending on the input object type.

Table 8. io_changes Events for Specific I/O Models

I/O Model	Event
dualslope	Event occurs when the conversion is complete.
ontime	Event occurs if the measured time has changed from the last time.
period	Event occurs if the measured time has changed from the last time.
pulsecount	Event occurs if the number of counts measured has changed from the last count.
quadrature	Event occurs if the number of counts measured has changed from the last count.

Example: A program could use the **io_changes** event to detect changes in an **ioProximity** input object:

```
when (io_changes(ioProximity)) {
  . . .
}
```

If you were interested only in when the **io_part_detector** detected a part (a value of 1 in this example), you could use the following **when** clause:

```
when (io_changes(ioProximity) to 1) {
  . . .
}
```

io_update_occurs Event

The **io_update_occurs** event is TRUE when the value read from the input object specified by *io_object_name* has an updated value.

The syntax for this event is:

io_update_occurs (*io-object-name***)**

The **io_update_occurs** event applies only to certain timer/counter input models. Timing for the event depends on the input model, as listed in **Table 9**.

Table 9. io_update_occurs Events for Specific I/O Models

I/O Model	Event
dualslope	Event occurs when the conversion is complete, and the value has changed.
ontime	Event occurs at the end of the time being measured.
period	Event occurs at the end of the time being measured.
pulsecount	Event occurs every 0.8388608 seconds, when a new pulse count value is available.
quadrature	Event occurs as soon as at least one count is accumulated.

input_value Variable

You use the **input_value** variable to retrieve the input value for an I/O object when either the **io_update_occurs** event or the **io_changes** event occurs. The **input_value** built-in variable is a **signed long**, and it can be cast in the same manner as any other C variable.

Example:

```
when (io_update_occurs(ioDevice)) {
  if (input_value > 2) {
    . . .
  }
}
```

Example: A lamp device could set the value of its **nvoSwitch** network variable based on the value of **input_value** (the switch value):

```
when (io_changes(ioSwitchInput)) {
  nvoSwitch.state =
      (input_value == SWITCH_ON) ? ST_ON : ST_OFF;
}
```

The value of the **input_value** variable depends on the context in which it is used. The following combination of **when** clauses is valid. Because both events refer to the same I/O object, there is no ambiguity about which object is providing the input.

```
when (io_changes(ioDevice) to 4)
when (io_changes(ioDevice) to 3)
{
  x = input_value;
}
```

However, the following combination of **when** clauses is not a valid context for use of **input_value**, because there is no way to know which object is providing the input value. If the first **when** clause evaluated to TRUE, **input_value** would refer to **ioDevice1**, but if the second **when** clause evaluated to TRUE, **input_value** would refer to **ioDevice2**.

```
when (io_update_occurs(ioDevice1))
when (io_update_occurs(ioDevice2))
{
  x = input_value;    // from ioDevice1 or ioDevice2?
}
```

In addition, **input_value** is valid only after an **io_update_occurs** or **io_changes** event. In the following example, using multiple **when** clauses produces an ambiguous value for **input_value** because the **timer_expires** event does not perform I/O. In such cases, use the **io_in( )** function to retrieve the value.

```
when (timer_expires(t))
when (io_update_occurs(ioDevice))
{
  x = io_in(ioDevice);  // don't use input_value here
}
```

Using Functions or Events

To determine whether an input value is new, you can use the **io_in( )** function with the **input_is_new** variable or you can use the **io_update_occurs** event with the **input_value** variable. Which method you choose depends on the specific I/O model and the specific task that the program is designed to perform.

The I/O event mechanism tends to be the simpler method, where the Neuron scheduler decides when to perform the I/O functions. However, when you are combining multiple events in a single block of logic, you might need to call the **io_in( )** function explicitly, combined with the **input_is_new** variable.

The two examples shown in **Table 10** demonstrate different ways to accomplish the same goal.

Table 10. Comparing io_update_occurs and io_in()

io_update_occurs with **input_value**

```
IO_5 input pulsecount ioPulsecount;

when (io_update_occurs(ioPulsecount)) {
  if (input_value > 2) {
    . . .
  }
}
```

io_in( ) with **input_is_new**

```
stimer delayTimer;

IO_5 input pulsecount ioPulsecount;

when (timer_expires(delayTimer)) {
  . . .
  if ((io_in(ioPulsecount) > 2) && input_is_new) {
    . . .
  }
}
```

Important: If you combine explicit calls to the **io_in( )** function with **when** clauses that contain I/O events, synchronization problems can result. For example, if a **when** clause evaluates to TRUE near the end of an I/O sampling period, the **io_in( )** call might not be executed until the following period, and the value obtained could be misleading.

```
when (io_update_occurs(ioPulsecount)) {
  . . .
  z = input_value;  // don't use io_in(ioPulsecount) here
  . . .
}
```

I/O Functions for Timer/Counter Objects

For multiplexed I/O objects, the last timer/counter I/O object declared in the program is the first to take effect after a reset. To change the selected I/O object, use the **io_select( )** function to specify which of the multiplexed pins is the owner of the timer/counter circuit.

The syntax for the **io_select( )** function is:

io_select (*io-object-name* [**,** *clock*] **)**

io-object-name

The name for the I/O object, which corresponds to the *io-object-name* in the I/O object declaration.

clock

Specifies a clock selector, which can be different from or the same as the clock selector specified in the object's declaration, in the range of 0 to 7. If you do not specify a *clock* value in the call to the **io_select( )** function, the *clock* value is set to the value in the I/O object's declaration.

Any timer/counter I/O object that has a *clock* argument in its declaration syntax can also be reprogrammed to an alternate clock value by using the **io_set_clock( )** function.

The syntax for the **io_set_clock( )** function is:

io_set_clock (*io-object-name*, *clock* **)**

io-object-name

The name for the I/O object, which corresponds to the *io-object-name* in the I/O object declaration.

clock

Required clock selector value in the range of 0 to 7 (for Series 3100 devices) or 0 to 15 (for Series 5000 and Series 6000 devices), regardless of the clock selector specified in the object's declaration. Some I/O objects might not function properly with all clock values. See the description for a particular I/O object in Chapter 5, *Timer/Counter Input* and Chapter 6, *Timer/Counter Output*

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of how the **io_set_clock( )** function affects the resolution and range of certain timer/counter I/O models.

When **io_set_clock( )** is used on multiplexed objects, the clock is changed regardless of whether the object itself is currently selected.

Example: The following code fragment shows several examples of the use of **io_select( )** and **io_set_clock( )**:

```
IO_1 output pulsecount clock(3) outPulsecount;
IO_5 input period clock(2) inPeriod;
IO_6 input ontime clock(3) inOntime;

when (reset) {
  io_set_clock(outPulsecount, 5);
  io_select(inOntime);
```

```
}

when (io_update_occurs(inOntime)) {
  io_select(inPeriod, 3);
}
```

When a new clock is set for an I/O object using the **io_select( )** function, this clock remains in effect until a new value is explicitly set. The next **io_select( )** call for the same I/O object resets the clock to the value specified in the I/O object declaration if there is no clock argument in the **io_select( )** call. If your application specifies an alternate clock value, it must call the **io_set_clock( )** function within the reset task and after each call to the **io_select()** function.

If an input measurement is attempted using **io_in( )** or a **when** clause on an I/O object that has not been selected with the **io_select( )** function, a data value of *overrange* (65535) is returned, and the **input_is_new** variable and **io_update_occurs** event remain FALSE.

Following a call to the **io_select( )** function and after resetting the Neuron Chip or Smart Transceiver, the first measurement taken for the newly selected I/O object is discarded to clear out any incomplete measurements (unless the function **io_preserve_input( )** is called before the **io_in( )** call). The **io_update_occurs** event actually occurs when the second measurement is read. Rely on either an **io_update_occurs** event or use the **input_is_new** variable to verify that an actual measurement has been made following a call to **io_select( )**.

Example 1: The following example shows the use of the **io_select( )** function with the multiplexed timer/counter circuit. For multiplexed I/O objects, the last I/O object declared in the program is the first to take effect after a reset.

```
// I/O Definitions
IO_5 input period mux clock (2) ioPeriod2;
IO_4 input period mux clock (2) ioPeriod1;

static long variable1, variable2;

// The following occurs only when ioPeriod1 is selected
when (io_update_occurs(ioPeriod1)) {
  variable1 = input_value;
  // select next I/O object
  io_select(ioPeriod2);
}

// The following occurs only when ioPeriod2 is selected
when (io_update_occurs(ioPeriod2)) {
  variable2 = input_value;
  // select next I/O object
  io_select(ioPeriod1);
}
```

Example 2: In the following example, the timer/counter is multiplexed between an **ontime** measurement on pin IO5 and a **period** measurement on pin IO6. Because the **ontime** input can cover a large range of values, this example uses a form of "auto-ranging." The clock value switches between 4 and 2 if the input measurement value extends beyond certain values. A variable is used when reselecting the **ontime** object because its clock can be one of the two values.

```
unsigned long slope1Raw, cycleAValue;
```

```
int slope1Clock = 2;
IO_5 input ontime clock (2) ioSlope1;
IO_6 input period clock (1) ioCycleA;
// Following reset, the ioCycleA object is selected
// because it is the last object declared using the mux

when (io_update_occurs(ioSlope1)) {
  if (input_value > 0x4000 && slope1Clock == 2) {
    // Range down (slower)
    slope1Clock = 4;
    io_set_clock(ioSlope1, slope1Clock);
  } else if (input_value < 0x4000 && slope1Clock == 4) {
    // Range up (faster)
    slope1Clock = 2;
    io_set_clock(ioSlope1, slope1Clock);
  } else {
    // Save the measured value, select the other object
    slope1Raw = input_value;
    io_select(ioCycleA);
  }
  // If auto-ranging has occurred, another measurement
  // will be made. Otherwise, the ioCycleA object
  // will be measured next.
}

when (io_update_occurs(ioCycleA)) {
  cycleAValue = input_value;
  // Now select the ioSlope1 object,
  // using the current clock range computed above
  io_select(ioSlope1, slope1Clock);
}
```

I/O Measurements, Outputs, and Functions

This section describes when the I/O pins are sampled or set, depending on the I/O model.

Direct, Serial, and Parallel I/O Models

For direct I/O models, input levels are sampled when an **io_in( )** function is called, or when a **when** clause that references the object is evaluated.

For **serial** and **parallel** I/O models, input levels are sampled when the **io_in( )** function is called. For a Series 3100 device with a 40 MHz input clock, output levels are set approximately 12.5 to 25 microseconds after invocation of the **io_out( )** function. This timing value scales with the input clock speed. See the specific I/O model for detailed timing diagrams.

Timer/Counter I/O Models

Values for timer/counter input models are latched periodically depending on the model or the I/O object clock. The relationship between when an **io_in( )** function or an I/O **when** clause is used and when the data has been latched is usually application dependent. After a value is latched, that value continues to

be returned by subsequent calls to **io_in( )** until a new value is latched based on the timing in the hardware.

The **period** input and **ontime** input models latch a new value on the falling edge of the input signal (or if the **invert** keyword is used, these models latch the new value on the rising edge of the input signal). The **pulsecount** input model latches a new value every 0.8388608 seconds. See the *Neuron C Programmer's Guide* for more information about timer frequencies and timer accuracy.

Generally, new values written to timer/counter output objects are acted upon at the end of the current output signal period. Exceptions to this rule are **oneshot** output and I/O models that have been disabled (that is, have a zero control value), all of which take effect upon return from the **io_out( )** function.

Output Models

The following timer/counter output models reflect a new output value at the end of the current output signal period:

- **edgedivide** output
- **frequency** output
- **pulsewidth** output
- **stretchedtriac** output
- **triac** output
- **triggeredcount** output

The following timer/counter output models reflect a new output value upon return from the **io_out( )** function:

- **oneshot** output
- **pulsecount** output

All timer/counter output models respond to a zero output value upon return from the **io_out( )** function.

2

Direct I/O Models

This chapter describes direct input/output models. Direct I/O models are based on a logic level at the I/O pins; none of the Neuron Chip or Smart Transceiver hardware's timer/counters are used in conjunction with these I/O objects. These models can be used in multiple, overlapping combinations within the same Neuron Chip or Smart Transceiver.

Bit Input/Output

The bit I/O model is used to read or control the logical state of a single pin, where 0 represents low and 1 represents high.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Pins IO0 – IO11 can be individually configured as single-bit input or output ports. Inputs can be used to sense transistor-transistor logic (TTL)-level compatible logic signals from external logic, contact closures, and so on. Outputs can be used to drive external CMOS and TTL-level compatible logic, switch transistors, and very low current relays to actuate higher-current external devices such as stepper motors and lights.

For Series 3100 devices, the high (20 mA) current sink capability of pins IO0 – IO3 (see **Figure 7**) allows these pins to drive many I/O devices directly.

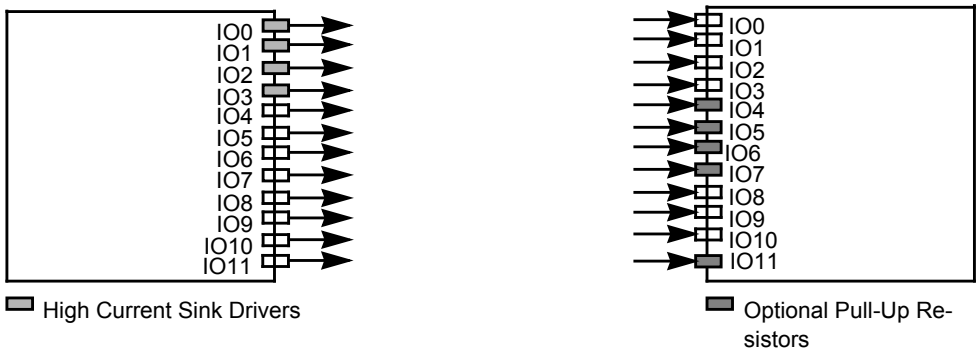


Figure 7. Bit I/O for Series 3100 Devices

Notes:

- After a reset, a Series 3100 power line device disables the IO4-IO7 and IO11 pull-up resistors. The pull-up resistors are not turned on until application initialization. Pull-ups are only enabled when specified in the application configuration using the **#pragma enable_io_pullups** Neuron C directive.
- After a reset, a Series 3100 FT device performs a self test, which includes enabling the IO4-IO7 pull-up resistors. Enabling the pull-up resistors could cause a positive transition on the pins.

Figure 8 and **Figure 9** show the bit input and bit output latency times, respectively. These are the times from the call to the **io_in()** or **io_out()** function, until a value is returned. The direction of bit ports can be changed between input and output dynamically by using the **io_set_direction()** function.

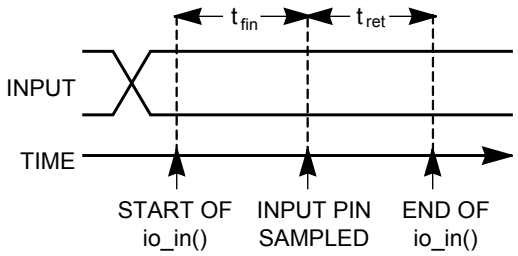


Figure 8. Bit Input Timing

Table 11. Bit Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to sample IO0 – IO10 IO11	 41 µs 8.4 µs
t_{ret}	Return from function IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7 IO8 IO9 IO10 IO11	 19 µs 23.4 µs 27.9 µs 32.3 µs 36.7 µs 41.2 µs 45.6 µs 50 µs 19 µs 23.4 µs 27.9 µs 7.8 µs

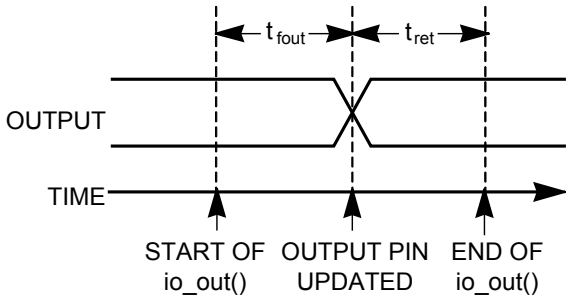


Figure 9. Bit Output Timing

Table 12. Bit Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to update IO3 – IO5, IO11 All others	 69 µs 60 µs
t_{ret}	Return from function IO0 – IO11	 5 µs

Programming Considerations

For bit input, the data type of the return value for **io_in( )** is an **unsigned short**. For bit output, the output value is treated as a Boolean value, so any non-zero value is treated as a 1.

The bit input and output models are both direct I/O models. Thus, bit input objects are sampled at the time of the **io_in()** call, and bit output objects are driven at the time of the **io_out()** call.

Although this I/O model is suitable for many simple use-cases, such as driving an LED or a single relay, many control applications require a synchronized reading and writing of various bit input and output devices. Applications that require a synchronized process image should consider using the byte or nibble models instead.

For Series 3100 devices, add a **#pragma enable_io_pullups** directive to enable the Neuron Chip or Smart Transceiver's built-in pull-up resistors on pins IO4 through IO7 and IO11.

IO11 is only available on PL 3120, PL 3150, PL 3170, Series 5000, and Series 6000 chips.

Syntax

pin **input bit** *io-object-name*;

pin **output bit** *io-object-name* [=*initial-output-level*];

pin

Specifies one of the twelve I/O pins, **IO_0** through **IO_11**. Bit input/output can be used on any pin.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

unsigned int *input-value***;**
unsigned int *output-value***;**

input-value **= io_in(***io-object-name***);**
io_out(*io-object-name***,** *output-value***);**

Bit Input Example

```
IO_1 input bit ioSwitch; // declares pin IO1 as a
// bit input object named ioSwitch

unsigned int switch_on_off;
...

when (reset) {
  io_change_init(ioSwitch);
}

when (io_changes(ioSwitch)) {
  switch_on_off = input_value;
}
```

Bit Output Example

```
IO_2 output bit ioLed;
unsigned int led_on_off;
...

when(...) {
  io_out(ioLed, led_on_off);
}
```

Byte Input/Output

The byte I/O model is used to read or control eight pins simultaneously.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers and to Series 6000 Neuron Processors and Smart Transceiver.

Hardware Considerations

Pins IO0 – IO7 can be configured as a byte-wide input or output port, which can be read or written using integers in the range 0 to 255. This is useful for reading or writing a synchronized process image, where multiple binary outputs are assigned (or sampled) simultaneously. Other uses include driving devices that require ASCII data, or other data, eight bits at a time. For example, an alphanumeric display panel can use byte function for data, and use pins IO8 – IO11 in bit function for control and addressing.

For Series 3100 devices, the high (20 mA) current sink capability of pins IO0 – IO3 (see **Figure 10**) allows these pins to drive many I/O devices directly.

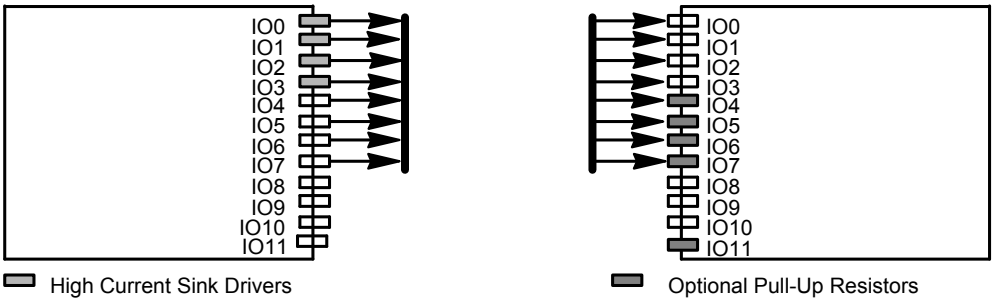


Figure 10. Byte I/O for Series 3100 Devices

Figure 11 and **Figure 12** show the byte input and byte output latency times, respectively. These are the times from the call to the **io_in()** or **io_out()** function, until a value is returned. The direction of bit ports can be changed between input and output dynamically by using the **io_set_direction()** function.

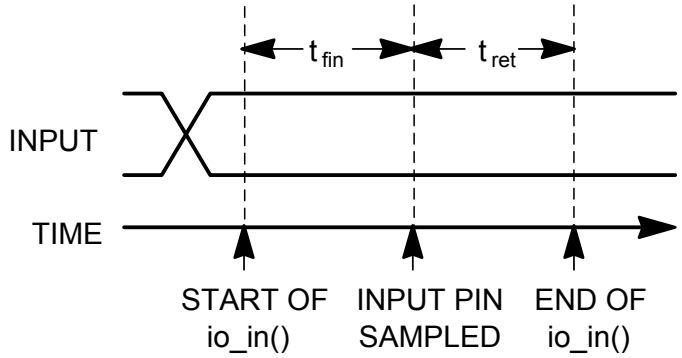


Figure 11. Byte Input Timing

Table 13. Byte Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to sample	24 µs
t_{ret}	Return from function	4 µs

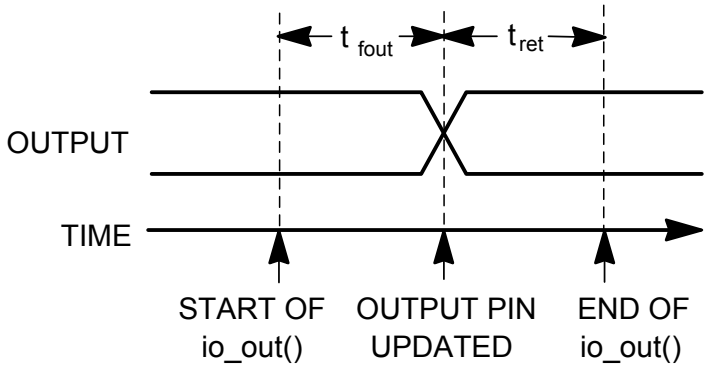


Figure 12. Byte Output Timing

Table 14. Byte Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to update	57 µs
t_{ret}	Return from function	5 µs

Programming Considerations

For byte input/output, the data type of the return value for **io_in( )**, and the data type of the output value for **io_out( )**, is an **unsigned short**.

Syntax

IO_0 input byte *io-object-name***;**

IO_0 output byte *io-object-name* [**=***initial-output-level*]**;**

IO_0

Specifies pin **IO_0** as the least significant bit of the byte. Byte input/output uses pins **IO_0** through **IO_7**. The pin specification denotes the lowest numbered pin of the set and must be **IO_0**.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be from 0 to 255. The default is 0.

Usage

unsigned int *input-value***;**
unsigned int *output-value***;**

input-value = **io_in(***io-object-name***);**
io_out(*io-object-name*, *output-value***);**

Byte Input Example

```
IO_0 input byte ioKeyboard;
unsigned int character;
...

when (reset) {
  io_change_init(ioKeyboard);
```

```
}

when (io_changes(ioKeyboard)) {
  character = input_value;
}
```

Byte Output Example

```
IO_0 output byte ioDisplay;
...

when (...) {
  io_out(ioDisplay, '?');
}
```

Leveldetect Input

The leveldetect I/O model is used to detect a low level (logical zero) on a single pin, for example, for a proximity detector.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transcievers.

Hardware Considerations

Pins IO0 – IO7 can be individually configured as leveldetect input pins, which latch a negative-going transition of the input level with a minimal low pulse width of 200 ns for a Series 3100 Smart Transceiver with a 10 MHz input clock, or a minimal low pulse width of 12.5 ns for a Series 5000 or Series 6000 Smart Transceiver with an 80 MHz system clock. The application can therefore detect short pulses on the input which might be missed by software polling. This detection is useful for reading devices, such as proximity sensors.

Important: This is the only direct I/O model that is latched before it is sampled.

The latch is cleared during the **when** statement sampling, and can be set again immediately after, if another transition should occur (see **Figure 13**).

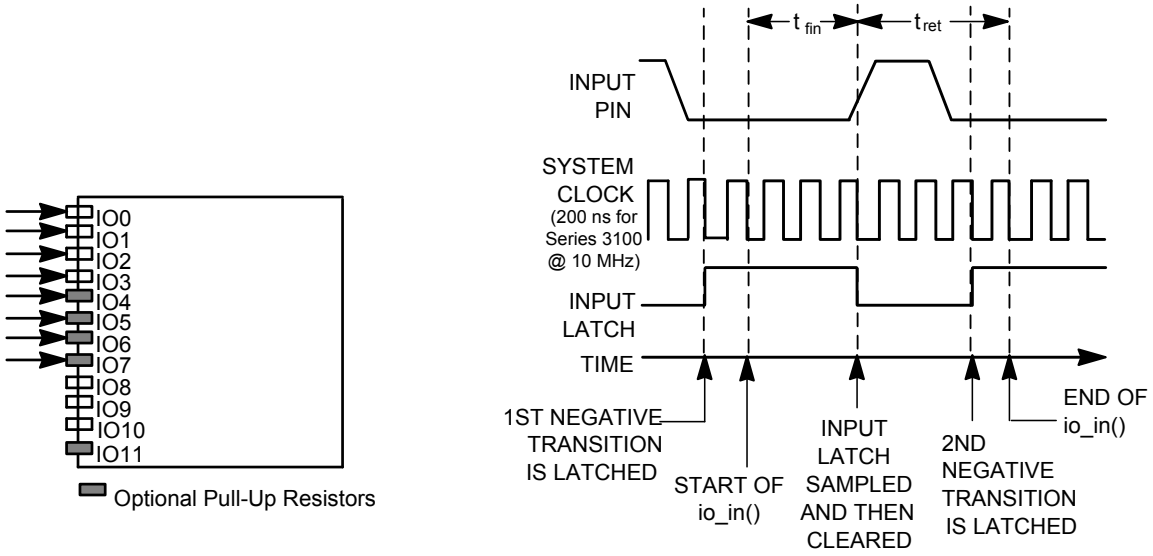


Figure 13. Leveldetect for Series 3100 Devices and Input Timing

Table 15. Leveldetect Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to sample IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7	 35 µs 39.4 µs 43.9 µs 48.3 µs 52.7 µs 57.2 µs 61.6 µs 66 µs
t_{ret}	Return from function	32 µs

Programming Considerations

The state of the input is latched in hardware every 50 ns for a Series 3100 device with a 40 MHz input clock or every 12.5 ns for a Series 5000 device with an 80 MHz system clock (the interval scales with clock speed), capturing any low level input. This event is represented by a **TRUE** (1) value returned from the **io_in( )** call, and the value is then cleared to 0 when read. However, as long as the input pin level stays at logical zero (0), each **io_in( )** call returns a 1 value.

The leveldetect input model is useful for capturing events of short duration that would otherwise be missed by the bit input model. For leveldetect input, the data type of **return_value** for **io_in( )** is an **unsigned short**.

For Series 5000 and Series 6000 chips, I/O interrupts are also available to implement low-latency application-specific response to the I/O pins. Because I/O

interrupts can be used independently from I/O objects, and can be triggered by positive or negative level, rising or falling edge, or either edge of the I/O signal (regardless of the I/O object's direction), I/O interrupts can often be used in place of a **leveldetect** object.

For Series 3100 devices, add a **#pragma enable_io_pullups** directive to enable the Neuron Chip's or Smart Transceiver's built-in pull-up resistors on pins **IO_4** through **IO_7**.

Syntax

pin **[input] leveldetect** *io-object-name***;**

pin

An I/O pin. Leveldetect input can specify one of the pins **IO_0** through **IO_7**.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *input-value***;**

input-value **= io_in(***io-object-name***);**

Example

```
IO_6 input leveldetect ioGrounded;

when (io_changes(ioGrounded) to TRUE) {
  ...
  // this task runs when I/0 reaches logical 0 level
}
```

Nibble Input/Output

The nibble I/O model is used to read or control four adjacent pins simultaneously.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to the Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Groups of four consecutive pins between IO0 – IO7 can be configured as nibble-wide (4-bit) input or output ports, which can be read or written to using integers in the range 0 to 15. This model is useful for reading or writing a synchronized process image, where multiple binary outputs are assigned (or sampled) simultaneously. Other uses include driving devices that require binary-coded decimal (BCD) data, or other data four bits at a time. For example, a 4x4 key switch matrix can be scanned by using one nibble to generate an output (row

select — one of four rows), and one nibble to read the input from the columns of the switch matrix.

The direction of nibble ports can be changed between input and output dynamically under application control (see *Programming Considerations*). The least-significant bit (LSB) of the input data is determined by the object declaration and can be any of the IO0 – IO4 pins.

For Series 3100 devices, the high (20 mA) current sink capability of pins IO0 – IO3 (see **Figure 14**) allows these pins to drive many I/O devices directly.

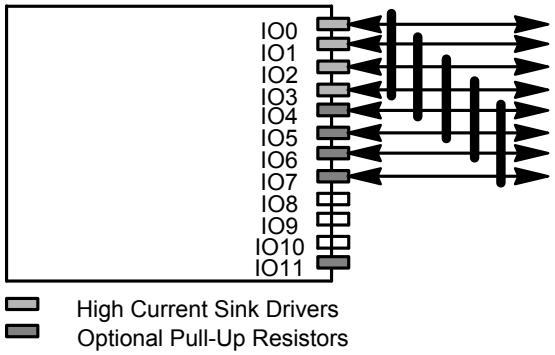


Figure 14. Nibble Input/Ouput

Figure 15 and **Figure 16** show the nibble input and nibble output latency times, respectively. These are the times from the call to the **io_in()** or **io_out()** function, until a value is returned. The direction of bit ports can be changed between input and output dynamically by using the **io_set_direction()** function.

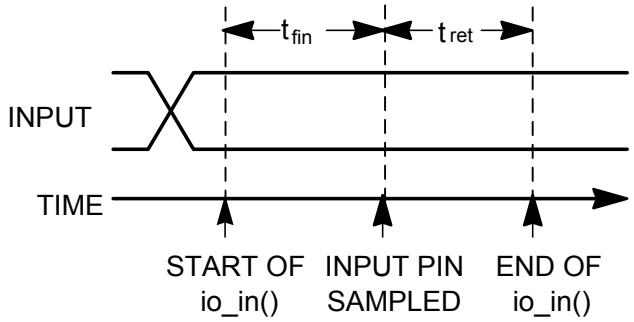


Figure 15. Nibble Input Timing

Table 16. Nibble Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to sample IO0 – IO4	41 µs

Symbol	Description	Typical at 10 MHz
t_{ret}	Return from function IO0 IO1 IO2 IO3 IO4	 18 µs 22.8 µs 27.5 µs 32.3 µs 36 µs

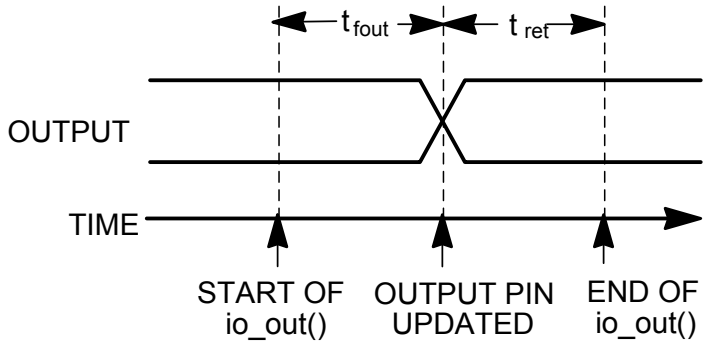


Figure 16. Nibble Output Timing

Table 17. Nibble Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to update IO0 IO1 IO2 IO3 IO4	 78 µs 89.8 µs 101.5 µs 113.5 µs 125 µs
t_{ret}	Return from function IO0 – IO4	 5 µs

Programming Considerations

For **nibble** input/output, the data type of *return_value* for the **io_in( )** function, and the data type of the output value for the **io_out( )** function is an **unsigned short**.

For Series 3100 devices, add a **#pragma enable_io_pullups** directive to enable the Neuron Chip's or Smart Transceiver's built-in pull-up resistors on pins **IO_4** through **IO_7**.

Syntax

pin **input nibble** *io-object-name***;**

pin **output nibble** *io-object-name* [**=** *initial-output-level*]**;**

pin

An I/O pin. Nibble input/output requires four adjacent pins. The pin specification denotes the lowest numbered pin of the set and can be **IO_0** through **IO_4**. The lowest numbered I/O pin is defined as the least significant bit of the nibble data.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be from 0 to 15. The default is 0.

Usage

unsigned int *input-value***;**
unsigned int *output-value***;**

input-value = **io_in(***io-object-name***);**
io_out(*io-object-name*, *output-value***);**

Nibble Input Example

```
IO_0 input nibble ioColumnRead;
unsigned column;

when (reset) {
  io_change_init(ioColumnRead);
}

when (io_changes(ioColumnRead)) {
  column = input_value;
}
```

Nibble Output Example

```
IO_4 output nibble ioRowWrite;

when (...) {
  io_out(ioRowWrite, 0b1000U);
}
```

Touch Input/Output

The touch I/O model is used to interface to any peripheral device that implements the 1-Wire® protocol developed by Dallas Semiconductor Corporation (now Maxim Integrated Products). This protocol provides communications with Touch Memory devices, iButton™ devices, and other similar devices. This protocol uses a one-wire, open-drain, bidirectional connection.

The touch I/O model operates only within the timing specifications set forth by Dallas Semiconductor Corporation for the 1-Wire protocol. This interface supports bi-directional data transfers across a signal and ground wire pair. An external pull-up is required, and the interface is connected directly to the designated I/O pin. This I/O pin is operated as an open-drain device in order to support the interface.

Up to 255 bytes of data can be transferred at a time.

For more information about this protocol, and the devices that it supports, see application note 937, *Book of iButton Standards*, from Maxim Integrated Products.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to the Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Up to eight 1-Wire memory busses can be connected to a Smart Transceiver through the use of the first eight I/O pins, IO0 – IO7. The only additional component required for this is a pull-up resistor on the data line (refer to the 1-Wire Memory specification below on how to select the value of the pull-up resistor). The high current sink capabilities of IO0 – IO3 pins of a Series 3100 Smart Transceiver can be used in applications where long wire lengths are required between the 1-Wire device and the Smart Transceiver.

The slave acquires all necessary power for its operation from the data line. Upon physical connection of a 1-Wire device to a master (in this case the Smart Transceiver), the 1-Wire Memory generates a low presence pulse to inform the master that it is awaiting a command. The Smart Transceiver can also request a presence pulse by sending a reset pulse to the 1-Wire device.

Commands and data are sent bit by bit to make bytes, starting with the least-significant bit (LSB). The synchronization between the Smart Transceiver and the 1-Wire devices is accomplished through a negative-going pulse generated by the Smart Transceiver.

Figure 17 shows the details of the reset pulse in addition to the read/write bit slots.

Note: NodeBuilder 3.1 features the ability to adjust the t_{low}, t_{wrd}, and t_{rdi} timing values.

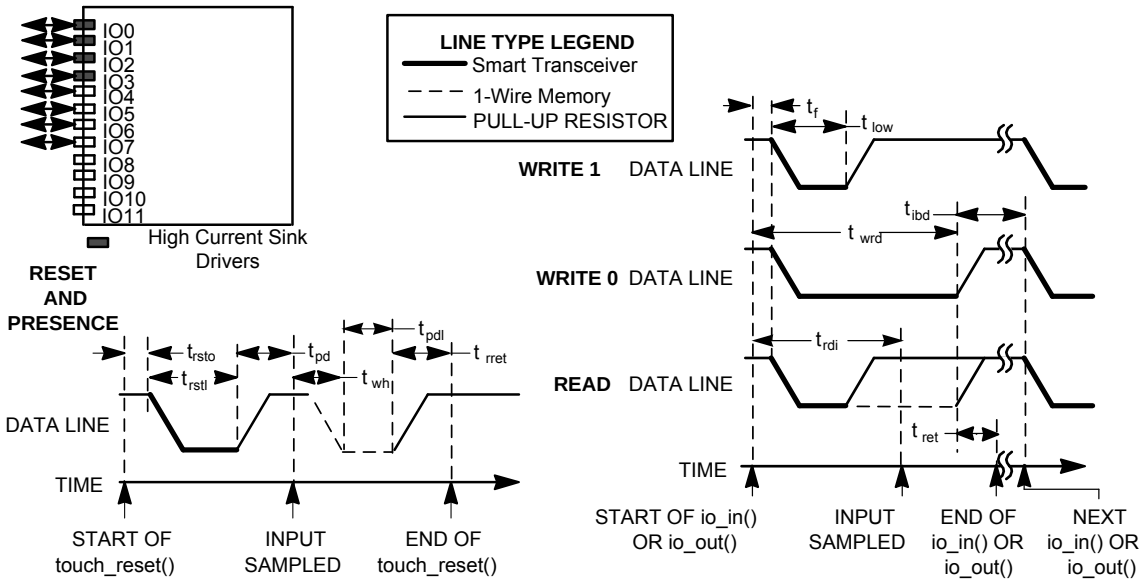


Figure 17. Touch I/O for Series 3100 Devices and Timing

Table 18. Touch I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{rsto}	Reset call to data line low	—	60.0 µs	—
t_{rstl}	Reset pulse width	—	500 µs	—
t_{pdh}	Reset pulse release to data line high 10 MHz 5 MHz	 4.8 µs 9.6 µs	—	 275 µs 275 µs
t_{pdl}	Presence pulse width	—	120.0 µs	—
t_{wh}	Data line high detect to presence pulse	—	80 µs	—
t_{rret}	Return from reset function	—	12.6 µs	—
t_f	I/O call to data line low (start of bit slot)	—	125.4 µs	—
t_{low}	Start pulse width 10 MHz 5 MHz	—	 4.2 µs 8.4 µs	—
t_{rdi}	Start pulse edge to sampling of input (read operation) 10 MHz 5 MHz	—	 15.0 µs 18.0 µs	—

Symbol	Description	Minimum	Typical	Maximum
t_{wrd}	Start pulse edge to Smart Transceiver releasing the data line 10 MHz 5 MHz	—	 66.6 µs 72.0 µs	—
t_{ibd}	Inter-bit delay 10 MHz 5 MHz	—	 61.2 µs 122.4 µs	—
t_{ret}	Return from I/O call	—	42.6 µs	—

The leveldetect input model can be used for detection of asynchronous attachments of 1-Wire devices to the Smart Transceiver. In such a case, the leveldetect input object is overlaid on top of the Touch I/O object. See *Overlaying I/O Objects* for information about I/O object overlays.

Programming Considerations

The **touch** I/O model is used to interface to the 1-Wire protocol, and allows up to 255 bytes of data can be transferred at a time.

Syntax

pin **touch** [**output_pin(***pin***)**] [**timing(***t-low***, *t-rdi*, *t-wrd***)**] *io-object-name***;**

pin

An I/O pin. Touch I/O can specify one of the pins **IO_0** through **IO_7**. Multiple Touch I/O objects can be declared. If you do not explicitly declare a separate output pin with the **output_pin( )** parameter, this pin specifies both the input and output pin. Otherwise, it specifies only the input pin.

output_pin(*pin***)**

Optionally specifies the output pin. If not specified, the output pin is the same as the input pin.

timing(…)

Optionally specifies three timing parameters. There are three time periods associated with each bit time slot. All values here apply to a Series 3100 device with a 10 MHz input clock (and *double* for a 5 MHz input clock). Because these timing controls affect the low-level single bit function used by both read and write operations, they are required for both 1-wire read and write operations. A value of 0 for a timing control is the same as a value of 256.

You can optionally specify the following three timing parameters when you declare the **touch** I/O object:

- *t-low*
  The length of t_{LOW}. This is the interval where the Neuron firmware asserts a low on the 1-Wire bus signaling the start of the bit slot.

This argument has a minimum value of 7.2 µs (*t_low* = 1) from the start of t_{LOW}. The incremental resolution of *t_low* is 3 µs, so the control range is 4.2 + n * 3 (in µs) where n is 1 to 255, and a *t-low* value of 0 is equivalent to n=256.

- *t-rdi*
  The length of t_{RDI}. This is the interval where the Neuron firmware asserts either a low or a high on the 1-Wire bus, depending on the output data bit polarity. For read operations, this data polarity is always high. This argument has a minimum value of 7.8 µs (*t-rdi* = 1) from the start of t_{RDI}. The incremental resolution of *t-rdi* is 3 µs, so the control range is 4.8 + n * 3 (in µs) where n is 1 to 255, and a *t-rdi* value of 0 is equivalent to n=256.
- *t-wrd*
  Start of t_{WRD} (end of t_{RDI}). This is the point where the Neuron firmware samples the 1-Wire bus for the input data bit, and occurs for both read and write operations. This argument has a minimum value of 15 µs (*t-wrd* = 1) from the start of t_{WRD}. The incremental resolution of *t-wrd* is 3 µs, so the control range is 12 + n * 3 (in µs) where n is 1 to 255, and a *t-wrd* value of 0 is equivalent to n=256.

At the end of *t-wrd*, the Neuron firmware releases the 1-Wire bus.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***;**

unsigned int *touch-buffer***[***buffer-size***];**

io_out(*io-object-name***,** *touch-buffer***,** *count***);**
io_in(*io-object-name***,** *touch-buffer***,** *count***);**

The *touch-buffer* can be any type or structure, and the type of the *touch-buffer* parameter is `const void*`. The address of the buffer is passed to the **io_out( )** and **io_in( )** functions. There are several additional support functions for the Touch I/O object: **touch_reset( )**, **touch_byte( )**, **touch_bit( )**, **touch_first( )**, **touch_next( )**, **crc8( )**, and **crc16( )**.

For Neuron Chips and Smart Transceivers that include system firmware version 18 or later, there are other additional support functions for the Touch I/O model: **touch_reset_spu( )**, **touch_byte_spu( )**, **touch_read_spu( )**, **touch_write_spu( )**, and **crc16_ccitt()**.

int touch_reset(*io-object-name***);**

The **touch_reset( )** function asserts the reset pulse. The function returns a 1 value if a presence pulse was detected, or a 0 value if no presence pulse was detected, or a -1 value if the 1-Wire bus appears to be stuck low. The operation of this function is controlled by several timing constants:

- The reset pulse period, 500 µs.
- After the reset pulse period, the Neuron firmware releases the 1-Wire bus and waits for the 1-Wire bus to return to the high state. This period is

limited to 275 µs, after which the **touch_reset( )** function returns a -1 value with the assumption that the 1-Wire bus is stuck low. There also is a minimum value for this period: for a Series 3100 device, it must be >4.8 µs @10 MHz, or >9.6 µs @5 MHz; for a Series 5000 or Series 6000 device, it must be >0.3 µs @ 80 MHz, or >4.8 µs @5 MHz.

After the 1-Wire bus has appeared to go high, the Neuron firmware waits for the presence pulse for a period up to 80 µs. If a low input level is not sensed within this period, the function returns a 0 value. When a presence pulse is detected, the Neuron firmware then waits for the end of the presence pulse by waiting for a high level on the bus. This period is limited to 250 µs, after which the function again returns a -1 if the period elapses with the input level still low. Otherwise, after the input level is high again, the function returns with a 1 value. The **touch_reset( )** function does not return until the end of the presence pulse has been detected.

unsigned touch_byte(*io-object-name***, unsigned** *write-data***);**

The **touch_byte( )** function sequentially writes and reads eight bits of data on the 1-Wire bus. It can be used for either reading or writing. For reading, the *write-data* argument should be all ones (0xFF), and the return value contains the eight bits as read from the bus. For writing, the bits in the *write-data* argument are placed on the 1-Wire bus, and the return value normally contains those same bits.

This function allows combined read and write operations within a single 8-bit boundary. For example, a 2-bit write can be followed by a 6-bit read. This read can be accomplished with a single call to the **touch_byte( )** function with a *write-data* argument of 0b*NN*111111 where *NN* represents the two bits of write data and (111111) is used to perform the 6-bit read.

unsigned touch_bit(*io-object-name***, unsigned** *write-data***);**

The **touch_bit( )** function writes and reads a single bit of data on the 1-Wire bus. It can be used for either reading or writing. For reading, the *write-data* argument should be one (0x01), and the return value contains the bit as read from the bus. For writing, the bit value in the *write-data* argument is placed on the 1-Wire bus, and the return value normally contains that same bit value, and can be ignored.

int touch_first(*io-object-name***, search_data *** *sd***);**
int touch_next(*io-object-name***, search_data *** *sd***);**

These functions execute the Search ROM command, as described in the *Book of iButton Standards*. Both functions use a **search_data_s** data structure for intermediate storage of a bit marker and the current ROM data:

```
typedef struct search_data_s {
  int search_done;
  int last_discrepancy;
  unsigned rom_data[8];
} search_data;
```

This data structure is automatically defined in Neuron C, regardless of whether a program references the touch I/O functions.

A return value of **TRUE** indicates whether a device was found, and if so, that the data stored at **rom_data[ ]** is valid. A **FALSE** return value indicates no device found. The **search_done** flag is set to **TRUE** when there are no more devices on

the 1-Wire bus. The **last_discrepancy** variable is used internally and should not be modified.

To start a new search:

1. Call **touch_first( )**
2. As long as the **search_done** flag is not set, call **touch_next( )** as many times as are required.

For a Series 3100 device, each call to **touch_first( )** or **touch_next( )** takes 41 ms to execute at 10 MHz (63 ms at 5 MHz) when a device is being read. For a Series 5000 or Series 6000 device, each call to **touch_first( )** or **touch_next( )** takes 14 ms to execute at 80 MHz (29 ms at 10 MHz) when a device is being read.

unsigned crc8(unsigned *crc***, unsigned** *new-data***);**

This function performs the Dallas 1-Wire 8-bit cyclic redundancy check (CRC) function on the *crc* and *new-data* arguments, and returns the new 8-bit CRC value. You must include **<stdlib.h>** in your program to use this function.

unsigned long crc16(unsigned long *crc***, unsigned** *new-data***);**

This function performs the Dallas 1-Wire 16-bit CRC function on the *crc* and *new-data* arguments, and returns the new 16-bit CRC value. You must include **<stdlib.h>** in your program to use this function.

Certain 1-Wire devices, such as the Maxim Integrated Products DS18S20 High-Precision 1-Wire Digital Thermometer, require that the 1-Wire bus be actively held high during certain operations. These devices require more current than a typical external pull-up resistor can provide for device operations. The following functions (named touch_*xxx*_spu) drive the Neuron Chip or Smart Transceiver output to an active high state when the pin is idle. These functions require system firmware version 18 or later.

int touch_reset_spu(unsigned *pinmask***);**

The **touch_reset_spu( )** function asserts the reset pulse, just as the **touch_reset( )** function does. The *pinmask* defines which pins are driven high when idle.

void touch_byte_spu(unsigned *pinmask***, unsigned** *data***);**

The **touch_byte_spu( )** function sequentially writes eight bits of data on the 1-Wire bus, just as the **touch_byte( )** function does. The *pinmask* defines which pins are driven high when idle. The *data* defines the read or write data.

void touch_read_spu(unsigned *pinmask***, unsigned ****dp***, unsigned** *count***);**

The **touch_read_spu( )** function reads a specified number of bits of data on the 1-Wire bus, similar to the **touch_bit( )** function. The *pinmask* defines which pins are driven high when idle. The *dp* pointer defines the buffer into which the function stores the read data. The *count* defines how many bits to read.

void touch_write_spu(unsigned *pinmask***, const unsigned ****dp***, unsigned** *count***);**

The **touch_write_spu( )** function writes a specified number of bits of data on the 1-Wire bus, similar to the **touch_bit( )** function. The *pinmask* defines which pins are driven high when idle. The *dp* pointer defines the buffer from which the function writes the data. The *count* defines how many bits to write.

Example

```
// In this example, a leveldetect input is used on the
// 1-Wire interface to detect the 'presence' signal
// when a Touch Memory device appears on the bus.

#include <stdlib.h>

#define DS_READ_ROM 0x33
unsigned int data[8];
IO_3 input leveldetect ioPresence;
IO_3 touch ioTouchWire;
. . .

when (io_in(ioPresence) == 1) {
  unsigned int i, crc;

  // Reset the device using touch_reset().
  // Skip if there is no device sensed.
  if (touch_reset(ioTouchWire)) {
    // Send a single READ_ROM command byte:
    id_data[0] = DS_READ_ROM;
    io_out(ioTouchWire, data, 1);

    // Read the 8 byte I.D.:
    io_in(ioTouchWire, data, 8);

    // check the crc of the I.D.:
    crc = 0;
    for (i=0; i<7; i++)
      crc = crc8(crc, data[i]);

    if (crc == id_data[7]) {
      // Valid crc: process I.D. data here.
    }
  }
  // Clear leveldetect input.
  (void)io_in(ioPresence);
}
```

3

Parallel I/O Models

This chapter describes parallel input/output models. Parallel I/O models are used for high-speed bidirectional I/O.

Muxbus Input/Output

The multiplexed bus (muxbus) I/O model provides a means of performing parallel I/O data transfers between a Smart Transceiver and an attached peripheral device or processor. This I/O model allows you to interface with any device that requires an address and a data bus, such as a programmable universal asynchronous receiver/transmitter (UART).

The **muxbus** I/O model uses eleven I/O pins to form an 8-bit address and bi-directional data bus interface. This I/O model uses pins **IO_0** through **IO_7** for the 8-bit address bus and the 8-bit data bus. Pins **IO_8** through **IO_10** are control signals that are always driven by the Neuron Chip or Smart Transceiver, as shown in **Table 19**.

Table 19. Muxbus Signals

Pin	Function
IO0 thru **IO7**	Address and bi-directional data
IO_8	C_ALS: Address latch strobe, asserted high
IO_9	C_WS~: Write strobe, asserted low
IO_10	C_RS~: Read strobe, asserted low

This I/O model provides the capability to build an 8-bit data bus system with an 8-bit address bus. Typically, an 8-bit D-type latch (such as a 74HC573) is connected to the Neuron I/O pins where pins **IO_0** through **IO_7** are connected to the eight Q inputs. Pin **IO_8** is connected to the Latch Enable input. In this configuration, eight bits of address are latched on the eight D output pins of the 74HC573 device.

Pins **IO_9** and **IO_10** are the write and read strobes, normally high.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to the Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Unlike the parallel input/output model, which uses a token-passing scheme for ensuring synchronization, the muxbus input/output enables a Smart Transceiver to essentially be in control of all read and write operations at all times. This control relieves the burden of protocol handling from the attached device and results in an easier-to-use interface at the expense of data throughput capacity. The data bus remains in the last state used.

Figure 18 shows the muxbus I/O latency times. These are the times from the call to the **io_in()** or **io_out()** function, until a value is returned. The direction of bit ports can be changed between input and output dynamically by using the **io_set_direction()** function.

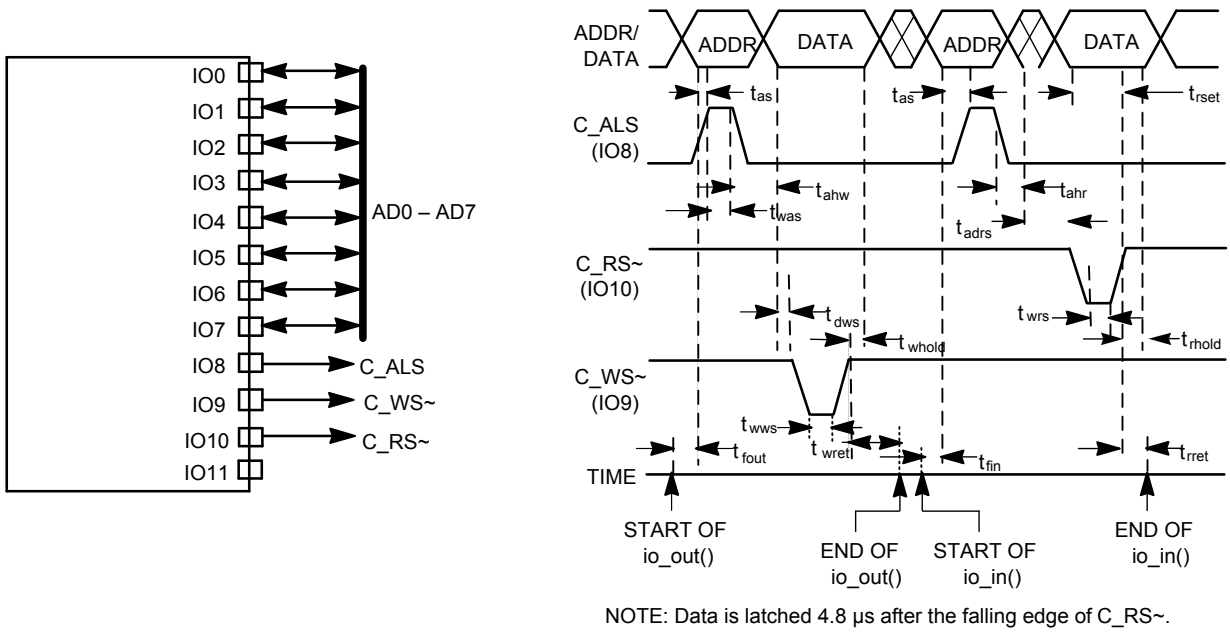


Figure 18. Muxbus I/O for Series 3100 Devices and Timing

Table 20. Muxbus I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{fout}	**io_out()** call to valid address	—	26.4 µs	—
t_{as}	Address valid to address strobe	—	10.8 µs	—
t_{ahw}	Address hold for write	—	4.8 µs	—
t_{ahr}	Address hold for read	—	6.6 µs	—
t_{was}	Address strobe width	—	6.6 µs	—
t_{wrs}	Read strobe width	—	10.8 µs	—
t_{wws}	Write strobe width	—	10.8 µs	—
t_{dws}	Data valid to write strobe	—	6.6 µs	—
t_{rset}	Read setup time	10.8 µs	—	—
t_{whold}	Write hold time	4.2 µs	—	—
t_{rhold}	Read hold time	0 µs	—	—
t_{adrs}	Address disable to read strobe	—	7.2 µs	—

Symbol	Description	Minimum	Typical	Maximum
t_{fin}	**io_in()** call to valid address	—	26.4 µs	—
t_{rret}	Function return from read	—	4.2 µs	—
t_{wret}	Function return from write	—	4.2 µs	—

Programming Considerations

For a **muxbus** output object, the **io_out( )** function requires an optional 8-bit address argument, and an 8-bit data argument. If the address argument is provided, the Neuron firmware first sets pins **IO_0** through **IO_7** as outputs, then places the address value on these pins, and pulses **C_ALS** from low to high to low. This latches the address into the address data latch device. If the address is not provided, this step is skipped. The current value latched in the address latch remains unchanged.

The Neuron firmware then places the data argument value on pins **IO_0** through **IO_7**, and pulses **C_WS~** from high to low to high.

For **muxbus** input, the **io_in( )** function allows an optional 8-bit address argument only. If this argument is provided, the address is emitted and latched in the same manner as for the **io_out( )** function.

Finally, the Neuron firmware sets pins **IO_0** through **IO_7** as inputs. It drops **C_RS~** from high to low, inputs the 8 bits of data from pins **IO_0** through **IO_7**, and raises **~_RS~** from low to high. The function then returns the 8-bit data value read.

After a read operation, pins **IO_0** to **IO_7** are left in the high impedance state. This could cause excessive power consumption of the 8-bit latch. Using pull-up resistors, or ensuring that the last I/O operation is a write, can avoid this situation.

The address argument is optional and can be left off as a performance enhancement where a bus device can be repeatedly read from or written to without changing the bus address. The application must keep track of the current bus address when using this feature.

No events are associated with this I/O model.

Syntax

IO_0 muxbus *io-object-name***;**

IO_0

Specifies pin **IO_0**. Muxbus input/output requires eleven pins and must specify pin **IO_0**.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *data-byte*;

data-byte = **io_in(***io-object-name*, *address***);**
data-byte = **io_in(***io-object-name***);**

io_out(*io-object-name*, *address*, *data-byte***);**
io_out(*io-object-name*, *data-byte***);**

Example

```
IO_0 muxbus ioMuxBus;

when (. . .) {
  // Write two bytes to addresses 0x20 and 0x21,
  // and wait for the data at 0x20 to contain
  // the 0x80 value.
  io_out(ioMuxBus, 0x20, 128);
  io_out(ioMuxBus, 0x21, 1);
  if ((io_in(ioMuxBus, 0x20) & 0x80) == 0) {
    // Continue to read the same address.
    while ((io_in(ioMuxBus) & 0x80) == 0);
  }
}
```

Parallel Input/Output

The **parallel** I/O model uses eleven I/O pins for an 8 bit parallel interface with handshaking. This interface allows data transfer at rates up to 3.3 Mbps. A parallel interface can be used for the following applications:

- To interface a Neuron Chip or Smart Transceiver to an attached microprocessor or to the bus of a computer system. This interface can use the Neuron Chip or Smart Transceiver as a communications chip with an existing processor-based system, provide more application performance, or supply more memory. This type of interface is enhanced with the Microprocessor Interface Program (MIP; with a parallel or dual-ported RAM interface). The MIP moves network variable and application message processing to the attached processor.
- For application-level gateways, two Neuron Chips or Smart Transceivers (or one of each) might be connected back to back across the parallel interface, producing two transceiver interfaces to transport data from one system to the other.

This interface is bidirectional, with the direction (read/write) controlled by the device that is declared as the master. When using this interface, the Neuron Chip or Smart Transceiver can be either a master or a slave. The parallel I/O model provides three different configurations of the parallel I/O interface: master, slave A, and slave B:

- Master and slave A connections are typically used for parallel port interfaces and for Neuron Chip/Smart Transceiver to Neuron Chip/Smart Transceiver communication.
- Slave B connections are typically used for communicating from a microprocessor bus to a Neuron Chip or Smart Transceiver.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to the Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Pins IO0 – IO10 can be configured as a bidirectional 8-bit data and 3-bit control port for connecting to an external processor. The other processor can be a computer, microcontroller, or another Neuron Chip or Smart Transceiver (for gateway applications). The parallel interface can be configured in master, slave A, or slave B mode. Typically, two Smart Transceivers interface in master/slave A mode, and a Smart Transceiver interfaces with a microprocessor in the slave B configuration, with the other microprocessor as the master. Handshaking is used in both modes to control the instruction execution, and application processing is suspended for the duration of the transfer (up to 255 bytes/transfer).

Upon a reset condition, the master processor monitors the low transition of the handshake (HS) line from the slave, then passes a **CMD_RESYNC** (0x5A) command for synchronization. This command must be sent within 0.84 seconds after reset goes high (for a Series 3100 slave running at 10 MHz or a Series 5000/6000 slave at any clock rate), to avoid a watchdog reset error condition.

The **CMD_RESYNC** command is followed by the slave acknowledging with a **CMD_ACKSYNC** (0x07) command. This synchronization ensures that both processors are properly reset before data transfer occurs. When interfacing two Smart Transceivers, these characters are passed automatically. However, when using parallel I/O to interface the Smart Transceiver to a microprocessor, that microprocessor must duplicate the interface signals and characters that are automatically generated by the parallel I/O function of the Smart Transceiver.

For additional information, see the *Parallel I/O Interface to the Neuron Chip* engineering bulletin.

The timing numbers listed in this section are valid for both an explicit I/O call or an implicit I/O call through a when clause, and are assumed to be for a Series 3100 Smart Transceiver running at 10 MHz.

Master Mode and Slave A Mode

The master mode and the slave A mode are recommended when interfacing two Neuron Chips or Smart Transceivers. In a master/slave A configuration, the master drives the IO8 pin as a chip select and the IO9 pin to specify a read or write cycle, and the slave drives the IO10 pin as a handshake (HS) acknowledgment (see **Figure 19**).

Important: The HS line should be pulled up (inactive) with a 10 kΩ resistor to ensure proper resynchronization behavior after the slave device resets.

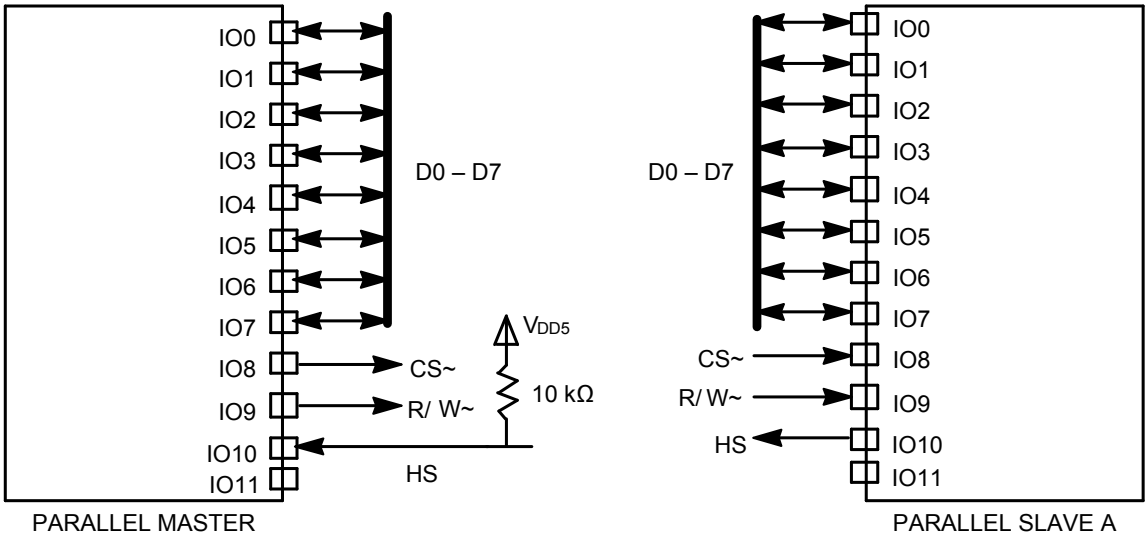


Figure 19. Master Mode and Slave A Mode

The maximum data transfer rate is 1 byte per 4 processor instruction cycles (2.4 µs per byte for a Series 3100 device with a 10 MHz input clock rate, or 300 ns per byte for a Series 5000 or Series 6000 device with an 80 MHz system clock). The data transfer rate scales proportionally to the input clock rate (a master write is a slave read).

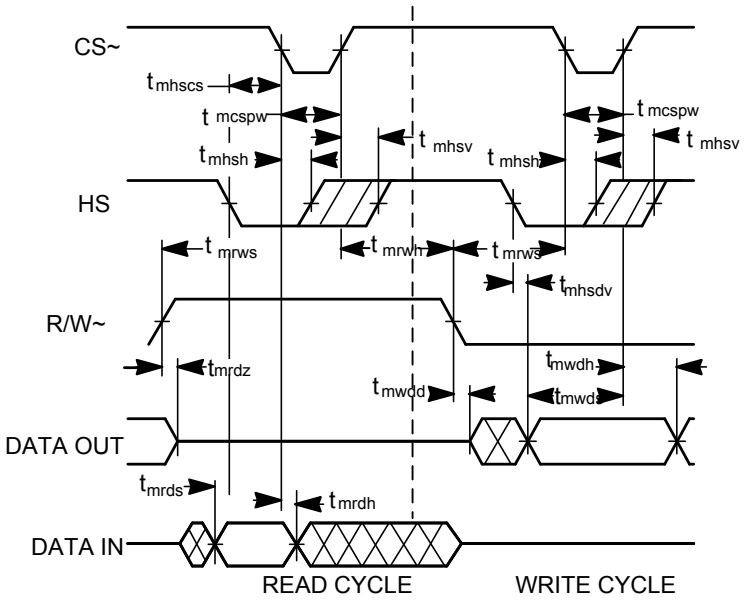


Figure 20. Master Mode Timing

Timing for the case where the Smart Transceiver is the master (**Table 21**), refers to measured output timing for a Series 3100 device at 10 MHz. After every byte write or byte read, the HS line is monitored by the master, to verify that the slave has completed processing (when HS = 0) and the slave is ready for the next byte transfer. This is done automatically in Smart Transceiver-to-Smart Transceiver (master/slave A mode) data transfers.

Slave A timing is shown in **Figure 21**.

Table 21. Master Mode Parallel I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{mrws}	R/W~ setup before falling edge of CS~ (See note 1)	150 ns	3 XIN	—
t_{mrwh}	R/W~ hold after rising edge of CS~	100 ns	—	—
t_{mcspw}	CS~ pulse width (See note 1)	150 ns	2 XIN	—
t_{mhsh}	HS hold after falling edge of CS~	0 ns	—	—
t_{mhsv}	HS checked by firmware after rising edge of CS~ (See note 1)	150 ns	10 XIN	—
t_{mrdz}	Master three-state DATA after rising edge of R/W~ (See notes 2, 3)	—	0 ns	25 ns
t_{mrds}	Read data setup before falling edge of HS (See note 4)	0 ns	—	—
t_{mhscs}	HS low to falling edge of CS~ (See notes 5, 1)	2 XIN	6 XIN	—
t_{mrdh}	Read data hold after falling edge of CS~	0 ns	—	—
t_{mwdd}	Master drive of DATA after falling edge of R/W~ (See notes 2, 1)	150 ns	2 XIN	—
t_{mhsdv}	HS low to data valid (See note 5)	—	50 ns	—
t_{mwds}	Write data setup before rising edge of CS~ (See note 1)	150 ns	2 XIN	—
t_{mwdh}	Write data hold after rising edge of CS~ (See note 6)	Note 6	—	—

Symbol	Description	Minimum	Typical	Maximum

Notes:

1. XIN represents the period of the Smart Transceiver input clock (100 ns for a Series 3100 device at 10 MHz), or the period of the system clock for Series 5000 and Series 6000 devices (12.5 ns at 80 MHz).
2. Refer to the appropriate Neuron Chip or Smart Transceiver data sheet for detailed measurement information.
3. For Smart Transceiver-to-Smart Transceiver operation, bus contention (t_{mrdz}, t_{sawdd}) is eliminated by firmware, ensuring that a zero state is present when the token is passed between the master and slave. See the *Parallel I/O Interface to the Neuron Chip* engineering bulletin for additional information.
4. HS high is used as a slave busy flag. If HS is held low, the maximum data transfer rate is 24 XIN per byte. If HS is not used for a flag, caution should be taken to ensure that the master does not initiate a data transfer before the slave is ready.
5. Parameters were added to aid interface design with the Smart Transceiver.
6. Master holds output data valid during a write until the slave device pulls HS high.
7. In a master read, CS~ pulsing low acts like a handshake to flag the slave that data has been latched in.

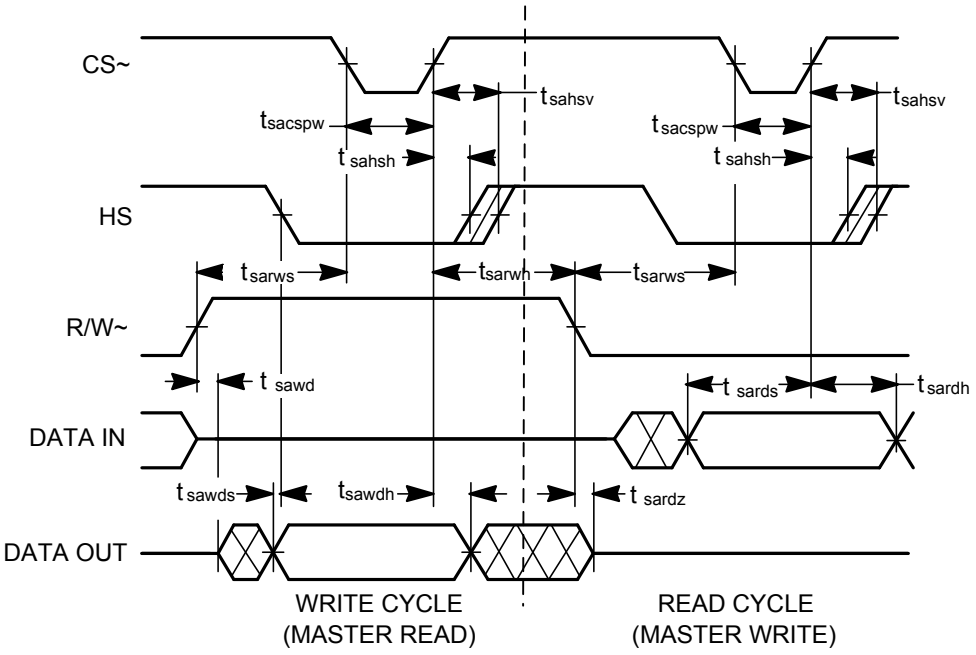


Figure 21. Slave A Mode Timing

Table 22. Slave A Mode Parallel I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{sarws}	R/W~ setup before falling edge of CS~	25 ns	—	—
t_{sarwh}	R/W~ hold after rising edge of CS~	0 ns	—	—
t_{sacspw}	CS~ pulse width	45 ns	—	—
t_{sahsh}	HS hold after rising edge of CS~	0 ns	—	—
t_{sahsv}	HS valid after rising edge of CS~	—	—	50 ns
t_{sawdd}	Slave A drive of DATA after rising edge of R/W~ (Notes 1, 2)	0 ns	5 ns	—
t_{sawds}	Write data valid before falling edge of HS (Note 3)	150 ns	2 XIN	—
t_{sawdh}	Write data valid after rising edge of CS~ (Note 3)	150 ns (Note 4)	2 XIN	—
t_{sardz}	Slave A three-state DATA after falling edge of R/W~ (Note 1)	—	—	50 ns
t_{sards}	Read data setup before rising edge of CS~	25 ns	—	—
t_{sardh}	Read data hold after rising edge of CS~	10 ns	—	—

Notes:

1. Refer to the appropriate Neuron Chip or Smart Transceiver data sheet for detailed measurement information.
2. For Smart Transceiver-to-Smart Transceiver operation, bus contention (t_{mrdz}, t_{sawdd}) is eliminated by firmware, ensuring that a zero state is present when the token is passed between the master and slave. See the *Parallel I/O Interface to the Neuron Chip* engineering bulletin for additional information.
3. XIN represents the period of the Smart Transceiver input clock (100 ns for a Series 3100 device at 10 MHz), or the period of the system clock for a Series 5000 device (12.5 ns at 80 MHz).
4. If $t_{sarwh} < 150$ ns, then $t_{sawdh} = t_{sarwh}$.
5. In slave A mode, the HS signal is high a minimum of 4 XIN periods. The typical time HS is high during consecutive data reads or consecutive data writes is also 4 XIN periods.

Example

This section describes a pair of example programs that transfer data in a parallel I/O master/slave A configuration. The code assumes two devices hardwired as shown in **Figure 19**. The master program writes the **test_data** to the input

buffer of the slave (because the master owns the token after reset and has the first option to write on the bus) and the slave then outputs data to the input buffer of the master. You can view the buffers using the NodeBuilder debugger to verify that the transfer was completed.

The master transmits the pattern [5,1,1,1,1,1] to the slave and the slave transmits the pattern [7,1,2,3,4,5,6,7,0,0,0,0,0,0] to the master. The first byte indicates the number of bytes being passed; the following non-zero valued bytes in this example are the actual data bytes transferred. The remaining length of the array, if any, is filled with zeroes.

The master program writes once to the slave and reads once from the slave. To implement continuous writes and reads, add an **io_out_request()** function call after the **io_in()** function call in the master program.

If a watchdog timeout occurs for either device, simultaneously reset the two devices.

Master Program

```
/*
 * This is the master program. After reset, the buffer is
 * filled with 1s and then the buffer is written to the
 * slave. The master then reads the slave's buffer. The
 * master's output buffer should contain [5,1,1,1,1,1]; the
 * input buffer should contain
 * [7,1,2,3,4,5,6,7,0,0,0,0,0,0].
 */

IO_0 parallel master parallelBus;

// data to be written in output buffer
#define TEST_DATA 1

// maximum length of input data expected
#define MAX_IN 13

// output length can be equal to or less than the max
#define OUT_LEN 5

// maximum array length
#define MAX_OUT 5

// output structure
struct parallel_out {
  // actual length of data to be output
  unsigned int length;
  // array setup for max length of data to be output
  unsigned int buffer[MAX_OUT];
} outData;

// input structure
struct parallel_in {
  // actual buffer length to be input
  unsigned int length;
  // maximum input array
  unsigned int buffer[MAX_IN];
} inData;
```

```
unsigned int i;
when (reset) {
  outData.length = OUT_LEN;  // assign output length
  for(i=0; i<OUT_LEN; ++i) {
    // fill output buffer with 1s
    outData.buffer[i] = TEST_DATA;
  }
  io_out_request(parallelBus); // request to output buffer
}

when (io_out_ready(parallelBus)) {
  // output buffer when slave is ready
  io_out(parallelBus, &outData);
}

when (io_in_ready(parallelBus)) {
  // declare the maximum input buffer acceptable
  inData.length = MAX_IN;
  io_in(parallelBus, &inData); // store input in buffer
}
```

Slave Program

```
/*
 * This is the slave program. After reset, the output
 * buffer is filled with data and then the slave reads from
 * the master. The slave then writes to the master. The
 * slave's input buffer should contain [5,1,1,1,1,1]; the
 * output buffer should contain
 * [7,1,2,3,4,5,6,7,0,0,0,0,0,0].
 */

IO_0 parallel slave parallelBus;

// maximum length of input data expected
#define MAX_IN 5

// output length can be equal to or less than the max
#define OUT_LEN 7

// maximum array length
#define MAX_OUT 13

// output structure
struct parallel_out {
  // actual length of data to be output
  unsigned int length;
  // array setup for max length of data to be output
  unsigned int buffer[MAX_OUT];
} outData;

// input structure
struct parallel_in {
  // actual length of buffer to be input
  unsigned int length;
  // maximum input array
```

```
  unsigned int buffer[MAX_IN];
} inData;

unsigned int i;
when (reset) {
  outData.length = OUT_LEN;  // assign output length
  for(i=0; i<OUT_LEN; ++i)  // fill output buffer with 1s
    outData.buffer[i]=i+1;
}

when (io_out_ready(parallelBus)) {
  io_out(parallelBus, &outData); // output buffer
}

when (io_in_ready(parallelBus)) {
  // declare the maximum input buffer acceptable
  inData.length = MAX_IN;
  io_in(parallelBus, &inData); // store input in buffer
  io_out_request(parallelBus); // request to output buffer
}
```

Slave B Mode

The slave B mode is recommended for interfacing a Smart Transceiver acting as the slave to a microprocessor acting as the master. When configured in slave B mode, the Smart Transceiver accepts the IO8 signal as a chip select and the IO9 signal to specify whether the master will read or write, and accepts the IO10 signal as a register select input. Series 5000 and Series 6000 devices accept the IO11 pin as an interrupt request signal. When the CS~ pin is asserted and either IO10 is low or IO10 is high and R/W~ is low, pins IO0 – IO7 form the bidirectional data bus. When IO10 is high, R/W~ is high, and CS~ is asserted, IO0 is driven as the HS acknowledgment signal to the master.

The Smart Transceiver can appear as two registers in the master's address space:

- A read/write data register
- A read-only status register

Therefore, reads by the master to an odd address access the status register for handshaking acknowledgments, and all other reads or writes access the data register for I/O transfers. The least-significant bit (LSB) of the control register, which is read through pin IO0, is the HS bit. The master reads the HS bit after every master read or write.

Important: The D0/HS line should be pulled up (inactive) with a 10 kΩ resistor to ensure proper resynchronization behavior after resets.

When acting as a slave to a microprocessor, the Smart Transceiver slave B mode handles all handshaking and token passing automatically. However, the master microprocessor must read the HS bit after each transaction and must also internally track the token passing. This mode is designed for use with a master processor that uses memory-mapped I/O, because the LSB of the master's address bus is typically connected to the IO10 pin of the Smart Transceiver. This is illustrated in **Figure 22** and **Figure 23**.

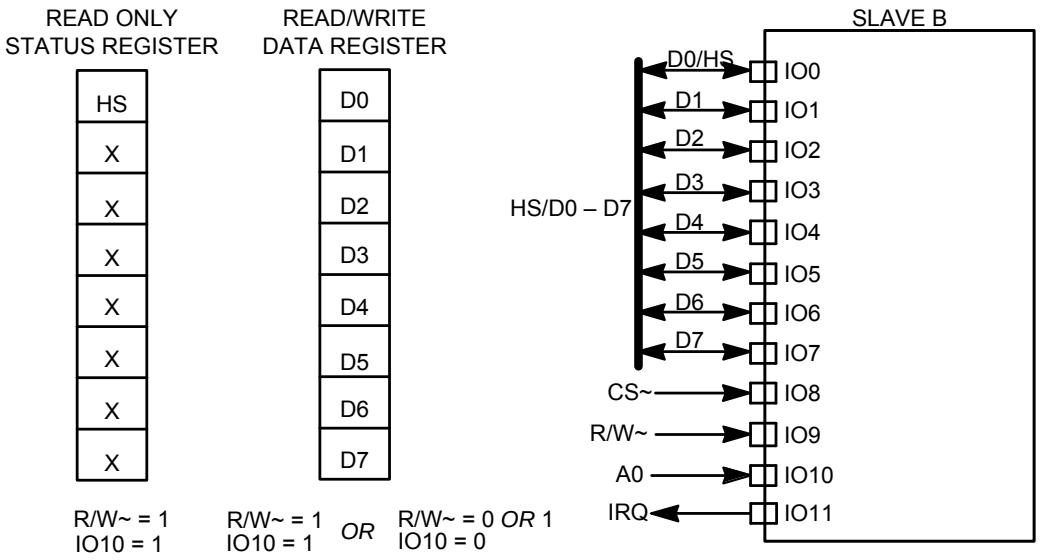


Figure 22. Parallel I/O Master/Slave B as a Memory-Mapped Device

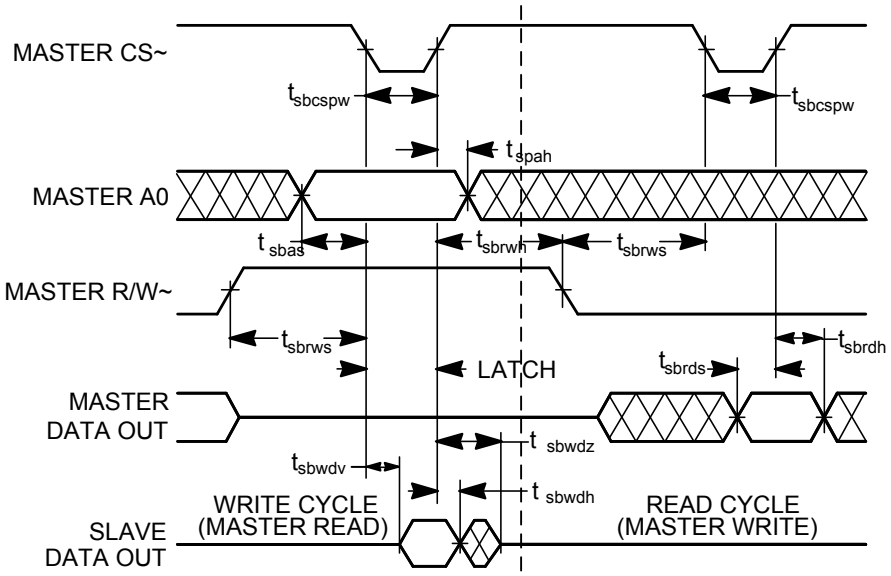


Figure 23. Slave B Mode Timing

Table 23. Slave B Mode Parallel I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{sbrws}	R/W~ setup before falling edge of CS~	0 ns	—	—
t_{sbrwh}	R/W~ hold after rising edge of CS~	0 ns	—	—
t_{sbcspw}	CS~ pulse width	Note 1	—	—

Symbol	Description	Minimum	Typical	Maximum
t_{sbas}	A0 setup to falling edge of CS~	10 ns	—	—
t_{sbah}	A0 hold after rising edge of CS~	0 ns	—	—
t_{sbwdv}	CS~ to write data valid	—	—	50 ns
t_{sbwdh}	Write data hold after rising edge of CS~ (Notes 2, 3)	0 ns	30 ns	—
t_{sbwdz}	CS~ rising edge to Slave B release data bus (Note 2)	—	—	50 ns
t_{sbrds}	Read data setup before rising edge of CS~	25 ns	—	—
t_{sbrdh}	Read data setup before rising edge of CS~	10 ns	—	—

Notes:

1. The slave B write cycle (master read) CS~ pulse width is directly related to the slave B write data valid parameter and master read setup parameter. To calculate the write cycle CS~ duration needed for a special application use:
   $t_{sbcspw} = t_{sbwdv}$ + master's read data setup before rising edge of CS~.
   Refer to the master's specification data book for the master read setup parameter.
   The slave read cycle minimum CS~ pulse width = 50 ns.
2. Refer to the appropriate Neuron Chip or Smart Transceiver data sheet for detailed measurement information.
3. The data hold parameter, t_{sbwdh}, is measured to the disable levels shown in the appropriate Neuron Chip or Smart Transceiver data sheet, rather than to the traditional data invalid levels.
4. In a slave B write cycle, the timing parameters are the same for a control register (HS) write as for a data write.
5. Special applications: Both the state of CS~ and R/W~ determine a slave B write cycle. If CS~ cannot be used for a data transfer, then toggling the R/W~ line can be used with no changes to the hardware. That is, if CS~ is held low during a slave B write cycle, a positive pulse (low to high to low) on R/W~ can execute a data transfer. The low-to-high transition on R/W~ causes slave B to drive data with the same timing parameters as t_{sbwdv} (redefined R/W~ to write data valid). Likewise, the falling edge of R/W~ causes slave B to release the data bus with the same timing limits as the CS~ rising edge in t_{sbwdz}. This scenario is only true for a slave B write cycle, and is not applicable to a slave B read cycle or any slave A data transitions. This application can be helpful if the master has separate read and write signals but no CS~ signal. Caution must be taken to ensure the bus is free before transfers to avoid bus contention.

Token Passing

Virtual token passing is implemented to eliminate the possibility of data bus contention. The token is owned by the master after synchronization and is passed between the master and slave devices. After each data transfer is completed, the token owner writes an end of message (EOM) (0x00) to indicate that the transfer is complete. The EOM is never read. Instead, "processing the EOM" indicates passing of the token.

Token passing can be achieved by executing either a data packet or a NULL transfer. Only the owner of the token can write to the bus. Therefore, when the master performs two writes of data (1 – 255 bytes each), a dummy read cycle (NULL character = 0x00) must be inserted between them in order to pass the token. Token passing is executed automatically in a Smart Transceiver-to-Smart Transceiver interface. See *Transferring Data* for master/slave flow transactions.

Handshaking

Handshaking allows the master to monitor the slave between every byte transfer, ensuring that both processors are ready for the byte to be transferred. If the master owns the token, the master waits for the HS from the slave before writing data to the bus. If the slave owns the token, the master monitors the low transition of the HS before reading the bus.

In master or slave A mode, the Smart Transceiver HS line is pin IO10. In slave B mode, the Smart Transceiver HS bit is monitored on IO0 which corresponds to the least significant data bit of the status register.

Transferring Data

The data transfer operation between the master and the slave is accomplished through the use of a virtual write token-passing protocol. The write token is passed alternatively between the master and the slave on the bus in an infinite ping-pong fashion. The owner of the token has the option of writing a series of data bytes, or alternatively, passing the write token without any data. **Figure 24** illustrates the sequence of operations for this token passing protocol.

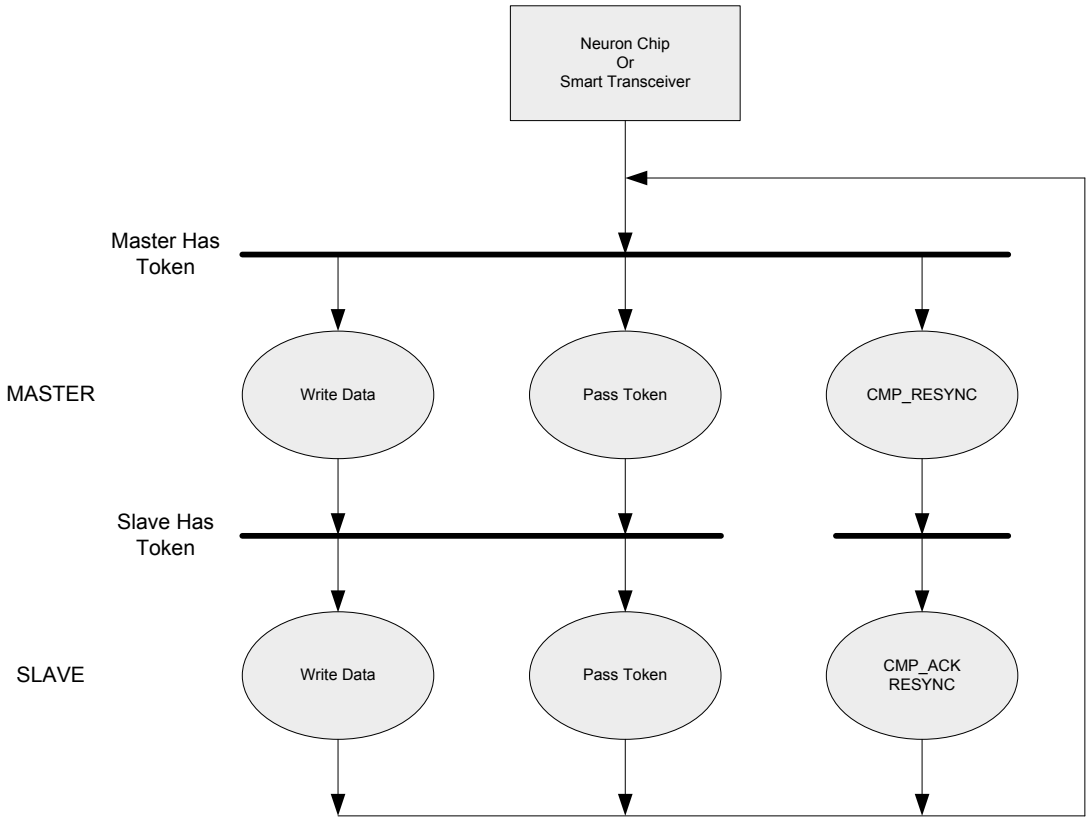


Figure 24. Handshake Protocol Sequence between Master and Slave

When in possession of the write token, the device (Neuron Chip, Smart Transceiver, or a host processor) can transfer up to 255 bytes of data. The stream of data bytes is preceded by the command and length bytes. The token holder keeps possession of the token until all data bytes have been written, after which the token is passed to the attached device.

The same process can now be repeated by the other side or, alternatively, the token can be passed back without any data.

Resynchronization Procedure

The procedure shown in **Table 24** through **Table 28** applies to master/slave A and master/slave B configuration. The master initiates the resynchronization with a RESYNC (0x5A) command, and the slave acknowledges with an ACKSYNC (0x07) command. If the slave does not respond, the master continues to send the RESYNC command until the slave responds correctly.

Table 24. Resynchronization

Step	Master	Slave	Comment
1	(Owns Token)		
2	Write RESYNC		Master initiates resynchronization (0x5A)

Step	Master	Slave	Comment
3		Read RESYNC	
4	Write EOM		End of message (EOM=0x00)
5		Process EOM	
6		Write ACKSYNC	Slave acknowledges resynching (0x07)
7	Read ACKSYNC		
8		Write EOM	
9	Process EOM		Master owns token when reset
10	(Owns Token)		

Table 25. Master Writes Buffer to Slave: R/W~=0

Step	Master	Slave	Comment
1	(Owns Token)		
2	Write XFER		Master has data to write (XFER=0x01)
3		Read XFER	
4	Write (length)		Length=number of bytes of data
5		Read (length)	
6	Write (data_0)		Master begins data transfer to slave
7		Read (data_0)	
8			Repeat steps 6 and 7 *length* times
9	Write (data_n)		Last byte of data to be transferred
10		Read (data_n)	
11	Write EOM		End of data transfer (EOM=0x00)
12		Process EOM	Exchange token
13		(Owns Token)	

Table 26. Slave Writes Buffer to Master: R/W~=1

Step	Master	Slave	Comment
1		(Owns Token)	
2		Write XFER	Slave has data to write (XFER=0x01)
3	Read XFER		
4		Write (length)	Length=number of bytes of data
5	Read (length)		
6		Write (data_0)	Salve begins data transfer to master
7	Read (data_0)		
8			Repeat steps 6 and 7 *length* times
9		Write (data_n)	Last byte of data to be transferred
10	Read (data_n)		
11		Write EOM	End of data transfer (EOM=0x00)
12	Process EOM		Exchange token
13	(Owns Token)		

Table 27. Master Passes Token to Slave

Step	Master	Slave	Comment
1	(Owns Token)		
2	Write NULL		Master has no data to send to slave
3		Read NULL	NULL=0x00
4	Write EOM		End of data transfer (EOM=0x00)
5		Process EOM	Exchange token
6		(Owns Token)	

Table 28. Slave Passes Token to Master

Step	Master	Slave	Comment
1		(Owns Token)	
2		Write NULL	Slave has no data to send to master
3	Read NULL		NULL=0x00
4		Write EOM	End of data transfer (EOM=0x00)
5	Process EOM		Exchange token
6	(Owns Token)		

Using the IRQ Signal

The Series 5000 and Series 6000 devices can use the IRQ pin as an indication that the network is ready, either for uplink or for downlink. The Neuron C application in the FT 5000 Smart Transceiver, Neuron 5000 Processor, FT 6000 Smart Transceiver or Neuron 6000 Processor would assert the IRQ pin high to cause an interrupt for the host device.

A downlink ready interrupt would allow the Series 5000 or Series 6000 device to inform the host when it has read the first byte of a transfer. This interrupt would account for the latency of the parallel interface, that is, between a host write for a downlink transfer and the Series 5000 or Series 6000 device read for the transfer. This latency would be on the order of 110 microseconds (for a 10 MHz system clock), but it could be longer if the Series 5000 or Series 6000 device is busy processing an incoming network frame. The host could initiate a downlink transfer by writing only the length byte; it then could let the interrupt service routine handle the rest of the transfer.

An uplink ready interrupt would provide an indication from the Series 5000 or Series 6000 device that uplink traffic needs to be transferred. The IRQ pin would be asserted only when the Series 5000 or Series 6000 device does not own the write token.

The IRQ pin would be deasserted during downlink activity.

Although there are two interrupt cases, there is only a single interrupt request (IRQ) line. The interrupt type would be determined by the host based on the state of the Series 5000 or Series 6000 device and token ownership.

Programming Considerations

Multiple slave B devices can be connected to a single bus. The difference between slave A and slave B concerns the use of one of the three control signals (see the description of the **slave**, **slave_b**, and **master** keywords).

No other I/O objects can be declared on pins **IO_0** through **IO_10** when the parallel I/O object is being used.

Neuron C Resources

In order to use the parallel I/O model of the Neuron Chip or Smart Transceiver, the **io_in( )** and **io_out( )** functions require a pointer to the **parallel_io_interface** structure:

```
struct parallel_io_interface {
  unsigned length;            // length of data field
  unsigned data[MAXLENGTH];   // data field
} piofc;
```

The **parallel_io_interface** structure must be declared in the application program, with an appropriate definition of **MAXLENGTH** to signify the largest expected buffer size for any data transfer, up to a maximum value of 255.

For the **io_out( )** function, **length** is the number of bytes to be transferred out and is set by the application program. For the **io_in( )** function, **length** is the number of bytes to be transferred in. If the incoming length is larger than **length**, then the incoming data stream is flushed, and **length** is set to zero. Otherwise, **length** is set to the number of data bytes read. The length field must be set before calling the **io_in( )** or **io_out( )** function. The maximum value for the **length** field is 255.

The following functions and events are provided specifically for use with the parallel I/O object:

io_in_ready

This event becomes **TRUE** whenever a message arrives on the parallel bus that must be read. The application must then call the **io_in( )** function to retrieve the data.

io_out_request( )

This function is used to request an **io_out_ready** indication for a parallel I/O object. It is up to the application to buffer the data until the **io_out_ready** event is **TRUE**. This function acquires the token for the parallel I/O interface.

io_out_ready

This event becomes **TRUE** whenever the parallel bus is in a state where it can be written to and the **io_out_request( )** function was previously called. The application must then call the **io_out( )** function to write the data to the parallel port. This function relinquishes the token for the parallel I/O interface.

Neuron C applications can also use the parallel bus in a unidirectional manner (that is, applications don't need to use both the **when(io_in_ready)** or **when(io_out_ready)** clauses if they only need to use one).

See *Performing I/O: Functions and Events*, the *Neuron C Programmer's Guide*, and the *Parallel I/O Interface to the Neuron Chip* engineering bulletin (part no. 005-0021-01) for additional information.

To prevent contention for the data bus, a virtual write token is passed back and forth between the master device and the slave device (in both slave A and slave B modes). The master device has the write token initially after a reset. The parallel I/O object automatically manages the write token; no specific application code is needed.

Syntax

IO_0 parallel slave | slave_b | master *io-object-name***;**

IO_0

Parallel input/output requires eleven pins and must specify pin **IO_0**. **Table 29** shows how the pins are used.

Table 29. Pins for Parallel I/O Object

Pin	Master	Slave A	Slave B
IO_0 thru IO_7	Data Bus	Data Bus	Data Bus
IO_8	Chip select output	Chip select input	Chip select input
IO_9	RD/~WR output	RD/~WR input	RD/~WR input
IO_10	HANDSHAKE input	HANDSHAKE input	A0 input
IO_11	N/A	N/A	IRQ
Note: IO_11 as IRQ is only available for Series 5000 and Series 6000 devices.			

slave | slave_b | master

Specifies slave A, slave B, or master mode. For master and slave A modes, **IO_10** is a handshake signal. For slave B mode, **IO_10** becomes an address line input, A0, and the handshake signal appears on the data bus on pin **IO_0** when **A0=1**. When **A0=0**, the data appears on the data bus. This mode is used to allow a Neuron Chip or Smart Transceiver to reside on a microprocessor bus with the data at one address location and the handshake signal at another.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

```
struct parallel_io_interface {
  unsigned int length;
  unsigned int data[data-size];
} piofc;

io_in(io-object-name, &piofc);

io_request(io-object-name);

io_out(io-object-name, &piofc);
```

Example

The following example shows how to use the **io_in_ready** and **io_out_ready** events, in conjunction with the **io_out_request( )** function, to handle parallel I/O processing.

```
#define DATA_SIZE 255
struct parallel_io_interface {
  unsigned int length;            // length of data field
  unsigned int data [DATA_SIZE];  // data field
} piofc;
IO_0 parallel slave slaveBus;

// ready to input data
when (io_in_ready(slaveBus)) {
  piofc.length = DATA_SIZE; // number of bytes to read
  io_in(slaveBus, &piofc); // get 10 bytes of incoming data
}

// ready to output data
when (io_out_ready(slaveBus)) {
  piofc.length = 10;     // number of bytes to write
  io_out(slaveBus, &piofc); // output 10 bytes from buffer
}

// user defined event
when (...) {
  io_out_request(slaveBus); // post the write request
}
```

4

Serial I/O Models

This chapter describes serial input/output models. Serial I/O objects are used for transferring data serially over a pin or a set of pins. Only one type of serial I/O model can be used within a single Neuron Chip or Smart Transceiver. Both the input and output versions of the serial type can coexist within a single Neuron Chip or Smart Transceiver.

Bitshift Input/Output

The **bitshift** I/O model is used to shift a data word of up to 16 bits into or out of the Neuron Chip or Smart Transceiver. Data is clocked in and out by an internally generated clock. This model is useful for transferring data to external logic that uses shift registers.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to the Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Pairs of adjacent pins can be configured as serial input or output lines. The first pin of the pair can be IO0-IO6, IO8, or IO9, and is used for the clock (driven by the Smart Transceiver). The adjacent higher-numbered I/O pin is then used for up to 16 bits of serial data. The bit rate can be configured as 1 kbps, 10 kbps, or 15 kbps for a Series 3100 device with a 10 MHz input clock; the bit rate can be configured as 16 kbps, 160 kbps, or 240 kbps for a Series 5000 and Series 6000 device with an 80 MHz input clock. The bit rate scales proportionally to the input clock rate. The active clock edge can be specified as either rising or falling. This function suspends application processing until the operation is complete.

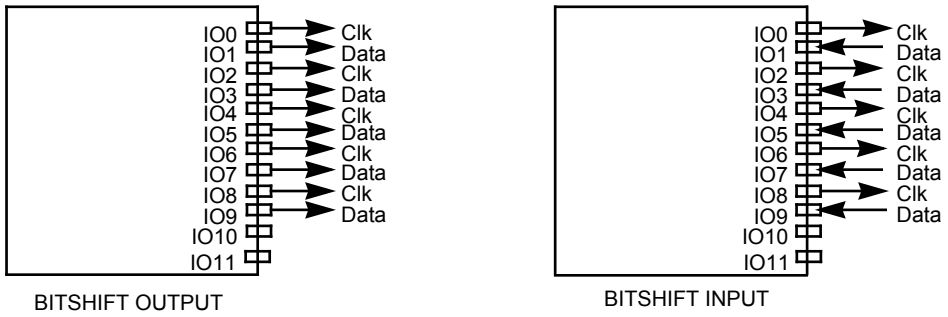
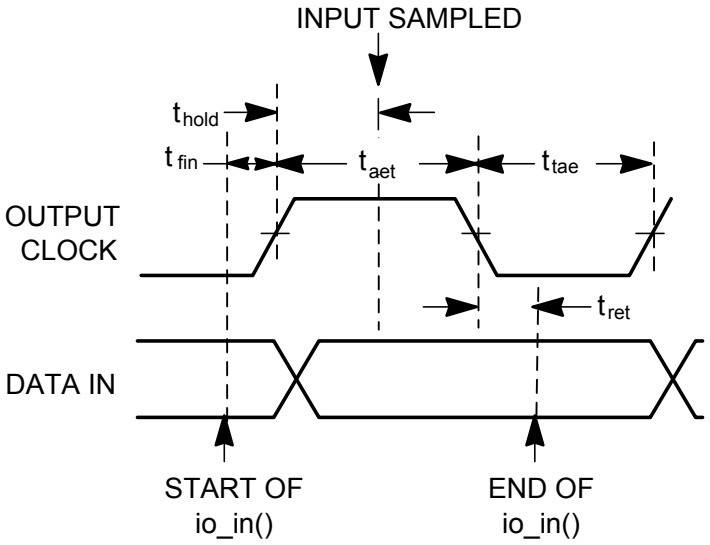


Figure 25. Bitshift I/O Examples

For bitshift input, the clock output is deasserted (to the inactive level) at the same time as the start of the first bit of data. For bitshift output, the clock output is initially inactive prior to the first bit of data (unless overridden by a bit output overlay).

Figure 26 and **Figure 27** show the bitshift input and output latency times, respectively. These are the times from the call to the **io_in()** or **io_out()** function, until a value is returned. The direction of bit ports can be changed between input and output dynamically by using the **io_set_direction()** function.



Active clock edge assumed to be positive in the above diagram.

Figure 26. Bitshift Input Timing

Table 30. Bitshift Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to first edge	156.6 µs
t_{ret}	Return from function	5.4 µs
t_{hold}	Active clock edge to sampling of input data 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 9 µs 40.8 µs 938.2 µs
t_{aet}	Active clock edge to next clock transition 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 31.8 µs 63.6 µs 961 µs
t_{tae}	Clock transition to next active clock edge 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 14.4 µs 14.4 µs 14.4 µs
f	Clock frequency = $1/(t_{aet} + t_{tae})$ 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 21.6 kHz 12.8 kHz 1.03 kHz

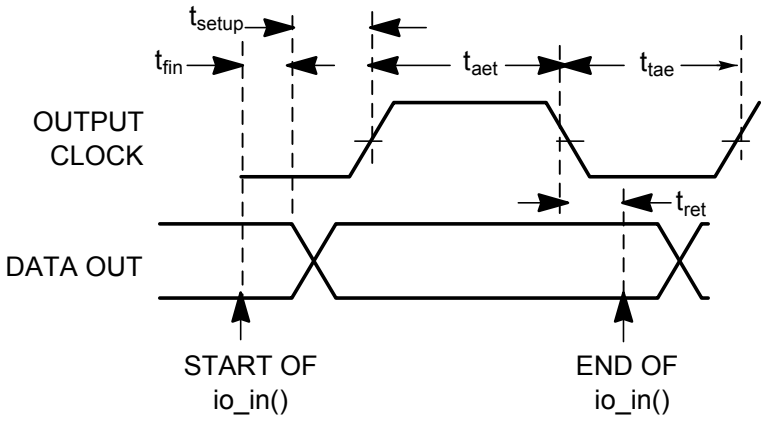


Figure 27. Bitshift Output Timing

Table 31. Bitshift Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to first data out stable 16-bit shift count 1-bit shift count	 185.3 µs 337.6 µs
t_{ret}	Return from function	10.8 µs
t_{setup}	Data out stable to active clock edge 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 10.8 µs 10.8 µs 10.8 µs
t_{aet}	Active clock edge to next clock transition 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 10.2 µs 42 µs 939.5 µs
t_{tae}	Clock transition to next active clock edge 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 34.8 µs 34.8 µs 34.8 µs
f	Clock frequency = $1/(t_{aet} + t_{tae})$ 15 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 22 kHz 13 kHz 1.02 kHz

Programming Considerations

For bitshift input/output, the data type of the return value for **io_in( )**, and the data type of the output value for **io_out( )**, is an **unsigned long**.

When using multiple serial I/O devices that have differing bit rates, you must use the following compiler directive: **#pragma enable_multiple_baud**. This pragma must appear prior to the use of any I/O function (such as **io_in( )** or **io_out( )**).

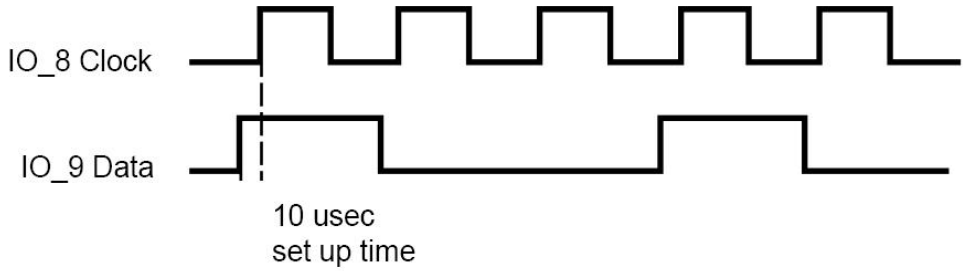


Figure 28. Bitshift Output

Syntax

pin **input bitshift** [**numbits (***const-expr***)**] [**clockedge (+|-)**]
[**kbaud (***const-expr***)**] *io-object-name***;**

pin **output bitshift** [**numbits (***const-expr***)**] [**clockedge (+|-)**]
[**kbaud (***const-expr***)**] *io-object-name* [**=***initial-output-level*]**;**

pin

An I/O pin. Bitshift input/output requires adjacent pins. The Clock pin is the pin specified, and the Data pin is the adjacent pin. The pin specification denotes the lower-numbered pin of the pair and can be **IO_0** through **IO_6**, **IO_8**, or **IO_9**.

numbits (*const-expr***)**

Specifies the number of bits to be shifted in or out. The *const-expr* expression can evaluate to any number from 1 to 31. The default is 16.

Data is shifted in and out with the most significant bit of data first. For the **io_in( )** function, only the last 16 bits shifted in are returned. For the **io_out( )** function, after 16 bits, zeros are shifted out.

You can also specify the number of bits to be shifted in the **io_in( )** or **io_out( )** call. This number temporarily overrides the number specified in the device declaration, for that one call only.

clockedge (+|-)

For inputs, this option specifies whether the data is read on the positive-going or negative-going edge of the clock. For outputs, it specifies whether the data is stable on the positive-going or negative-going edge of the clock. The default value is [**+**].

kbaud (*const-expr***)**

Specifies the bit rate. The expression *const-expr* can be 1, 10, or 15. The default is 15. The firmware uses this value as a multiplier based on the Series 3100 input clock, or the Series 5000 ore Series 6000 system clock. For example, for a Series 3100 device at 10 MHz, **kbaud(15)** yields 15 kbps; for a Series 5000 or Series 6000 device at 10 MHz, **kbaud(15)** yields 30 kbps. The bit rate scales proportionally with the input or system clock.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the clock pin at initialization. The initial state can be 0 or 1; this applies to the clock pin only. The default is 0.

Usage

unsigned long *input-value***;**

unsigned long *output-value***;**

input-value **= io_in(***input-object* [**,** *numbits*]**);**
io_out(*output-object***,** *output-value*[**,** *numbits*]**);**

Bitshift Input Example

```
IO_6 input bitshift numbits(8) ioShiftregister;
unsigned long data;
...

when (...) {
  data = io_in(ioShiftregister);
}
```

Bitshift Output Example

```
IO_8 output bitshift numbits(5) clockedge(+) ioAdcControl;
...

when (...) {
  io_out(ioAdcControl, 0b10010UL);
}
```

I2C Input/Output

The I²C I/O model type is used to interface a Neuron Chip or Smart Transceiver to any device that uses the Inter-Integrated Circuit (I²C) bus protocol developed by Philips Semiconductors (now NXP® Semiconductors). See the Bitshift, Neurowire, SCI, SPI, or Touch I/O models for alternate forms of serial I/O.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

The Smart Transceiver is always the master, with IO8 as the serial clock (SCL) signal and IO9 the serial data (SDA) signal. Alternatively, IO0 can be used as the serial clock (SCL) and IO1 as the serial data (SDA). These I/O lines are operated in the open-drain mode in order to accommodate the special requirements of the I²C protocol. With the exception of two pull-up resistors, no additional external components are necessary for interfacing a Smart Transceiver to an I²C device.

Up to 255 bytes of data can be transferred at a time. At the start of all transfers, a right-justified 7-bit I²C address argument is sent out on the bus immediately after the I²C "start condition." For more information about this protocol, refer to *UM10204: I²C-bus specification and user manual* from NXP Semiconductors.

Figure 29 shows the I²C I/O latency times. These are the times from the call to the **io_in()** or **io_out()** function, until a value is returned. The direction of bit ports can be changed between input and output dynamically by using the **io_set_direction()** function.

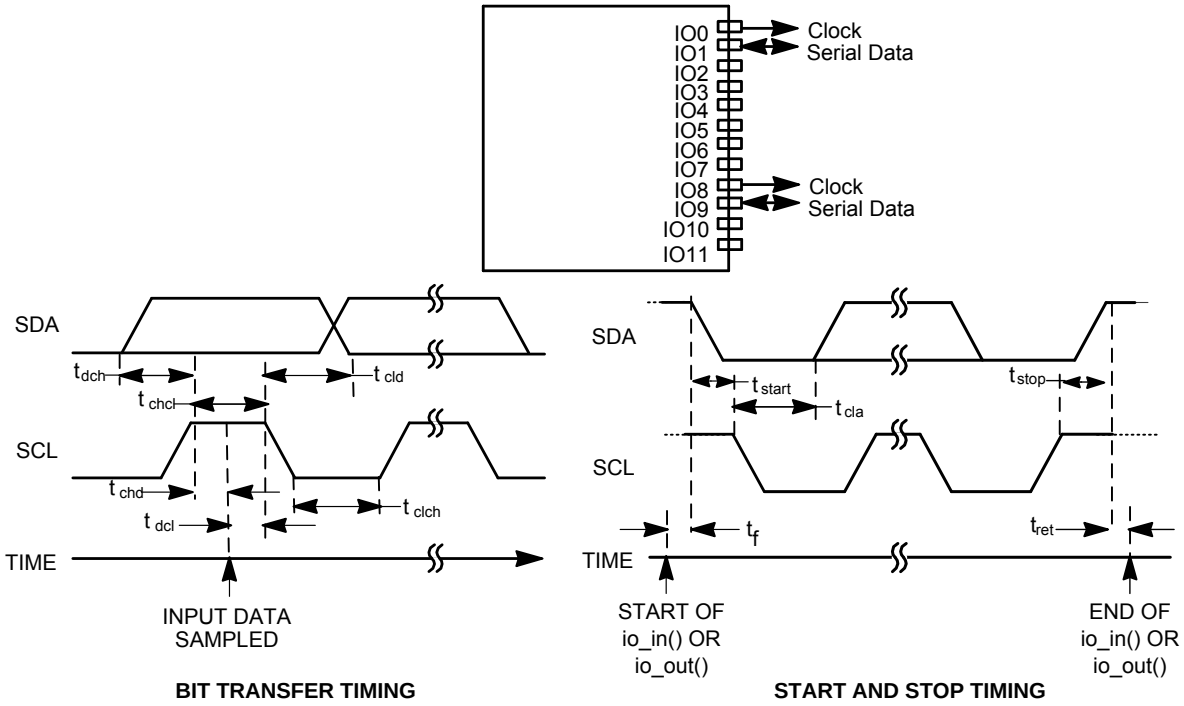


Figure 29. I2C I/O and Timing

Table 32. I2C I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_f	I/O call to start condition io_in() io_out()	—	 54.6 µs 43.4 µs	—
t_{start}	End of start condition io_in() io_out()	 5.4 µs 5.4 µs	—	—
t_{cla}	End of start to start of address io_in() io_out()	 24.0 µs 24.0 µs	—	—
t_{cld}	SCL low to data for io_out()	24.6 µs	—	—
t_{dch}	Data to SCL high for io_out()	7.2 µs	—	—
t_{chcl}	Clock high to clock low for io_out()	12.6 µs	—	—
t_{chd}	SCL high to data sampling for io_in()	13.2 µs	—	—
t_{dcl}	Data sample to SCL low for io_in()	7.2 µs	—	—
t_{clch}	Clock low to clock high for io_in()	24.0 µs	—	—
t_{stop}	Clock high to data io_in() io_out()	 12.6 µs 12.6 µs	—	—
t_{ret}	SDA high to return from function io_in() io_out()	—	—	 4.2 µs 4.2 µs

Programming Considerations

The **i2c** I/O object can be declared with pin **IO_8** as the serial clock (SCL) line, and pin **IO_9** as the serial data (SDA) line, or it can be declared with pin **IO_0** as the serial clock line, and pin **IO_1** as the serial data line. The Neuron Chip or Smart Transceiver acts as a master only. Two external pull-ups are required, and the interface is connected directly to the I/O pins. These I/O pins are operated as open-drain devices in order to support the interface.

An **i2c** I/O object declared on pin **IO_8** can be declared with the **use_stop_condition** option keyword. This option allows for combined format data transfers. For example, you can address and write to a peripheral device with one or more **io_out( )** calls with stop set to **FALSE**, followed by a call to **io_out( )** with stop set to **TRUE** to finish the transfer with the STOP condition.

For all transfers, an I²C device address argument is required. This byte must be the right-justified 7-bit I²C device address. Up to 255 bytes of data can be transferred at a time. The address is written to the bus at the start of any transfer, immediately following the I²C bus start condition. A count argument is also required; this controls how many data bytes are to be written or read.

For I²C input/output, **io_in( )** and **io_out( )** return a 0 or 1 value reflecting the fail (0) or pass (1) status of the transfer. A failed status indicates that the addressed device did not acknowledge positively on the bus, or that the SCL was low at the start of the transfer.

For more information on this protocol and the devices that it supports, see documentation for Philips Semiconductors Microcontroller Products, under I²C bus descriptions. This I/O model implementation was modified for Neuron C Version 2.1. To use the previous implementation (in case of modification to existing applications where the previous implementation is required for memory considerations) use the **#pragma codegen use_i2c_version_1** compiler directive. See the *Neuron C Reference Guide* for information about this pragma. If you use the Version 1 **i2c** model, you must use pin **IO_8** and you cannot use any modifiers.

Syntax

pin **i2c** [**use_stop_condition**] [**__slow**] [**__fast**] *io-object-name***;**

pin

Specify pin **IO_0** or **IO_8**. The **i2c** model requires pins **IO_0** and **IO_1**, or **IO_8** and **IO_9**.

use_stop_condition

Optionally specifies that data transfers should be repeated until a stop condition is reached. A stop condition is defined as a change in the state of the data line (SDA), from LOW to HIGH, while the clock line (SCL) is HIGH.

__slow

Optionally specifies that the I²C bus should use the standard mode of 100 kbps. This mode is the default if no mode is specified. Mutually exclusive with **__fast**.

__fast

Optionally specifies that the I²C bus should use the fast mode of 400 kbps. Mutually exclusive with **__slow**.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

boolean *return-value***;**
unsigned int *data-buffer***[***buffer-size***];**
unsigned int *dev-address***,** *count***;**

// i2c I/O object without the stop condition specified (stop assumed)
return-value **= io_in(***io-object-name***,** *data-buffer***,** *dev-address***,** *count***);**
return-value **= io_out(***io-object-name***,** *data-buffer***,** *dev-address***,** *count***);**

// i2c I/O object with the stop condition specified
return-value **= io_in(***io-object-name***,** *data-buffer***,** *dev-address***,** *count***,** *stop***);**
return-value **= io_out(***io-object-name***,** *data-buffer***,** *dev-address***,** *count***,** *stop***);**

Example

```
#define AD_ADDR 0x48 // address of the A/D converter
IO_8 i2c ioI2C;
unsigned int buffer[5];
unsigned int control;
boolean result;
. . .

when (...) {
  // Read the A/D converter.
  // First, write a control word byte.
  control = 0x04;
  result = io_out(ioI2C, &control, AD_ADDR, 1);

  // Next, perform a 5-byte read of the A/D converter.
  result = io_in(ioI2C, buffer, AD_ADDR, 5);
}
```

Magcard Bitstream Input

The **magcard_bitstream** I/O model provides the ability to read un-processed serial data streams from most magnetic stripe card readers in real time. This model can be used to read magnetic card data in either direction, forward or reverse, because the data does not need to follow any specific format.

This I/O model can read up to 65 535 bits of data, stored in 8192 bytes of data, from a magnetic stripe card reader.

This model applies to 3120 Power Line Smart Transceivers, to 3150 Power Line Smart Transceivers, to 3170 Power Line Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Figure 30 shows the magcard bitstream input.

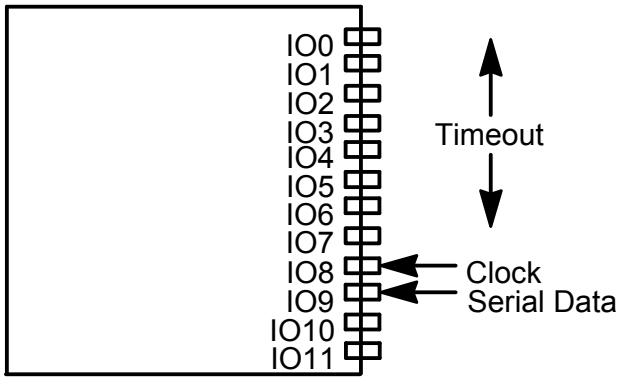


Figure 30. Magcard Bitstream Input

Programming Considerations

The data item unit is a single bit, and the *maxbits* and *count* values indicate the number of bits that can be read, or have been read, respectively. In case of a timeout, the *count* will be less than *maxbits*.

Syntax

IO_8 [**input**] **magcard_bitstream** [**timeout (***pin-nbr***)**] [**clockedge (+|-)**] [**invert**] *io-object-name***;**

IO_8

Specifies pin **IO_8**. The magcard bitstream input requires both pins **IO_8** and **IO_9**. Pin **IO_8** is the negative-going clock, **IO_9** is the serial data input.

timeout(*pin-nbr***)**

Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron Chip or Smart Transceiver checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a high logic level is sensed on the timeout pin, the transfer is terminated.

clockedge (+|-)

Specifies the polarity of the clock input signal. The default is **clockedge (-)**.

invert

Specifies that the data input signal is inverted. The default is no inversion.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *count***,** *maxbits***;**
unsigned short *input-buffer***[***buffer-size***];**

count **= io_in(***io-object-name***,** *input-buffer***,** *maxbits***);**

Example

```
IO_8 magcard_bitstream timeout(IO_7) ioMagcard;
const unsigned long maxbits = 64*8;
unsigned long count;
unsigned short input_buffer[64];

when (...) {
  count = io_in(ioMagcard, input_buffer, maxbits);
}
```

Magcard Input

The **magcard** I/O model is used to transfer synchronous serial data from an ISO 7811 Track 2 magnetic stripe card reader in real time.

See the **magtrack1** I/O model for track 1 compatible input, and the **magcard_bitstream** input model for a general-purpose magnetic card input.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

The data is presented as a data signal input on pin IO9, and a clock, or a data strobe, signal input on pin IO8. The data on pin IO9 is clocked on or just following the falling (negative) edge of the clock signal on IO8, with the least-significant bit (LSB) first. In addition, any one of the pins IO0 – IO7 can be used as a timeout pin to prevent lockup in case of abnormal abort of the input bit stream during the input process.

Up to 40 characters can be read at one time. Both the parity and the Longitudinal Redundancy Check (LRC) are checked by the Neuron Chip or Smart Transceiver.

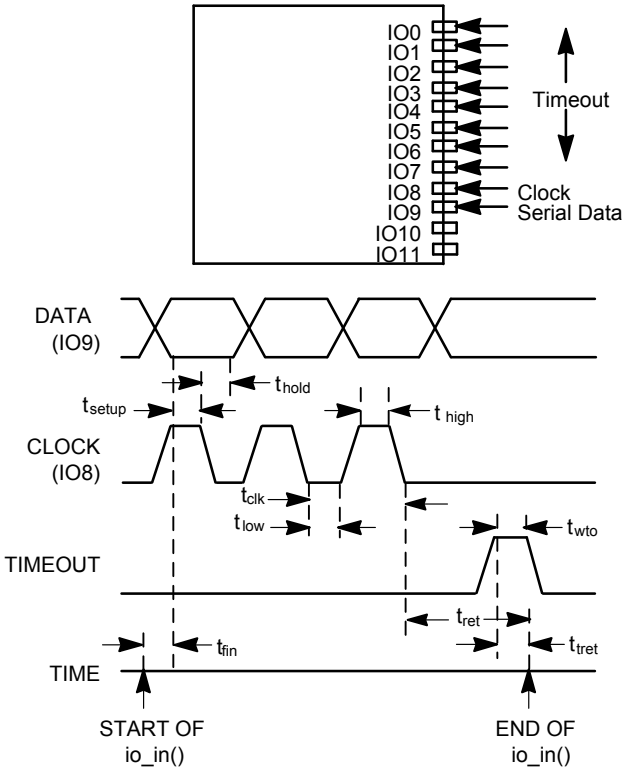


Figure 31. Magcard Input and Timing

Table 33. Magcard Input Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{fin}	I/O call to first clock input	—	45.0 µs	—
t_{hold}	Data hold	0 µs	—	—
t_{setup}	Data setup	0 µs	—	—
t_{low}	Clock low width	60 µs	—	—
t_{high}	Clock high width	60 µs	—	—
t_{wto}	Width of timeout pulse	60 µs	—	—
t_{clk}	Clock period	120 µs	—	—
t_{tret}	Return from timeout	21.6 µs	—	81.6 µs
t_{ret}	Return from function	—	—	301.8 µs

Programming Considerations

The magcard input model reads track 2 in the forward direction only (the **magcard_bitstream** input model can read in either direction). The data is presented as a data signal input on pin **IO_9**, and a clock, or data strobe, signal input on pin **IO_8**. The data on pin **IO_9** is clocked on or just following the falling edge of the clock signal on **IO_8**, with the least significant bit first.

Data is recognized as a series of 4-bit characters plus an odd parity bit per character. This process begins when the start sentinel (0x0B) is recognized, and continues until the end sentinel (0x0F) is recognized. No more than 40 characters, including the two sentinels, will be read. The data is stored as packed binary-coded decimal (BCD) digits in the buffer space pointed to by the buffer pointer argument to the **io_in( )** function with the parity bit stripped, and includes the start and end sentinel characters. This buffer should be 20 bytes long. The data is stored with the first character in the most significant nibble of the first byte in the buffer.

For magcard input, the **io_in( )** function requires a pointer to a data buffer, into which the series of BCD pairs are stored. The **io_in( )** function returns a **signed int** that contains the actual number of characters stored.

The parity of each character is checked. The longitudinal redundancy check (LRC) character, which appears just after the end sentinel, is also checked. If either of these tests fail, if more than 40 characters are being clocked in, or if the process aborts due to an input pin event, the **io_in( )** function returns the value (-1). The LRC character is not stored.

The magcard object optionally uses one of I/O pins **IO_0** through **IO_7** as a timeout/abort pin. Use of this feature is suggested because the **io_in( )** function updates the watchdog timer during clock wait states, and could result in a lockup if the card were to stop moving in the middle of the transfer process. If a high level is detected on the I/O timeout pin, the **io_in( )** function aborts. This input can be a one-shot timer counter output, an RC circuit, or a ~Data_valid signal from the card reader.

A Series 3100 Neuron Chip or Smart Transceiver with a 10 MHz input clock rate can process a bit rate of up to 8334 bps (at a bit density of 75 bits per inch, this is a card speed of 111 inches per second). Most magnetic card stripes contain a 15-bit sequence of zero data at the start of the card, allowing time for the application to start the card reading function. At 8334 bps, this period is about 1.8 ms. If the scheduler latency is greater than the 1.8 ms value, for example, due to application processing in another **when** task, the **io_in( )** function can miss the front end of the data stream.

Syntax

IO_8 [**input**] **magcard** [**timeout (***pin-nbr***)**] [**clockedge (+|-)**] [**invert**] *io-object-name***;**

IO_8

Specifies pin **IO_8**. Magcard input requires both pins **IO_8** and **IO_9**. Pin **IO_8** is the negative-going clock, **IO_9** is the serial data input.

timeout(*pin-nbr***)**

Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron Chip or Smart Transceiver checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a high logic level is sensed on the timeout pin, the transfer is terminated.

clockedge (+|-)

Specifies the polarity of the clock input signal. The default is **clockedge (-)**.

invert

Specifies that the data input signal is inverted. The default is no inversion.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***,** *input-buffer***[***buffer-size***];**

count = **io_in(***io-object-name*, *input-buffer***);**

Example

```
// In this example I/O pin IO_7 is connected to a
// ~Data_valid signal which is asserted low as long
// as a valid clock input is being generated by the
// reader device.

IO_8 input magcard timeout(IO_7) ioCardData;

// This next object allows monitoring of
// the ~Data_valid input signal.
IO_7 input bit ioDataValid;

int nibbles;
unsigned int buffer[20];
. . .

when (io_changes(ioDataValid) to 0) {
  nibbles = io_in(ioCardData, buffer);
}
```

Magtrack1 Input

The **magtrack1** I/O model is used to transfer synchronous serial data from an ISO 3554 track 1 magnetic stripe card reader.

See the **magcard** input model for track 2 compatible input, and the **magcard_bitstream** input model for a general-purpose magnetic card input.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

The data input is on pin IO9, and the clock, or data strobe, is presented as input on pin IO8. The data on pin IO9 is clocked in just following the falling edge of the clock signal on IO7, with the least-sigificant bit (LSB) first.

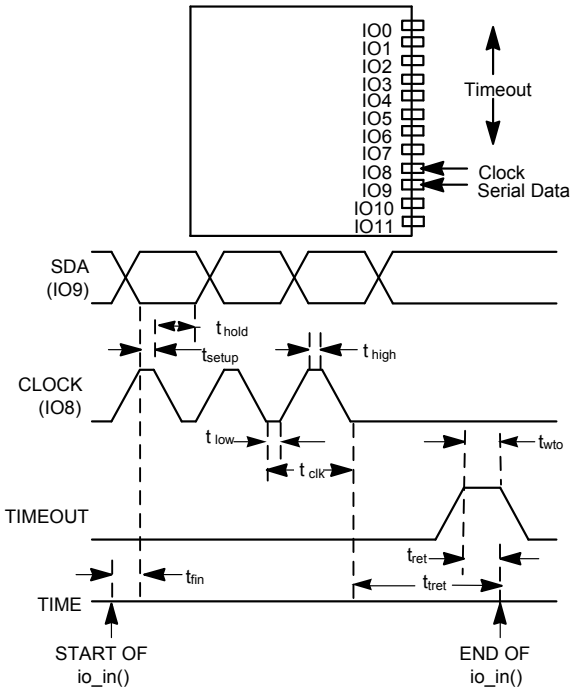


Figure 32. Magtrack1 Input and Timing

Table 34. Magtrack1 Input Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{fin}	I/O call to first clock input	—	45.0 µs	—
t_{hold}	Data hold	t_{low}	—	t_{clk}
t_{setup}	Data setup	0 µs	—	—
t_{low}	Clock low width	31 µs	—	—
t_{high}	Clock high width	31 µs	—	—
t_{wto}	Width of timeout pulse	60 µs	—	—
t_{clk}	Clock period	138 µs	—	—
t_{tret}	Return from timeout	21.6 µs	—	81.6 µs
t_{ret}	Return from function	—	—	301.8 µs

The minimum period for the entire bit cycle (t_{clk}) is greater than the sum of t_{low} and t_{high}. The t_{setup} and t_{hold} times should be such that the data is stable for the duration of t_{low}.

The magtrack1 input object optionally uses one of the I/O pins IO0 – IO7 as a timeout/abort pin. Use of this feature is suggested because the **io_in()** function updates the watchdog timer during clock wait states, and could result in a lockup if the card were to stop moving in the middle of the transfer process. If a logic 1 level is detected on the I/O timeout pin, the **io_in()** function aborts. This input can be a oneshot timer counter output, an R/C circuit, or a DATA_VALID~ signal from the card reader.

A PL Smart Transceiver with a clock rate of 10 MHz can process an incoming bit rate of up to 7246 bits/second when the strobe signal has a 1/3 duty cycle (t_{high} = 46 µs, t_{low} = 92 µs). At a bit density of 210 bits/inch, this translates to a card speed of 34.5 inches/second. The bit rate processing capability scales with PL Smart Transceiver input clock rate. Most magnetic card stripes contain a series of zero data at the start of the card, allowing time for the application to start the card reading function.

Programming Considerations

The data is presented as a data signal input on pin **IO_9**, and a clock, or data strobe, signal input on pin **IO_8**. The data on pin **IO_9** is clocked on or just following the falling edge of the signal on **IO_8**, least significant bit first.

Data is recognized in the International Air Transport Association (IATA) format as a series of 6-bit characters plus an odd parity bit per character. This process begins when the start sentinel (0x05) is recognized, and continues until the end sentinel (0x0F) is recognized. No more than 79 characters, including the two sentinels and the longitudinal redundancy check (LRC) character, are read. The data is stored in the buffer pointed to by the *input-buffer* pointer argument to the **io_in( )** function. The data is stored without the parity bit, and the data includes the start and end sentinel characters. This buffer should be 78 bytes long.

For **magtrack1** input, the **io_in( )** function requires a pointer to a data buffer, into which the series of 6-bit characters are stored. The **io_in( )** function returns a **signed int** that contains the actual number of bytes stored.

The parity of each character is checked. The LRC character, which appears just after the end sentinel, is also checked. If either of these tests fail, if more than 79 characters are being clocked in, or if the process aborts due to an input pin event (see below), the **io_in( )** function returns the value (-1) as an error indication. The LRC character is not stored.

The **magtrack1** object optionally uses one of I/O pins **IO_0** through **IO_7** as a timeout or abort pin. Use of this feature is suggested because the **io_in( )** function updates the watchdog timer during clock wait states, and could result in a lockup if the card were to stop moving in the middle of the transfer process. If a high level is detected on the I/O timeout pin, the **io_in( )** function aborts. This input can be a one-shot timer counter output, an RC circuit, or a ~Data_valid signal from the card reader.

Syntax

IO_8 [**input**] **magtrack1** [**timeout (***pin-nbr***)**] [**clockedge (+|-)**]
[**invert**] *io-object-name***;**

IO_8

Specifies pin **IO_8**. Magtrack1 input requires both pins **IO_8** and **IO_9**. Pin **IO_8** is the negative-going clock, **IO_9** is the serial data input.

timeout(*pin-nbr***)**

Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron Chip or Smart Transceiver checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a high logic level is sensed on the timeout pin, the transfer is terminated.

clockedge (+|-)

Specifies the polarity of the clock input signal. The default is **clockedge (-)**.

invert

Specifies that the data input signal is inverted. The default is no inversion.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***;**
unsigned int *input-buffer***[***buffer-size***];**

count **= io_in(***io-object-name***,** *input-buffer***);**

Example

```
// In this example I/O pin IO_7 is connected to a
// ~Data_valid signal which is asserted low as long
// as a valid clock input is being generated by the
// reader device.
IO_8 input magtrack1 timeout(IO_7) ioCardData;

// This next object allows monitoring of the
// ~Data_valid input signal.
IO_7 input bit ioDataValid;

int read;
unsigned int buffer[78];
. . .

when (io_changes(ioDataValid) to 0) {
  read = io_in(ioCardData, buffer);
}
```

Neurowire Input/Output

The **neurowire** I/O model implements a full-duplex synchronous transfer of data to a peripheral device, and is used to transfer data using a fully synchronous serial data format.

Neurowire I/O is useful for external devices, such as analog-to-digital (A/D) and digital-to-analog (D/A) converters, and display drivers incorporating serial interfaces that conform with National Semiconductor's Microwire™ serial interface or Motorola's Serial Peripheral Interface (SPI).

Important: The Neurowire I/O model is provided for legacy support. Echelon recommends using the hardware SPI I/O model instead of the legacy software I/O model (see *SPI Input/Output*). The hardware SPI interface provides higher performance with lower software overhead.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

The **neurowire** I/O model can operate as the master (drive a clock out) or as the slave (accept a clock in). In both master and slave modes, up to 255 bits of data can be transferred at a time. The Neurowire I/O suspends application processing until the operation is completed.

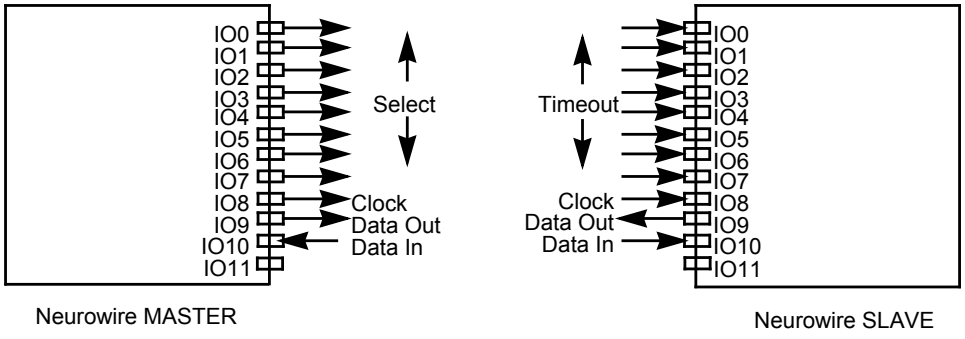


Figure 33. Neurowire I/O

Neurowire Master Mode

In Neurowire master mode, pin IO8 is the clock (driven by the Smart Transceiver), IO9 is the serial data output, and IO10 is the serial data input. Serial data is clocked out on pin IO9 at the same time as data is clocked in from pin IO10. Data is clocked by the rising edge of the clock signal by default. In addition, one or more of the IO0 – IO7 pins can be used as a chip select, allowing multiple Neurowire devices to be connected on a three-wire bus. The clock rate can be specified as 1 kbps, 10 kbps, or 20 kbps for a Series 3100 device with an input clock rate of 10 MHz, or as 16 kbps, 160 kbps, and 320 kbps for a Series

5000 or Series 6000 device with a system clock rate of 80 MHz; these scale proportionally with input clock.

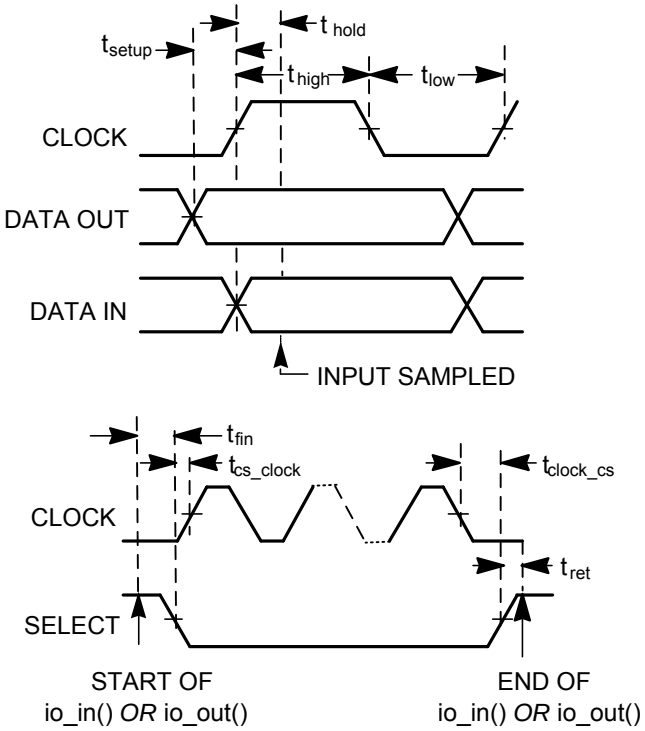


Figure 34. Neurowire Master Timing

Table 35. Neurowire Master Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to CS~ active	69.9 µs
t_{ret}	Return from function	7.2 µs
t_{hold}	Active clock edge to sampling of input data 20 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 11.4 µs 53.4 µs 960.6 µs
t_{high}	Period, clock high (active clock edge = 1) 20 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 25.8 µs 67.8 µs 975.0 µs
t_{low}	Period, clock low (active clock edge = 1)	33.0 µs
t_{setup}	Data output stable to active clock edge	5.4 µs
t_{cs_clock}	Select active to first active clock edge	91.2 µs

Symbol	Description	Typical at 10 MHz
t_{clock_cs}	Last clock transition to select inactive	81.6 µs
f	Clock frequency = $1/(t_{high} + t_{low})$ 20 kbps bit rate 10 kbps bit rate 1 kbps bit rate	 17.0 kHz 9.92 kHz 992 Hz

Neurowire Slave Mode

In Neurowire slave mode, pin IO8 is the clock (driven by the external master), IO9 is the serial data output, and IO10 is the serial data input. Serial data is clocked out on pin IO9 at the same time as data is clocked in from pin IO10. Data is clocked by the rising edge of the clock signal (default), which can be up to 18 kbps for a Series 3100 device at 10 MHz. This data rate scales with Smart Transceiver input clock rate. One of the IO0 – IO7 pins can be designated as a timeout pin. A logic 1 level on the timeout pin causes the Neurowire slave I/O operation to be terminated before the specified number of bits has been transferred. This prevents the Smart Transceiver watchdog timer from resetting the chip in the event that fewer than the requested number of bits are transferred by the external clock.

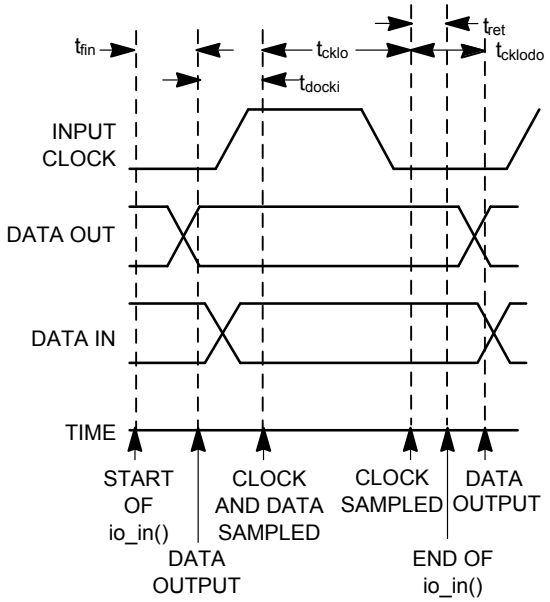


Figure 35. Neurowire Slave Timing

Table 36. Neurowire Slave Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to data bit out	41.4 µs
t_{ret}	Return from function	19.2 µs
t_{docki}	Data out to input clock and data sampled	4.8 µs
t_{cklo}	Data sampled to clock low sampled	24.0 µs
t_{cklodo}	Clock low sampled to data output	25.8 µs
f	Clock frequency (max)	18.31 kHz

The algorithm for each bit of output/input for the Neurowire slave objects is described below. In this description, the default active clock edge (positive) is assumed; if the invert keyword is used, all clock levels stated should be reversed.

1. Set IO9 to the next output bit value.
2. Test pin IO8, the clock input, for a high level (to test for the rising edge of the input clock). If the input clock is still low, sample the timeout event pin and abort if high.
3. When the input clock is high, store the next data input bit as sampled on pin IO10.
4. Test the input clock for a low input level (to test for the falling edge of the input clock). If the input clock is still high, sample the timeout event pin and abort if high.
5. When the input clock is low, return to step 1 if there are more bits to be processed.
6. Else return the number of bits processed.

When either clock input test fails (that is, the clock is sampled *before* the next transition), there is an additional timeout check time of 19.8 µs (wait for clock high) or 19.2 µs (wait for clock low) added to that stage of the algorithm.

The chip select logic for the Neurowire slave can be handled by the user through a separate bit input object, along with an appropriate handshaking algorithm implemented by the user application program. To prevent unnecessary timeouts, the setup and hold times of the chip select line, relative to the start and end of the external clock, must be satisfied.

The timeout input pin can either be connected to an external timer or to an output pin of the Smart Transceiver that is declared as a oneshot object.

Programming Considerations

The Neurowire I/O object can be configured in master mode or slave mode. The primary difference between master and slave modes is that the clock signal is an

output for the master mode, and an *input* for the slave mode. Data is shifted in at the same time as data is shifted out.

In Neurowire master mode, one or more of the pins **IO_0** through **IO_7** can be used as a chip select, allowing multiple Neurowire devices to be connected on a 3-wire bus. The clock rate can be specified as 1, 10, or 20 kbps for a Series 3100 Neuron Chip or Smart Transceiver with an input clock rate of 10 MHz, or as 16, 160, or 320 kbps for a Series 5000 or Series 6000 device with an input clock of 80 MHz; these scale proportionally with input clock.

In Neurowire slave mode, one of the **IO_0** through **IO_7** pins can be designated as a timeout pin. A logic one level on the timeout pin causes the Neurowire slave I/O operation to be terminated before the specified number of bits has been transferred. This prevents the Neuron Chip or Smart Transceiver watchdog timer from resetting the chip in the event that fewer than the requested number of bits are transferred by the external clock.

In both master and slave modes, up to 255 bits of data can be transferred at a time. Neurowire I/O suspends application processing until the operation is complete.

For Neurowire input/output, the **io_in( )** and **io_out( )** functions require a pointer to the data buffer as the **input_value** and **output_value**. Because Neurowire I/O is bidirectional, input and output occur at the same time, and therefore, the calls **io_in( )** and **io_out( )** are equivalent. Use of either call initiates a bidirectional transfer. Data is transmitted 8 bits at a time, most significant bit first. The clock edge used to clock the data is specified by the **clockedge** parameter. Data is also then transferred into the same buffer pointed to by **input_value** or **output_value**, most significant bit first, following the clock edge, overwriting the original contents of the buffer. If the number of bits to be transferred is not a factor of eight as defined by *count*, the last byte transferred into the buffer will contain undefined data bit values in the remaining (unfilled) bit locations.

When using multiple serial or Neurowire I/O objects that have differing bit rates, the following compiler directive must be used: **#pragma enable_multiple_baud**. This pragma must appear prior to the use of any I/O function, such as **io_in( )** or **io_out( )**.

For examples on the use of the Neurowire input/output model, see the following engineering bulletins: *Driving a Seven Segment Display with the Neuron Chip* (part no. 005-0014-01) and *Analog-to-Digital Conversion with the Neuron Chip* (part no. 005-0019-01).

Syntax

IO_8 neurowire master | slave [**select (***pin-nbr***)**] [**timeout (***pin-nbr***)**] [**kbaud (***const-expr***)**] [**clockedge (+|-)**] *io-object-name***;**

IO_8

Specifies pin **IO_8**. The Neurowire object requires pins **IO_8** through **IO_10** and must specify **IO_8**. The **select** pin must be one of **IO_0** through **IO_7**. Pin **IO_8** is the clock, driven by the Neuron Chip or Smart Transceiver (or the external master). Pin **IO_9** is serial data output and **IO_10** is serial data input. Up to 255 bits of data can be transferred at a time.

master

Specifies that the Neuron Chip or Smart Transceiver provides the clock on pin **IO_8**, which is configured as an output pin.

slave

Specifies that the Neuron Chip or Smart Transceiver senses the clock on pin **IO_8**, which is configured as an input pin. The maximum input clock rate is 72 kbps, 50/50 duty cycle, for a Series 3100 device with a 40 MHz input clock. This rate scales proportionally to the input clock.

select (*pin-nbr***)**

Specifies the chip select pin for a Neurowire master. This keyword is applicable to master mode only.

Before the data transfer, the chip select pin goes low; after the data transfer, the select pin goes high. In addition to this declaration with the **select** keyword, the chip select pin must also be declared with a bit output object, unless there is no chip select pin in use. If no chip select pin is in use, the pin declared as the select pin can also be declared as any of the allowable input objects for that pin (for example, bit input).

timeout (*pin-nbr***)**

Specifies the optional timeout signal pin for a Neurowire slave, in the range of **IO_0** to **IO_7**. This keyword is applicable to slave mode only.

When a timeout signal pin is used, the Neuron firmware checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a logic level of 1 is sensed, the transfer is terminated. This allows the use of an external timeout signal, or an internally generated timeout signal, such as an inverted oneshot output object, to limit the duration of the transfer. The watchdog timer is updated by this object with every falling edge of the clock on pin **IO_8**.

kbaud (*const-expr***)**

Specifies the bit rate for a Neurowire master. The expression *const-expr* can evaluate to 1, 10, or 20. The default is 20. The firmware uses this value as a multiplier based on the Series 3100 input clock or Series 5000 and 6000 system clock. For example, for a Series 3100 device at 10 MHz, **kbaud(10)** yields 10 kbps; for a Series 5000 or Series 6000 device at 10 MHz, **kbaud(10)** yields 20 kbps. The bit rate scales proportionally with the input or system clock.

Not used for a Neurowire slave.

clockedge (+|-)

Specifies the polarity of the clock signal. The default is a rising edge clock, **clockedge (+)**. Specifying **clockedge (-)** causes the data to be clocked at the falling edge of the clock signal.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***,** *io-buffer***[***buffer-size***];**

io_out(*io-object-name***,** *io-buffer***,** *count***);**

Example

```
IO_8 neurowire master select(IO_2) ioDisplay;
IO_2 output bit ioDisplaySelect = 1;  // active low

// 8 bits=>display config reg
unsigned int config = 0x01;

// 24 bits=>display data reg
unsigned int data[3];

when (...) {
  io_out(ioDisplaySelect, 0);
  config = 0x01;
  io_out(ioDisplay, &config, 8);
  data[0] = 0x80;
  data[1] = 0xAB;
  data[2] = 0xCD;
  io_out(ioDisplay, data, 24);
  io_out(ioDisplaySelect, 1);
}
```

SCI (UART) Input/Output

You can use the hardware Serial Communications Interface (SCI) I/O model in applications that you develop for Smart Transceivers or Neuron Chips with integrated universal asynchronous receiver/transmitter (UART) hardware such as the PL 3120 Smart Transceiver, PL 3150 Smart Transceiver, PL 3170 Smart

Transceiver, Series 5000 device, or a Series 6000 device. SCI is an asynchronous serial communication interface that is compatible with EIA-232 serial interfaces (with the exception of voltage levels). External driver hardware can be used to adjust the voltage levels. The SCI I/O model uses the UART hardware and interrupt capability in designated Neuron Chips and Smart Transceivers. You cannot use both hardware SCI and hardware SPI I/O in the same application.

The hardware SCI I/O object does not include any form of hardware flow control, such as CTS/RTS flow control. If your application requires flow control, you must implement some form of handshaking in your application.

This model applies to 3120 Power Line Smart Transceivers, 3150 Power Line Smart Transceivers, 3170 Power Line Smart Transceivers, Series 5000 Neuron Processors and Smart Transceivers, and Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Pins IO8 and IO10 can be configured as asynchronous SCI input and output lines, respectively. The SCI object model supports the following bit rates for half-duplex transfers: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200 bits per second. The effective transmitted data rate for half-duplex transfers corresponds to the bit rate at all speeds. There are no inter-byte idle periods, and the bit rate for the input and output can not be independently specified.

For full-duplex transfers, when data is being received and transmitted at the same time, the effective bit rate will be 60% at 57600 bits per second, and 30% at 115200 bits per second. All other bit rates specified above for half-duplex transfers are also supported for full-duplex transfers. No errors are introduced (other than inter-byte spacing of transmitted data) under these conditions.

For 6.5536 MHz operation (Series 3100 power line Smart Transceivers), the bit rates are limited to a maximum of 19200 bits per second for both half and full-duplex transfers.

The frame format is one start bit, eight data bits, and one stop bit plus a parity bit or two stop bits. Up to 255 output bytes and 255 input bytes can be transferred at a time. If an input stop bit has the wrong polarity, the interface attempts to recover and re-synchronize. However, a framing error is flagged in the status register. If necessary, the application code can use other bit I/O pins for flow-control handshaking.

This I/O model depends on interrupts to receive data at high speed. After reception has been set up, control is returned to the application immediately, and the application needs to poll the I/O model for reception completion. Reception can be suspended and resumed by disabling and enabling interrupts. Turning off interrupts might be required when going off-line, or for ensuring that other time-critical application execution is not disturbed by background interrupts. Additionally, SCI reception can also be aborted. Note that for Series 3100 devices, sustained reception at 115200 bps can starve the application processor. Care must be given to allow the Smart Transceiver to process received bytes in a timely manner and update the watchdog timer.

However, for the Series 3100 data transmission is *not* handled by interrupts; control is returned to the application only after the last byte has been placed in the transmission shift register. It is important to note that if previously set up,

reception interrupts work even while transmission is taking place, thus providing the full duplex interface.

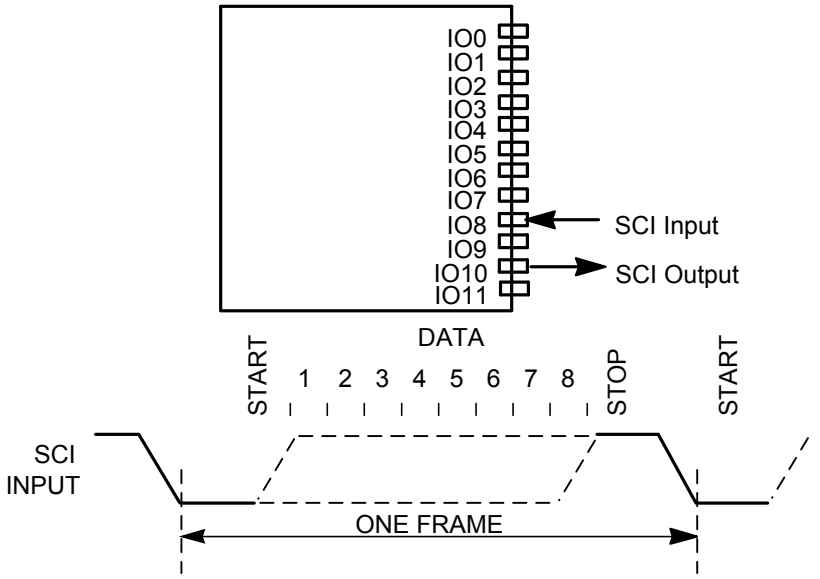


Figure 36. SCI I/O and Timing

Programming Considerations

You can enable and disable SCI interrupts. For example, you can turn off interrupts when going offline, or to assure that other time-critical application functions are not disturbed by SCI interrupts. The SCI interrupt signal is used by the firmware driver for the SCI I/O object. It is not directly accessible by the application program.

The SCI interrupt is enabled by default. For Series 3100 devices, the **io_idis()** function disables I/O interrupts. The function has the following signature:

void io_idis(void);

For Series 3100 devices, the **io_iena()** function enables I/O interrupts. The function has the following signature:

void io_iena(void);

For Series 5000 and Series 6000 devices, you cannot disable the SCI interrupt.

To cancel an SCI operation currently in progress, use the **sci_abort()** function rather than disabling interrupts.

When using hardware SCI I/O, the Neuron C application must specify the clock frequency that drives the on-chip SCI UART. To specify this frequency, the I/O clock rate, use the following compiler directive:

#pragma specify_io_clock *clock-rate*

The *clock-rate* value must be one of the following quoted strings: “20 MHz”, “10 MHz”, “6.5536 MHz”, “5 MHz”, or “2.5 MHz”. If the pragma-specified clock rate does not match the Series 3100 physical clock frequency or the Series 5000 or Series 6000 system clock rate, the Neuron Exporter component of the

NodeBuilder FX Development Tool reports an error to prevent generation of an incorrect communications bit rate.

If you do not specify the **#pragma specify_io_clock** compiler directive, the compiler uses a default I/O clock rate of 10 MHz.

Syntax

IO_8 sci [**baud (***const-expr***)**] [**twostopbits**] [**parity (***parity-expr***)**] *io-object-name***;**

baud (*const-expr***)**

Optionally specifies the serial bit rate through use of the enumeration values found in the **<io_types.h>** include file. These enumeration values are:

- SCI_300
- SCI_600
- SCI_1200
- SCI_2400
- SCI_4800
- SCI_9600
- SCI_19200
- SCI_38400
- SCI_57600
- SCI_115200

The enumeration values select serial bit rates of 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200, respectively. This clause is optional in the declaration, but, if omitted, the **io_set_baud( )** function must be used.

These bit rates are accurate for devices running at input or system clock rates that are a multiple of 2.5 MHz. Devices using the 6.5536 MHz clock rate can be inaccurate (off by more than 3%) at baud rates of 38400 and higher because the bit rate divisor has been optimized for input clocks that are a multiple of 2.5 MHz.

twostopbits

Set this option to use two stop bits. By default, there is one stop bit.

You cannot use two stop bits if you also specify even or odd parity. That is, to use two stop bits, you must specify **__parity(none)**.

This keyword is not supported for Series 5000 or Series 6000 devices.

__parity (*parity-expr***)**

Specifies optional parity for the serial communications. A parity bit ensures that the number of "1" bits between the start and stop bits is always even or odd. Using parity allows you to perform error checking of the communications channel.

The *parity-expr* can be one of the following values: none, odd, or even. **__parity(none)** is the default.

The use of parity is supported for the Series 5000 and Series 6000 devices.

io-object-name

Specifies a name for the I/O object, in the ANSI C format for variable identifiers.

You can call the **io_set_baud(***io-obj-name***,** *rate***)** function to change the bit rate for the SCI interface. The specified *rate* must be one of the enumeration values listed above.

Usage

unsigned short *buffer-size***;**
unsigned short *buffer***[***buffer-size***];**

unsigned short io_in_request(*io-object-name, buffer***,** *buffer-size***);**
unsigned short io_out_request(*io-object-name, buffer***,** *buffer-size***);**

unsigned short io_in_ready(*io-object-name***);**
unsigned short io_out_ready(*io-object-name***);**
unsigned short sci_get_error(*io-object-name***);**
void sci_abort(*io-object-name***);**

The SCI I/O object uses pins **IO_8** for RX data (in) and **IO_10** for TX data (out).

The **io_in( )** and **io_out( )** functions are not available with the hardware SCI model. Instead, use the **io_in_request()** and **io_out_request()** functions:

- For input, call **io_in_request(***io-object-name***, void ****buf***, unsigned** *len***)** to set up and initiate an input operation.
- For output, call **io_out_request(***io-object-name***, void ****buf***, unsigned** *len***)** to set up and initiate an output operation.

A call to either **io_in_request( )** or **io_out_request( )** clears any previous SCI error code – see **sci_get_error( )**.

You can use the **io_in_ready(***io-object-name***)** and **io_out_ready(***io-object-name***)** event functions to test the state of the SCI interface. You can use these events to determine when the transmission is complete. The **io_out_ready** event returns **TRUE** after the Neuron firmware loads the output data into the hardware UART. The UART then continues transmitting the remaining data. The **io_in_ready** event returns the number of bytes read in as an **unsigned short**, so when this value matches the *len* parameter from the call to **io_in_request( )** the input operation is complete.

You can use the **sci_get_error(***io-object-name***)** function to test for SCI errors, including parity errors. Calling the **sci_get_error( )** function clears the SCI error code after it is returned. This function returns a cumulative OR of the following bits that reflect data errors:

0x02 Parity error

0x04 Framing error

0x08 Noise detected

0x10 Receive overrun detected

You can use the **sci_abort(***io-object-name***)** function to terminate any reception in progress. After an abort, the **io_in_ready( )** function returns the number of characters read up to the abort.

Example

```
#pragma specify_io_clock "10 MHz"
IO_8 sci twostopbits baud(SCI_2400) ioSci;
unsigned short buffer[20];

when (...) {
  io_set_baud(ioSci, SCI_38400); // Optional baud change
  io_out_request(ioSci, buffer, 20);
}

when (io_out_ready(ioSci)) {
  unsigned short sciError;
  sciError = sci_get_error(ioSci));
  if (sciError) {
    // Process SCI error
    ...
  }
  else {
    // Process end of SCI transmission
    ...
  }
}
```

Serial Input/Output

The **serial** I/O model is used to transfer data using an asynchronous serial data format, such as EIA-232 (formerly RS-232) and Serial Communications Interface (SCI) communications. This I/O model is useful for devices such as intelligent LCD displays, terminals, modems, and computer serial interfaces. External driver circuitry is required to adjust the signal voltage levels to be compatible to the EIA-232 standard.

Important: The serial I/O model is provided for legacy support. Echelon recommends using the SCI (UART) model instead of the legacy software serial I/O model (see *SCI (UART) Input/Output*). The hardware UART provides higher performance with lower software overhead.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Pin IO8 can be configured as an asynchronous serial input line, and pin IO10 can be configured as an asynchronous serial output line. The bit rates for input and for output can be independently specified to be 600, 1200, 2400, or 4800 bits/second for a Series 3100 device with a 10 MHz input clock rate, or 4800, 9600, 19200, 38400, or 76800 bits/second for a Series 5000 or Series 6000 device with an 80 MHz system clock. The data rate scales proportionally to the input clock rate.

Either a serial input or a serial output operation (but not both) can be in effect at any one time. The interface is half-duplex only. This function suspends application processing until the operation is completed. If the stop bit has the wrong polarity (it should be a 1), the input operation is terminated with an error. Parity is not supported for this model. The application code can use bit I/O pins for flow control handshaking if required.

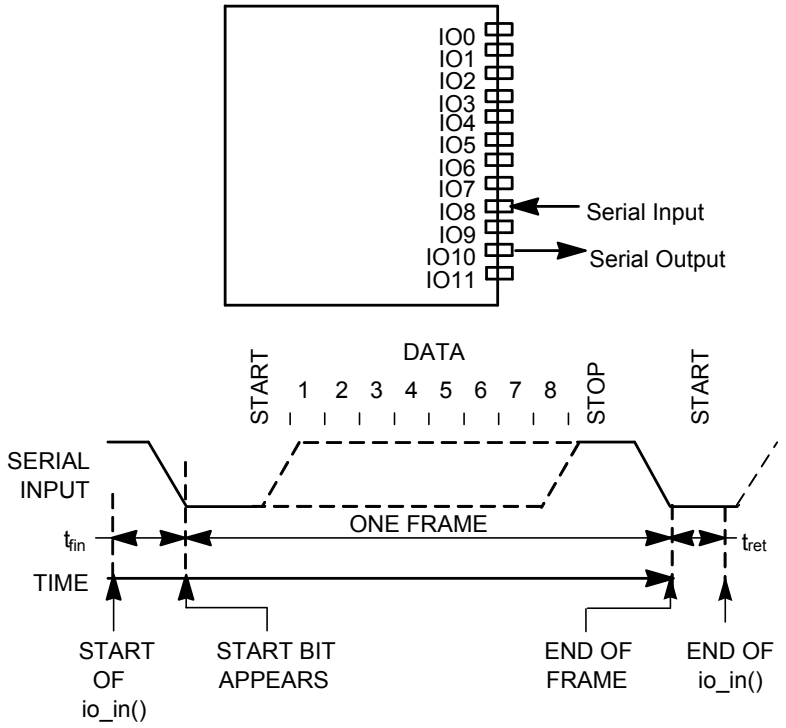


Figure 37. Serial Input and Timing

Table 37. Serial Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to input sample Min (first sample) Max (timeout)	 67 µs 20 byte frame
t_{ret}	Return from function	10 µs

The duration of this function call is a function of the number of data bits transferred and the transmission bit rate. t_{fin} (max) refers to the maximum amount of time this function waits for a start bit to appear at the input. After this time, the function returns a 0 as data. t_{fin} (min) is the time to the first sampling of the input pin. For example, the timeout period at 2400 bits/second is (20 x 10 x 1/2400) + t_{fin} (min).

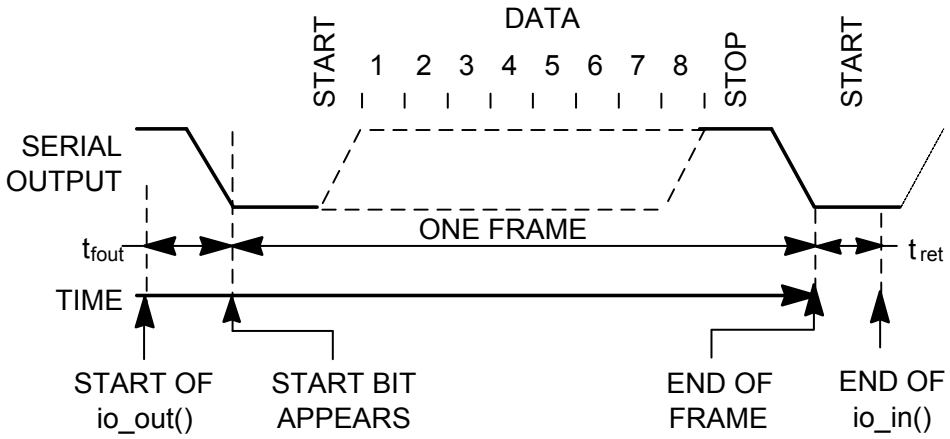


Figure 38. Serial Output Timing

Table 38. Serial Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to start bit	79 µs
t_{ret}	Return from function	10 µs

The duration of this function call is a function of the number of data bits transferred and the transmission bit rate. For example, to output 100 bytes at 300 bits/second requires a time duration of (100 x 10 x 1/300) + t_{fout} + t_{ret}.

Programming Considerations

The format for data frame transfer is fixed: one start bit, followed by eight data bits (least significant bit first), followed by one stop bit. The input serial I/O object waits for the start of the data frame to be received for up to the time it would take to receive 20 characters before timing out and returning a zero. Input is terminated when either the total count in bytes is received, or the amount of time it would take to receive 20 characters has passed with no data received. The input serial I/O model stops receiving data on invalid stop bit. At 2400 bps, the input timeout is 83 ms.

Unlike the SCI and SPI I/O models, which are available only for certain Neuron Chip models, the serial input/output model does not require special hardware and is available for all Neuron Chip models.

Both serial input and output models are purely software I/O models, with no hardware support other than the physical I/O pins. The serial stream is read in and transmitted out using CPU timing. See the **sci** I/O model for an equivalent I/O object that uses UART hardware on certain Smart Transceivers and Neuron Chips. The following issues should be considered when using the **serial** I/O model:

- The **io_out( )** function is a blocking function, so the function does not return until the entire data set is transmitted.

- Serial input can only work successfully if the application is responsive enough to capture the start bit of the first byte received. Usually the best way to succeed with the serial input model is to employ bi-directional handshaking using two additional I/O pins, so that the sender can coordinate the transfer with the Neuron C application. If this is not possible, the serial input can be monitored with a **when(io_changes(***io-object-name***))** statement or an I/O interrupt task, however, you must ensure that the **io_in( )** function is called less than 25% into the start bit. For example, the start bit is approximately 4.2 ms at 2400 bps. For reliable reception of a 2400 bps start bit, the **io_in( )** function must be called within 1 ms of the beginning of the start bit. The minimum scheduler latency is approximately 0.24 ms with for a Series 3100 device with a 40 MHz input clock, and is typically longer depending on the number and type of **when** clauses in the application. See *I/O Timing Issues* for a description of the scheduler-related I/O timing. Scheduler latencies do not affect an I/O interrupt task; see the *Neuron C Programmer's Guide* for more information about timing of application-defined interrupt tasks.

When using multiple serial I/O devices that have differing bit rates, you must use the **#pragma enable_multiple_baud** compiler directive. This pragma must appear prior to the use of any I/O function, such as, **io_in( )** or **io_out( )**.

For serial input/output, the **io_in( )** and **io_out( )** functions require a pointer to the data buffer as the **input_value** and **output_value**. The **io_in( )** function returns an **unsigned short int** that contains the count of the actual number of bytes received. See the *EIA-232C Serial Interfacing with the Neuron Chip* engineering bulletin (part no. 005-0008-01) for more information.

The serial input model provides only one bit of buffering and a maximum speed of 4800 bps. For higher bit rates, use a Smart Transceiver or Neuron Chip with integrated UART hardware, such as the PL 3120 Smart Transceiver, PL 3150 Smart Transceiver, PL 3170 Smart Transceiver, Series 5000 device, or a Series 6000 device. Alternatively, for bit rates up to 115.2 kbps, and 16 bytes of buffering, consider using the PSG-20 or PSG/3 programmable serial gateway devices. See the *LTS-20 LonTalk Serial Adapter and PSG-20 User's Guide* for more details.

Syntax

pin **input serial** [**baud (***const-expr***)**] *io-object-name***;**

pin **output serial** [**baud (***const-expr***)**] *io-object-name***;**

pin

An I/O pin. Serial input requires one pin and must specify **IO_8**. Serial output also requires one pin and must specify **IO_10**.

baud (*const-expr***)**

Specifies the bit rate. The expression *const-expr* can be 600, 1200, 2400, or 4800. The default is 2400. The firmware uses this value as a multiplier based on the Series 3100 input clock or Series 5000 system clock. For example, for a Series 3100 device at 10 MHz, **baud(4800)** yields 4800 bps; for

a Series 5000 or Series 6000 device at 10 MHz, **baud(4800)** yields 9600 bps. The baud rate scales proportionally with the input or system clock.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***,** *input-buffer***[***buffer-size***],** *output-buffer***[***buffer-size***];**

count **= io_in(***io-object-name***,** *input-buffer***,** *count***);**
io_out(*io-object-name***,** *output-buffer***,** *count***);**

Serial Input Example

```
IO_8 input serial ioKeyboard;
char buffer[20];
unsigned int chars;

when (...) {
  chars = io_in(ioKeyboard, buffer, 20);
}
```

Serial Output Example

```
IO_10 output serial ioDisplay;

when (...) {
  io_out(ioDisplay, “Hello world.\r\n”, 14);
}
```

SPI Input/Output

You can use the hardware Synchronous Peripheral Interface (SPI) I/O model in applications that you develop for Smart Transceivers or Neuron Chips with integrated SPI hardware such as the PL 3120 Smart Transceiver, PL 3150 Smart Transceiver, PL 3170 Smart Transceiver, Series 5000 or Series 6000 device. SPI is a full-duplex synchronous serial communication interface initially advanced by Motorola, but now available on a wide variety of devices. The **spi** I/O model uses the SPI hardware and the I/O interrupt capability in designated Neuron Chips and Smart Transceivers. You cannot use both hardware SCI and hardware SPI I/O in the same application.

The hardware SPI I/O model does not include any form of hardware flow control such as CTS/RTS or TREQ/R/W flow control. If your application requires flow control, you must implement some form of handshaking in your application.

This model applies to 3120 Power Line Smart Transceivers, 3150 Power Line Smart Transceivers, 3170 Power Line Smart Transceivers, Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Pins IO8, IO9 and IO10 can be configured as a SPI port. The directions of the pins vary with the configuration:

- In master mode, pin IO8 is the clock (driven by the Smart Transceiver), IO9 is serial data input (Master In Slave Out or MISO), and IO10 is serial data output (Master Out Slave In or MOSI).
- In slave mode, pin IO8 is the clock input, IO9 is serial data output (MISO), and IO10 is serial data input (MOSI).

The declaration of the SPI I/O object supports several modifiers, including **neurowire**. If the **neurowire** keyword is used, the pins assume a Neurowire-compatible direction in which IO9 is always output and IO10 is always input. Serial data is clocked out on the output pin at the same time as it is clocked in on the input pin. In SPI master mode, no other masters are allowed on the bus. IO7 can be used as a select pin in slave mode, allowing the Smart Transceiver to coexist with other slave mode devices on a 3-wire bus. A logic one level on the select line disables the output drivers of the output pins and puts them in a high impedance state.

If the Smart Transceiver is the only slave device on the SPI bus, and the master device does not drive the Slave Select (SS~) signal (the signal is disabled), then you can initialize the IO7 pin to a value of 1 and use it as an input pin:

- Pin IO7 should be declared as an input pin and externally grounded.

  OR

- Pin IO7 must be declared in the following order:

```
IO_7 output bit io7out = 1;     // initialize to '1'
IO_7 input bit io7in;
```

Note that SS~ should be used whenever possible to ensure proper synchronization and recovery in the event of framing errors from the master device.

The bit rates supported by the SPI port are summarized in **Table 39** through **Table 42**.

Table 39. SPI Master Mode for Series 3100 Power Line Devices

Clock	Bit Rate for 10 MHz	Bit Rate for 6.5536 MHz
7	19.531 kbps	12.8 kbps
6	39.063 kbps	25.6 kbps
5	78.125 kbps	51.2 kbps
4	156.250 kbps	102.4 kbps
3	312.500 kbps	204.8 kbps
2	625.000 kbps	409.6 kbps

Clock	Bit Rate for 10 MHz	Bit Rate for 6.5536 MHz
1	1250.000 kbps	819.2 kbps
0	2500.000 kbps	1638.4 kbps
Note: For Clock 5 and higher bit rates, the bit rate shown is the peak rate. The data is burst out in pairs of bytes, and the overall average data rate is limited to approximately 40 kbps and 25 kbps for 10 MHz and 6.5536 MHz input clocks, respectively.		

Table 40. SPI Master Mode for Series 5000/6000 Devices

Clock	Bit Rate for 80 MHz	Bit Rate for 40 MHz	Bit Rate for 20 MHz	Bit Rate for 10 MHz	Bit Rate for 5 MHz
7	156.25 kbps	78.125 kbps	39.063 kbps	19.531 kbps	9.765 kbps
6	312.5 kbps	156.25 kbps	78.125 kbps	39.063 kbps	19.531 kbps
5	625 kbps	312.5 kbps	156.25 kbps	78.125 kbps	39.063 kbps
4	1.25 Mbps	625 kbps	312.5 kbps	156.25 kbps	78.125 kbps
3	2.5 Mbps	1.25 Mbps	625 kbps	312.5 kbps	156.25 kbps
2	5 Mbps	2.5 Mbps	1.25 Mbps	625 kbps	312.5 kbps
1	10 Mbps	5 Mbps	2.5 Mbps	1.25 Mbps	625 kbps
0	20 Mbps	10 Mbps	5 Mbps	2.5 Mbps	1.25 Mbps
Note: For Clock 5 and higher bit rates, the bit rate shown is the peak rate. The data is burst out in pairs of bytes, and the overall average data rate is limited to approximately 430 kbps for an 80 MHz system clock.					

Table 41. SPI Slave Mode for Series 3100 Power Line Devices

Parameter	Value for 10 MHz	Value for 6.5536 MHz
Max burst rate	1250 kbps	819.2 kbps
Max burst size	2 bytes	2 bytes
Min burst spacing (from start of one burst to next)	400 µs	640 µs
Max sustained data rate	40 kbps	25 kbps

Table 42. SPI Slave Mode for Series 5000/6000 Devices

Parameter	Value for 80 MHz	Value for 40 MHz	Value for 20 MHz	Value for 10 MHz	Value for 5 MHz
Max burst rate	10 Mbps	5 Mbps	2.5 Mbps	1.25 Mbps	625 kbps
Max burst size	16 bytes	16 bytes	16 bytes	16 bytes	16 bytes
Min burst spacing (from start of one burst to next)	100 µs	200 µs	400 µs	800 µs	1600 µs
Max sustained data rate	430 kbps	210 kbps	100 kbps	40 kbps	25 kbps

Sustained reception in slave mode at high bit rates can starve the application processor and cause overruns, and presents a possible risk of watchdog timeout. Care must be given to allow the Smart Transceiver to process received bytes in a timely manner. Master mode has no such restriction because the Smart Transceiver regulates the data transfer.

The **clockedge** and **invert** keywords are used to determine the point at which data is sampled and the idle level of the clock signal. By default, the clock signal is idle at the logic 1 level. Use the **invert** keyword to change the idle state to correspond to a logic 0 level. Common SPI implementations use the terms clock phase (CPHA) and clock polarity (CPOL) to determine the behavior of the clock signal during SPI transmissions. These terms relate directly to the clockedge and invert keywords used in the I/O object declaration, as described in **Table 43**.

Table 43. Relating CPHA and CPOL to Neuron C Declarations

SPI Clock Signal State	Neuron C Declaration
CPHA 0 1	 clockedge(-) clockedge(+)
CPOL 0 1	 invert [default]

The active edge of the clock is determined by the **clockedge** and **invert** keywords. If the clock signal is idle at logic 1 (default), then **clockedge(-)** indicates that the falling edge of the clock signal is active. If the **invert** keyword is used, the rising edge of the clock signal would be active (see **Figure 39** and **Figure 40**). In-phase interfaces (CPHA=1) present the data bit on the first transition of the clock signal, and latch it on the second transition. Out-of-phase interfaces (CPHA=0) present the data bit before the first transition of the clock signal, and latch it on the first transition.

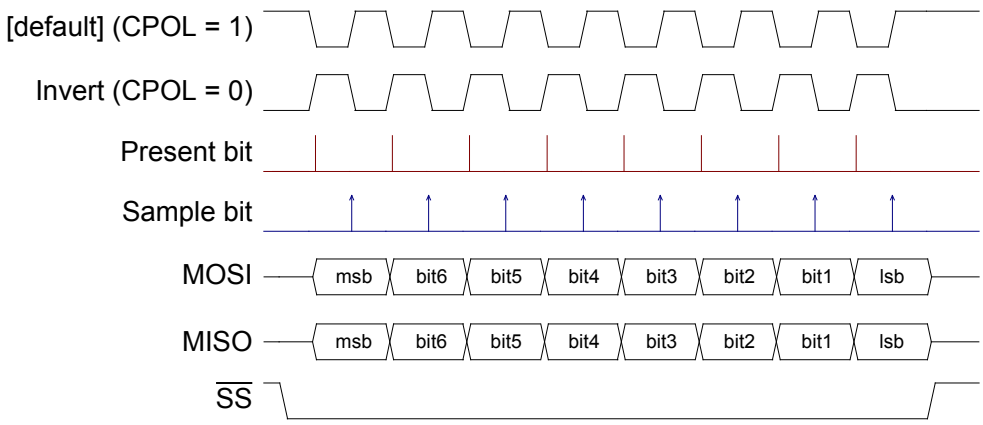


Figure 39. Transmission Timing for Clockedge(-) (CPHA : 0)

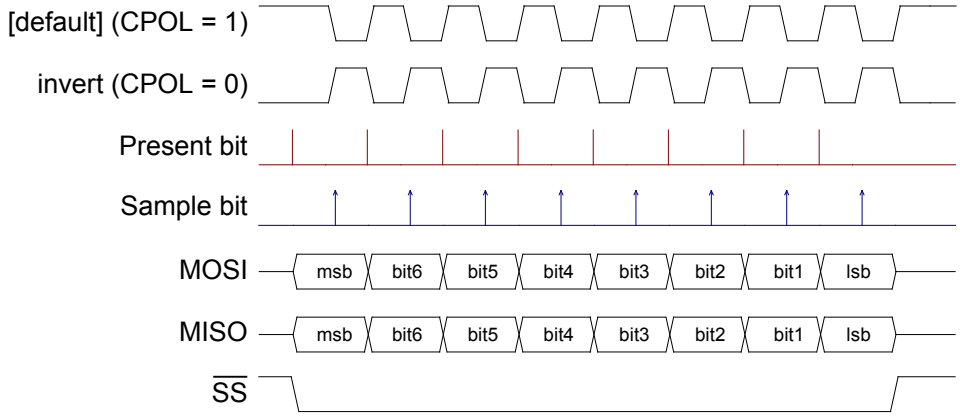


Figure 40. Transmission Timing for Clockedge(+) (CPHA : 1)

Up to 255 bytes can be bi-directionally transferred at a time. This I/O model depends on interrupts to process data at high speed and does not use the **io_in()** and **io_out()** function calls. After transfer is initiated, control is returned to the application immediately, and the application needs to poll the I/O model for completion. Transfers can be suspended and resumed by disabling and enabling interrupts. Turning off interrupts might be required when going off-line, or for assuring that other time-critical application execution is not disturbed by background interrupts. Additionally, transfers can be aborted.

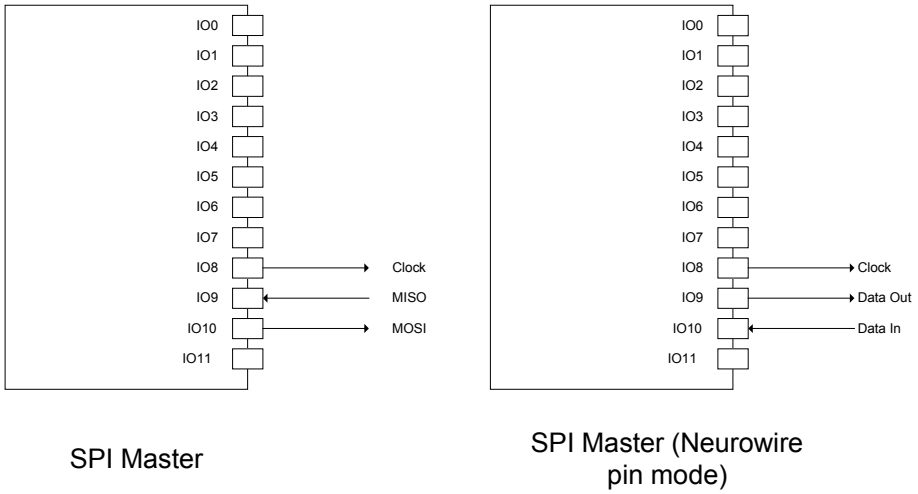


Figure 41. SPI Master Mode I/O

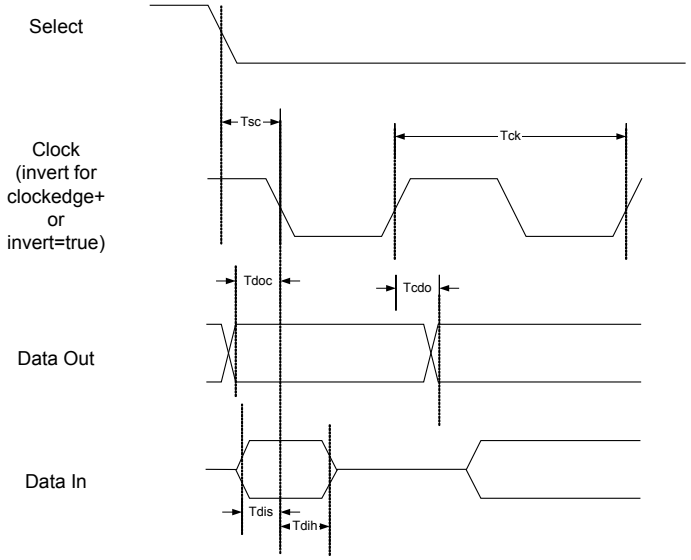


Figure 42. SPI Master Mode Timing

Table 44. SPI Master Mode I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
T_{ck}	Clock cycle (user specified)	—	—	—
T_{sc}	Select low to Clock transition	4.8 µs	—	—
T_{doc}	Data out to Clock (1st bit of invert mode)	0.5 * T_{ck}	—	—
T_{cdo}	Clock to data out	—	—	5 ns
T_{dis}	Data in setup	10 ns	—	—

Symbol	Description	Minimum	Typical	Maximum
T_{dih}	Data in hold	10 ns	—	—

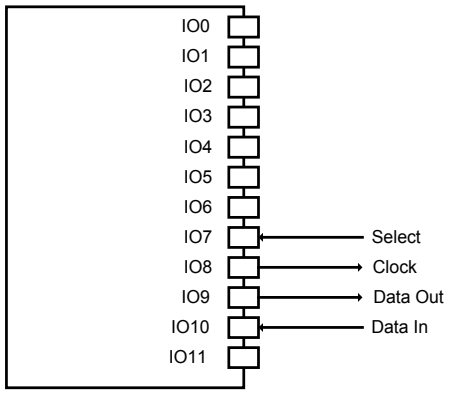


Figure 43. SPI Slave Mode I/O

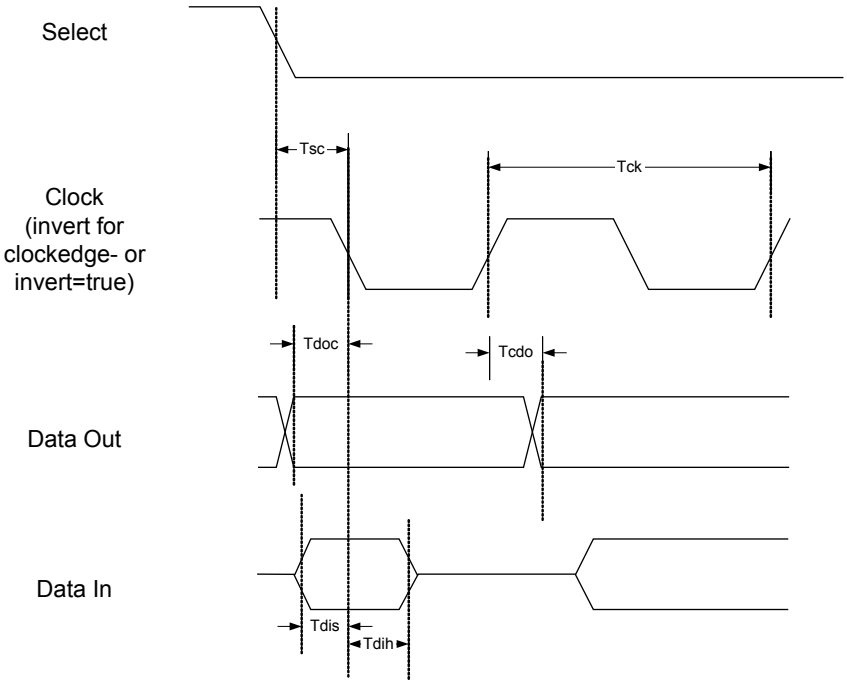


Figure 44. SPI Slave Mode Timing

Table 45. SPI Slave Mode I/O Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
T_{ck}	Clock cycle (user specified)	—	—	1.25
T_{sc}	Select low to Clock transition	220 µs	—	—
T_{doc}	Data out to Clock (1st bit of invert mode)	440 ns	—	—
T_{cdo}	Clock to data out	—	—	45 ns
T_{dis}	Data in setup	10 ns	—	—
T_{dih}	Data in hold	10 ns	—	—
T_{sdz}	Select high to data in high impedance	—	—	220 ns

Programming Considerations

You can enable and disable SPI interrupts. For example, you can turn off interrupts when going offline, or to assure that other time-critical application functions are not disturbed by background interrupts. The SCI interrupt signal is used by the firmware driver for the SCI I/O model. It is not directly accessible by the application program.

The SPI interrupt is enabled by default. For Series 3100 devices, the **io_idis()** function disables I/O interrupts. The function has the following signature:

void io_idis(void);

For Series 3100 devices, the **io_iena()** function enables I/O interrupts. The function has the following signature:

void io_iena(void);

For Series 5000 and Series 6000 devices, you cannot disable the SPI interrupt.

To cancel an SPI operation currently in progress, use the **spi_abort()** function rather than disabling interrupts.

Syntax

IO_8 spi master|slave [**select(IO_7)**] [**clock(***const-expr***)**] [**invert**] [**clockedge(+|-)**] [**neurowire**] *io-object-name***;**

master|slave

Determines whether the hardware is in master or slave mode. When using a Neuron C device in SPI master mode, no other masters can be used in the same bus.

select(IO_7)

Set this option to have pin **IO_7** used as a slave select (SS) signal in slave mode. In slave mode, this option is used when there are multiple slaves connected to a master. However, when the device is the only slave (and thus there is no need for the master to use a dedicated slave select signal), then pin **IO_7** should be separately declared as an input pin and externally grounded.

In master mode, the **select** keyword is not used; thus, **IO_7** can be used for other purposes.

clock(*const-expr***)**

The clock selection can be an integer from 0 to 7, and selects a clock divisor for the SPI interface. This clock divisor and the input clock control the serial bit rate of the SPI interface. Clock selection applies only to master mode. For a Series 3100, Series 5000, or Series 6000 device at 10 MHz, the minimum serial bit rate is 19531 bps and the maximum rate is 156250 bps.

If you omit this keyword, the default used is **clock(0)**.

invert

By default, the clock is idle at 1. Set this option to specify that the clock is idle at 0.

This definition relates directly to the clock polarity (CPOL) parameter defined for other SPI implementations. Using the **invert** keyword is equivalent to defining CPOL = 0; the default declaration is equivalent to defining CPOL =1. Both the SPI master and the SPI slave are required to use the same clock polarity.

See *Hardware Considerations* for more information.

clockedge(+|-)

Set this option to **+** for in-phase interfaces (CPHA=1) to specify that data is valid on the rising edge of the clock. Set this option to – for out-of-phase interfaces (CPHA=0) to specify that data is valid on the falling edge of the clock. By default, if you omit this parameter, data is valid on the rising edge of the clock (**clockedge(+)**). The clock phase (CPHA) must be identically specified for both the SPI master and SPI slave devices.

In-phase interfaces present the data bit on the first transition of the clock signal, and latch it on the second transition. Out-of-phase interfaces present the data bit before the first transition of the clock signal, and latch it on the first transition.

See *Hardware Considerations* for more information.

neurowire

Set this option to select Neurowire compatible mode, where the MOSI and MISO pins do not change direction based on any slave select. The default is SPI mode.

io-object-name

Specifies a name for the I/O object, in the ANSI C format for variable identifiers.

You can call the **io_set_clock( )** function to change the clock divisor and clock edge at run-time. You cannot change the master/slave or Neurowire/SPI modes at run-time.

io_set_clock(*io-object-name***,** *clock-value*, **clockedge(***clock-code***));**

io_set_clock(*io-object-name***,** *clock-value*, **clockedge(***clock-code***), invert);**

The *clock-value* value corresponds to the specification for the **clock()** keyword. For SPI slave mode devices, the *clock-value* should be 0. The *clock-code* value can either be a single plus character ("**+**") or a single minus character ("–"), as described under the **clockedge** parameter above.

Usage

unsigned short *buffer-size***;**
unsigned short *buffer***[***buffer-size***];**

unsigned short io_in(*io-object-name, buffer***,** *buffer-size***);**
unsigned short io_out(*io-object-name, buffer***,** *buffer-size***);**

unsigned short io_in_ready(*io-object-name***);**
unsigned short io_out_ready(*io-object-name***);**
unsigned short spi_get_error(*io-object-name***);**
void spi_abort(*io-object-name***);**

The SPI I/O object uses pins **IO_8**, **IO_9**, and **IO_10** depending on the mode, as shown in **Table 46**.

Table 46. Pin Use for SPI I/O Object

Mode	IO_8	IO_9	IO_10
master	Clock output	Data input	Data output
slave	Clock input	Data output	Data input
neurowire	—	Data output	Data input

You can use the **io_in( )** and **io_out( )** functions to read and write a hardware SPI interface. This interface is very similar to the **neurowire** I/O model. The **io_in( )** and **io_out( )** calls are functionally equivalent, because SPI input and output occur simultaneously. Because the SPI interface is full duplex, the same buffer is used for both transmission and reception of data, with data transferred serially out of and into the single data buffer at the same time.

A call to the **io_in( )** or **io_out( )** function causes the firmware to set up the receive/transmit buffer and update the receive and transmit counts, which are initially equal. Because the SPI model is interrupt-driven, when the SPI transmitter is empty, the hardware transfers the first (and second if the count is greater than 1) bytes to the hardware shift register; at this point, the transmit count is greater by 2 than the receive count. Thus, after the call to the **io_out( )**

function, there are two bytes of data that have moved out of the buffer and into the hardware.

A consequence of this transmit behavior is that if a SPI transfer by a slave device is set up to transmit a maximum number of bytes and truncate the transfer based on data within the transfer itself, the truncation will likely include 1 to 2 extra bytes. For example, if a transfer is set up by the slave for 100 bytes, and there is a need to stop the transfer after 50 bytes, the interrupt driven firmware will have most likely placed bytes 51 and 52 into the SPI's transmit hardware.

You can use either the **io_in( )** or **io_out( )** function to initiate an I/O operation:

io_in(*io-object-name***, void *** *buf***, unsigned** *len***)**

io_out(*io-object-name***, void *** *buf***, unsigned** *len***)**

The **io_in( )** and **io_out( )** functions are non-blocking; they just initiate the data transfer. You can use the **io_in_ready(***io-object-name***)** and **io_out_ready(***io-object-name***)** event functions to test the state of the SPI interface. These functions are used to determine when the transmission is complete. The **io_out_ready** event returns **TRUE** when output is complete. The **io_in_ready** event returns the number of bytes read in as an **unsigned short**, so when this value matches the *len* parameter from the call to **io_in_request( )** the input operation is complete.

Although the **io_in_ready( )** and **io_out_ready( )** event functions both test the SPI interface, the **io_out_ready( )** function returns **TRUE** before the **io_in_ready( )** function returns a count for the expected count. This difference is due to the fact that the transmit data register chain holds two bytes of data and completes the transmit process before the receive process completes. Thus, you should use the **io_in_ready( )** function to qualify the completion of a transfer.

You can use the **spi_get_error(***io-object-name***)** function to test for SPI errors. Calling **io_in( )** or **io_out( )** clears any previous SPI error code. The **spi_get_error( )** function also clears any SPI error code after returning it. This function returns a cumulative OR of the following bits that reflect data errors:

0x10 Mode fault occurred

0x20 Receive overrun detected

You can use the **spi_abort(***io-object-name***)** function to terminate any operation in progress. After an abort, the **io_in_ready( )** function returns the number of characters read up to the abort.

Example

```
IO_8 spi master clock(4) ioSpi;

when (...) {
  io_out(ioSpi, “Hello SPI World!\r\n”, 18);
}

when (io_out_ready(ioSpi)) {
  unsigned short spiEerror;
  spiEerror = spi_get_error(ioSpi));
  if (spiEerror) {
    // Process SPI error
```

```
    ...
  }
  else {
    // Process end of SPI transmission
    ...
  }
}
```

Wiegand Input

The **wiegand** input model provides an easy interface to any card reader that supports the Wiegand interface standard.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, as well as Series 5000 and Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

Data from the reader is presented to the Neuron Chip or Smart Transceiver through two of its first eight I/O pins, IO0 – IO7. Up to four Wiegand devices can be connected to the Smart Transceiver. Data is read most-significant bit (MSB) first.

Wiegand data starts as a negative-going pulse on one of the two pins selected. One input represents a logical 0 bit and the other pin a logical 1 bit, as selected through the I/O declaration. The bit data on the two lines are mutually exclusive, and are spaced at least 150 μs apart. **Figure 45** shows the timing relationship of the two data lines with respect to each other and to the Smart Transceiver.

Any unused I/O pin from IO0 to IO7 can be optionally selected as the timeout pin. When the timeout pin goes high, the function aborts and returns. The application processor's watchdog timer is automatically updated during the operation of this input object.

Incoming data on any of the Wiegand input pins is sampled by a Series 3100 device every 200 ns for a 10 MHz input clock and by the Series 5000 and Series 6000 devices every 12.5 ns for an 80 MHz system clock (scales inversely with the clock frequency). Because the Wiegand data is usually asynchronous, care must be taken in the application program to ensure that this function is called in a timely manner in order that no incoming data is lost.

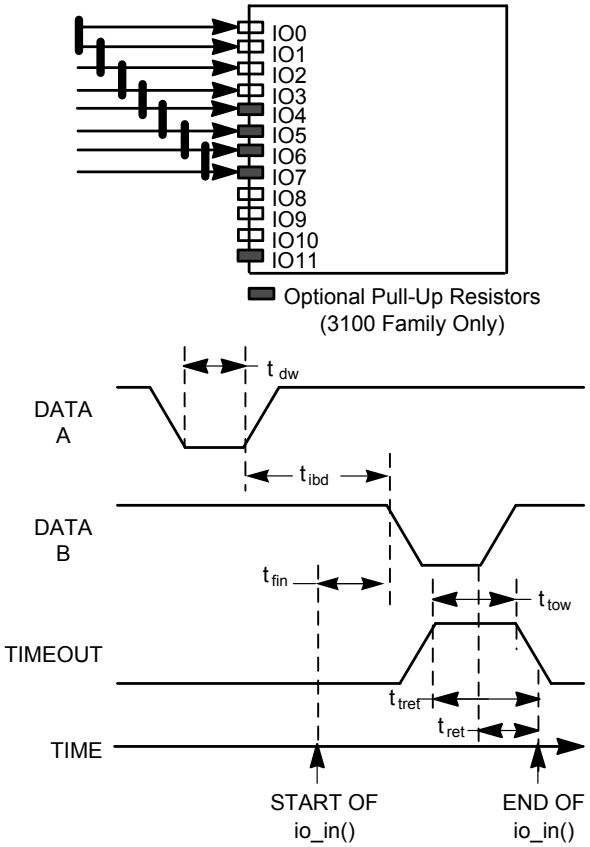


Figure 45. Wiegand Input and Timing

Table 47. Wiegand Input Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{fin}	Function call to start of second data edge	—	75.6 µs	—
t_{dw}	Input data width (at 10 MHz)	200 ns	100 µs	880 ms
t_{ibd}	Inter-bit delay	150 µs	—	900 µs
t_{low}	Timeout pulse width	—	39 µs	—
t_{tret}	Timeout to function return	—	18.0 µs	—
t_{ret}	Last data bit to function return	—	74.4 µs	—

Programming Considerations

The **wiegand** I/O model is used to transfer data from a Wiegand format data stream source. This format encodes data as a series of pulses on two signal lines:

- The zero data bit signal

- A one data bit signal

Data pulses appear exclusively of each other and are typically spaced approximately 1 ms apart. Specifications for the duration of the pulse are typically between 50 to 100 µs, but they can be as short as 50 ns for a Series 3100 device with a 40 MHz input clock, or 12.5 ns for Series 5000 and Series 6000 devices with an 80 MHz system clock. **Table 48** shows the pulse width and inter-bit period (the period between bit pulses).

Table 48. Wiegand Pulse Width and Inter-Bit Period

Parameter	Minimum	Maximum	Typical
Pulse Width	200 ns	880 ms	100 µs
Inter-Bit Time	150 µs	None	900 µs

Wiegand data is asynchronous. The **io_in( )** function must be executing before the second bit arrives, otherwise the first bit data is lost because it then becomes impossible to determine the order of a zero and one event sequence. Data is read most-significant bit (MSB) first, that is, the first data bit read will be stored in the most significant bit location of the first byte of the array when eight bits are read into that byte. If the number of bits transferred is not a multiple of eight, as defined by *count*, the last byte transferred into the array contains the remaining bits right justified within the byte.

For Wiegand input, one of the **IO_0** through **IO_7** pins can optionally be designated as a timeout pin. A logic one level on the timeout pin causes the Wiegand input operation to terminate before the specified number of bits has been transferred. The Neuron Chip or Smart Transceiver updates the watchdog timer while waiting for the next zero or one data bit to arrive. This timeout input can be a one-shot timer counter output, an RC circuit, or a ~Data_valid signal from the reader device.

The **return_value** for the **io_in( )** function for this model is an **unsigned short**, and it indicates the number of bits stored into the array. Whenever the **io_in( )** function for a Wiegand I/O object is called, it immediately returns if there is currently no activity on the indicated I/O pins. Otherwise, the function continues to process input data until either *count* bits are stored, or until the timeout event occurs. When the timeout event occurs, the number of bits read and stored is returned. The **io_in( )** function is blocking, and can take more than one second to process the card information, depending upon the speed at which the card travels through the reader. Because this function ties up the application processor, it handles updates to the watchdog timer.

Syntax

pin [**input**] **wiegand** [**timeout(***pin-nbr***)**] *io-object-name***;**

pin

An I/O pin. Wiegand input requires two adjacent pins. The DATA 0 pin is the pin specified, and the DATA 1 pin is the following pin. The pin specification denotes the lower-numbered pin of the pair and can be **IO_0** through **IO_6**.

timeout (*pin*-nbr**)**

Optionally specifies the timeout signal pin, in the range of **IO_0** to **IO_7**. The Neuron firmware checks the logic level at this pin whenever it is waiting for a pulse at either the DATA 0 or DATA 1 pins. If a logic level 1 is sensed, the transfer is terminated.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *count***,** *input-buffer***[***buffer-size***],** *bit-count***;**

count **= io_in(***object-name***,** *input-buffer***,** *bit-count***);**

Example

```
// This application is written so that the
// Wiegand input is being polled for a majority
// of the time, breaking out and returning to the
// scheduler only periodically. This makes the
// probability of capturing the first bits of
// the input much higher since the bits arrive
// asynchronously. Timeout is from a hardware oneshot.

unsigned int data[4], breaker, nbits;
IO_2 input wiegand timeout (IO_0) ioCardData;
IO_0 output oneshot invert clock (7) ioPinTimer = 1;

when(TRUE) {
  for (breaker=200; breaker; breaker--) {
    io_out(ioPinTimer, 19500UL);
    // Store 26 bits into data
    nbits = io_in(ioCardData, data, 26);
    if (nbits) {
      . . .   // Process data just read
    }
  }
}
```

5

Timer/Counter Input Models

This chapter describes timer/counter input models. Timer/counter I/O models use a timer/counter circuit in the Neuron Chip or Smart Transceiver. Each Neuron Chip and each Smart Transceiver has two timer/counter circuits: One whose input can be multiplexed, and one with a dedicated input.

Introduction

A Neuron Chip or Smart Transceiver has two 16-bit timer/counters:

- For the first timer/counter, IO0 is used as the output, and a multiplexer selects one of pins IO4 – IO7 as the input.
- The second timer/counter uses IO1 as the output and IO4 as the input.

Figure 46 shows the basic timer/counter circuits for the Neuron Chip and Smart Transceiver.

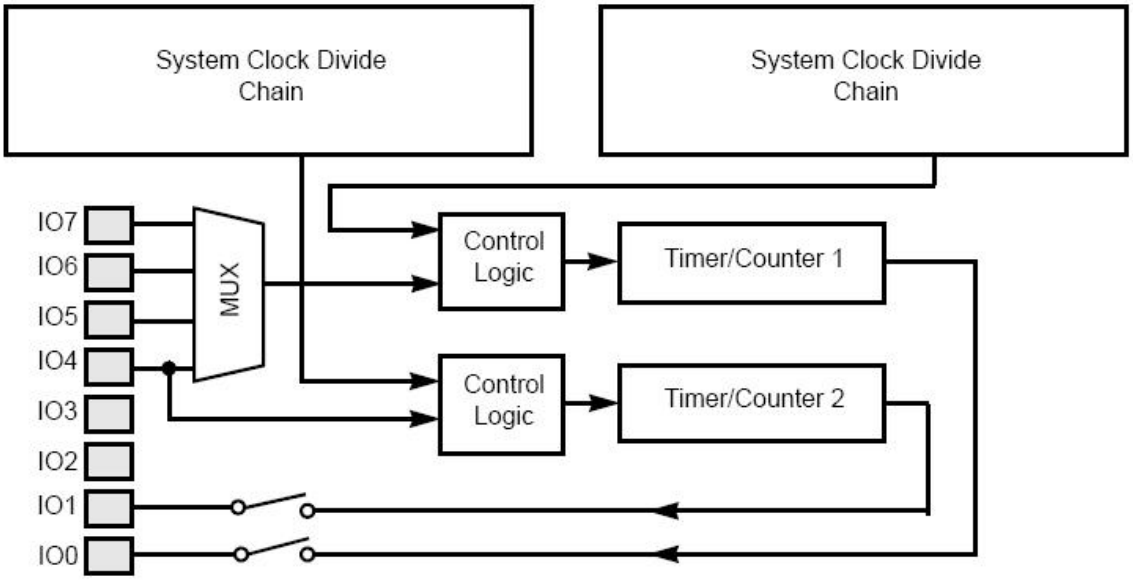


Figure 46. Timer/Counter Circuits

A single application can declare multiple input devices that use timer/counter I/O models. By calling the **io_select()** function, the application can use the first timer/counter in up to four different input functions. If a timer/counter is configured in one of the output functions, or as a quadrature input, then it cannot be reassigned to another timer/counter object in the same application program.

The timing numbers shown in this chapter are valid for either an explicit I/O call or an implicit I/O call through a **when** clause, and are assumed to be for a Series 3100 Smart Transceiver running at 10 MHz.

Input timer/counter models have the advantage (over non-timer/counter objects) in that input events are captured even if the application processor is occupied doing something else when the event occurs. A **when** statement condition for an event being measured by a timer/counter is **TRUE** when the measurement is complete and a value is returned to an event register. If the processor is delayed due to software processing and cannot read the register before another event occurs, then the value in the register reflects the status of the last event. The timer/counters are automatically reset upon completion of a measurement.

Timer/counter I/O models can also be used to trigger application-specific interrupts. See the *Neuron C Programmer's Guide* for more information about application interrupts.

Important: The *first* measured value of a timer/counter is always discarded to eliminate the possibility of a bad measurement after the chip comes out of a reset condition.

Single events cannot be measured with the timer/counters. **Figure 47** shows an example of how the timer/counter objects are processed with a Neuron C when statement.

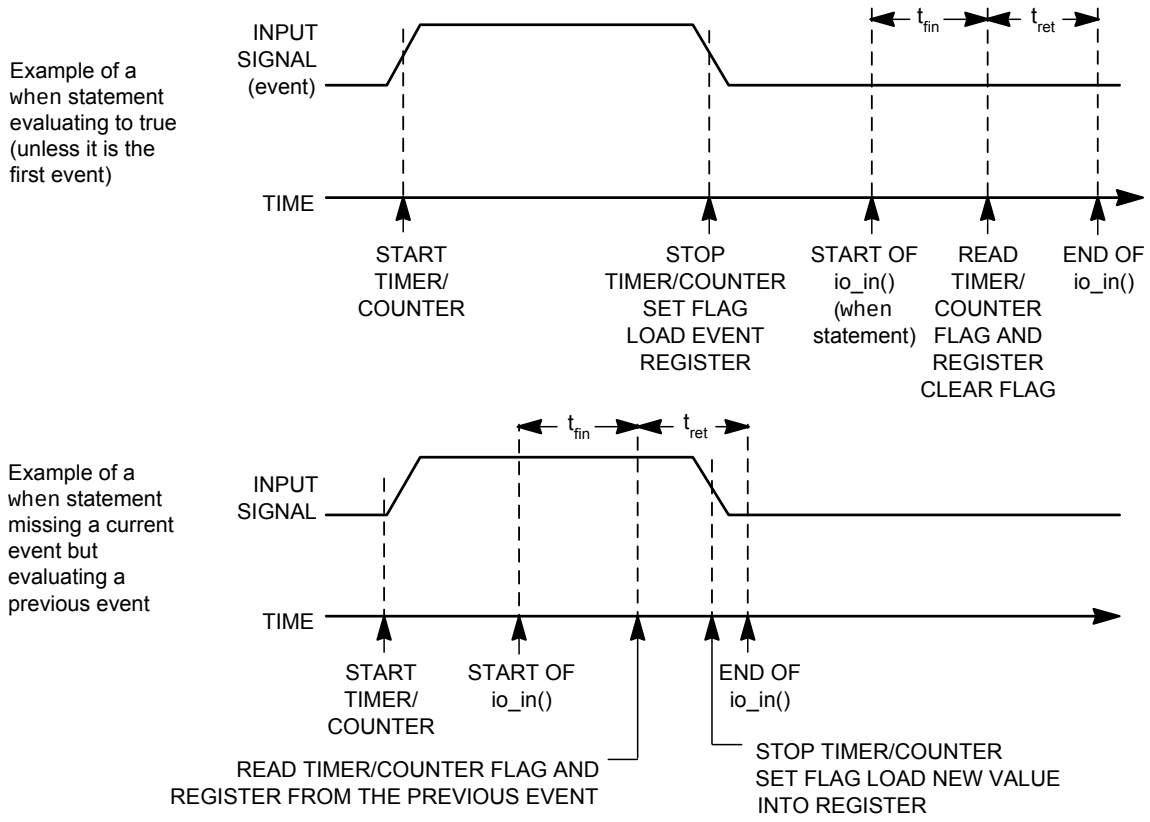


Figure 47. Example of when Statement Processing for the Ontime Input Object

As with all CMOS devices, floating I/O pins can cause excessive current consumption. To avoid this excess current consumption, declare all unused I/O pins as **bit** output. Alternatively, unused I/O pins can be connected to + VDD5 (for Series 3100 devices), +VDD33 (for Series 5000 devices), or GND.

Dualslope Input

The **dualslope** I/O model is used to control and measure the integration periods of a dualslope integrating analog-to-digital (A/D) converter. You can use this I/O model to implement low-cost A/D converters for analog input.

The I/O model controls a timer/counter output pin based on a **control_value** argument and the state of a timer/counter input pin. When combined with external analog circuitry, the Neuron Chip or Smart Transceiver performs A/D measurements with 16 bits of resolution for as little as a 3.278 ms integration period for a Series 3100 device with a 40 MHz input clock (the period scales with

the input clock). Faster conversion rates are attainable at the expense of bit resolution.

For a Series 3100 device, the duration of the first integration period is a function of **control_value** and the selected clock value:

duration (ns) = control_value * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, the duration of the first integration period is a function of **control_value** and the selected clock value:

duration (ns) = control_value * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

For a Series 3100 device, the value read back by this device reflects the length of the second integration period, and is also in units of the selected clock value:

2nd_integration (ns) = input_value * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, the value read back by this device reflects the length of the second integration period, and is also in units of the selected clock value:

2nd_integration (ns) = input_value * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

A single timer/counter provides the control out signal and senses a comparator output signal. The control output signal controls an external analog multiplexer that switches between the unknown input voltage and a voltage reference. The timer/counter's input pin is driven by an external comparator that compares an integrator output with a voltage reference.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

The timer/counter provides the control output signal, and senses a comparator output signal. The control output signal controls an external analog multiplexer that switches between the unknown input voltage and a voltage reference. The timer/counter's input pin is driven by an external comparator that compares the integrator's output with a voltage reference. At the end of conversion, the external comparator drives a low level to one of pins IO4 – IO7. If external circuitry indicates "end of conversion" with a high level, use the **invert** keyword in the I/O object's declaration.

The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*.

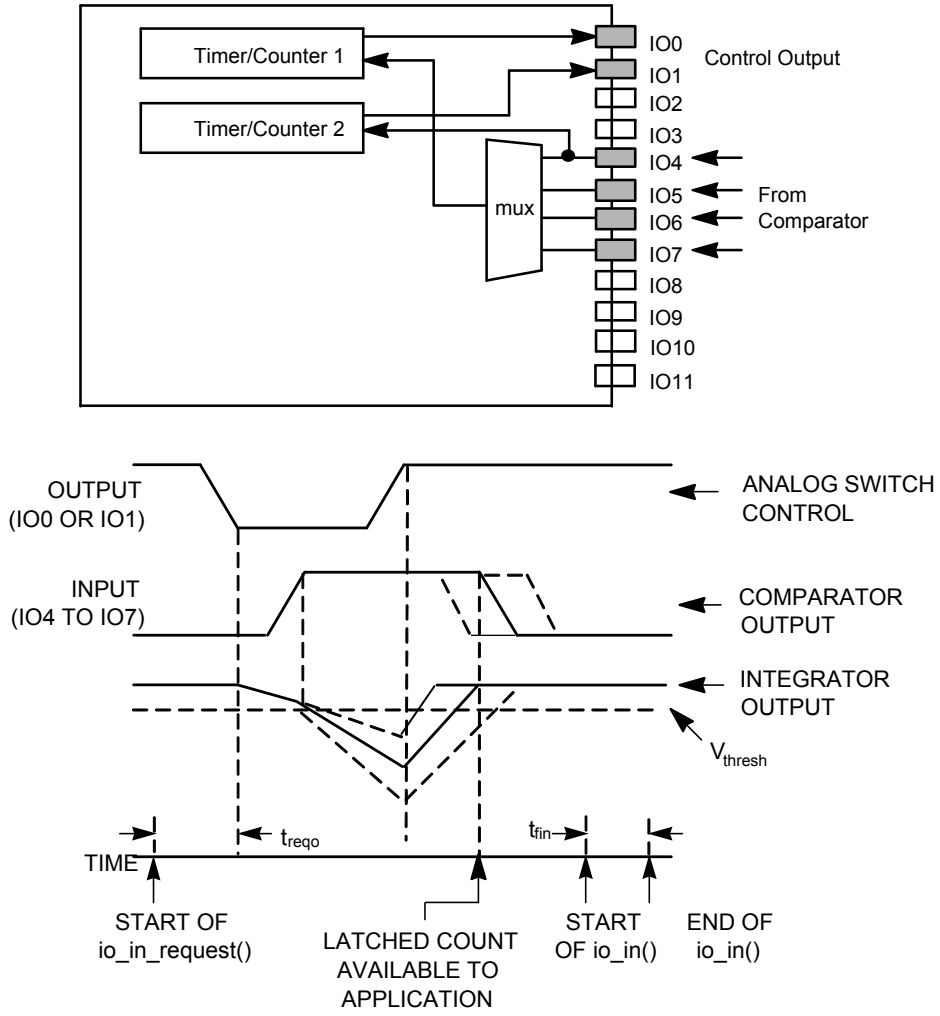


Figure 48. Dualslope Input and Timing

Table 49. Dualslope Input Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{reqo}	io_in_request() to output toggle	—	75.6 µs	—
t_{fin}	Input function call and return	—	82.8 µs	—

Programming Considerations

For dualslope input, the data type of the **control_value** for the **io_in_request( )** function is an **unsigned long**. The return value of the **io_in( )** function is an **unsigned long**. Both the return value for **io_in( )** and the value stored at **input_value** is a number biased negatively by the **control_value** used for the **io_in_request( )** function, and can be corrected by adding the **control_value** value into it.

For additional information regarding dualslope A/D conversion, see the *Analog to Digital Conversion with the Neuron Chip* engineering bulletin (part no. 005-0019-02).

Neuron C Resources

The following functions and events are provided for use with the dualslope input model:

io_in_request( )

Starts the first step of the integration process. The *control_value* argument controls the length of the first integration period.

io_update_occurs

Signals the end of the entire conversion process. The value at *input_value* now contains the new measurement data.

Syntax

pin [**input**] **dualslope** [**mux** | **ded**] [**invert**] [**clock (***const-expr***)**] *io-object-name***;**

pin

An I/O pin. Dualslope input can specify pins **IO_4** through **IO_7**.

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin **IO_4** is the input pin. The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter.

When the dedicated timer/counter is used, the control output pin will be **IO_1**. When the multiplexed timer/counter is used, the control output pin will be **IO_0**. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert

Reverses the logical value of the input pin. Use this keyword if the comparator output is high when the converter is in the idle state.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on a Series 5000 device, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of

the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()** value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value*, *control-value*;

io_in_request(*io-object-name*, *control-value***);**
input-value **= io_in(***io-object-name***);**

Example

```
IO_4 input dualslope ded clock(0) ioDualSlope;
mtimer repeating goTime;
unsigned long data;
...

when (reset) {
  goTime = 500; // Perform a measurement every 500ms
}

when (timer_expires(goTime)) {
  // Start the first integration period (9ms at 10MHz).
  io_in_request(ioDualSlope, 45000UL);
}

when (io_update_occurs(ioDualSlope)) {
  // The value at input_value is biased by the
  // negative value of the control value used.
  // Correct this by adding it back now.
  data = input_value + 45000UL;
}
```

Edgelog Input

The **edgelog** I/O model can record a stream of input pulses that measure the consecutive low and high periods at the input and store them in user-defined storage (see **Figure 49**). The values stored represent the units of clock period between rising and falling input signal edges.

For a Series 3100 device, this I/O model measures a series of both high and low input signal periods on a single input pin, **IO_4**, in units of the clock period:

time_on/time_off (ns) = value_stored * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, this I/O model measures a series of both high and low input signal periods on a single input pin, **IO_4**, in units of the clock period:

time_on/time_off (ns) = value_stored * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

Edgelog input can be used to capture complex waveforms such as infrared command input (see also *Infrared Input*), or to decode any type of bitstream that contains data in the time domain (an arbitrarily-spaced stream of input edges or pulses), such as bar code input.

This model applies to Series 3100 Neuron Chips and Smart Transceivers,to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

The measurement series starts on the first rising (positive) edge, unless the **invert** keyword is used in the I/O object declaration. The measurement process stops whenever an overflow condition is sensed on either timer/counter.

The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*.

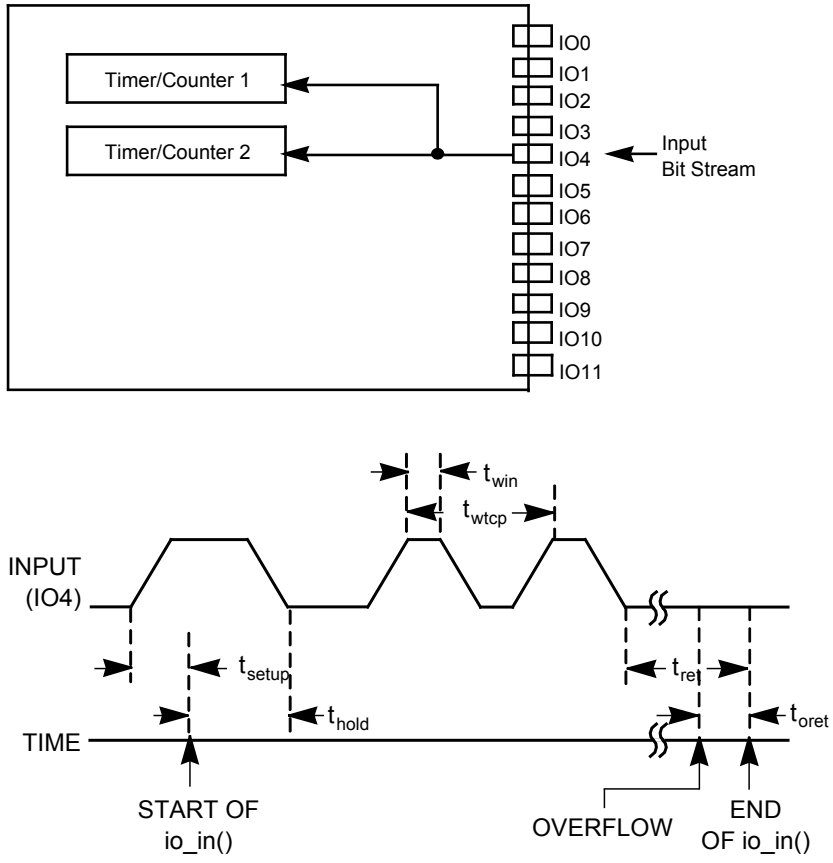


Figure 49. Edgelog Input and Timing

Table 50. Edgelog Input Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{setup}	Input data setup	0	—	—
t_{win}	Input pulse width	1 *T/C clk*	—	65534 *T/C clks*
t_{hold}	io_in() call to data input edge for inclusion of that pulse	26.4 µs	—	—
t_{wtcp}	Two consecutive pulse widths	104 µs	—	—
t_{oret}	Return on overflow	—	42.6 µs	—
t_{ret}	Return on count termination	—	49.6 µs	—
Note: *T/C clk* represents the period of the clock used during the declaration of the I/O object.				

Programming Considerations

For edgelog input, the **io_in( )** function requires a pointer to a data buffer, into which the series of **unsigned long** values are stored, and a count argument, which controls the number of values to be stored. The values stored represent the units of clock period between input signal edges, rising or falling. The **io_in( )** function returns an **unsigned short int** that contains the actual number of edge-to-edge periods stored. No input events are associated with an edgelog input object.

During the **io_in( )** function call, the measurement process stops whenever the maximum period is exceeded. In this case, the value returned will not be equal to the count argument passed.

If a preload value is specified, it must be added to the value returned by **io_in( )**. The resulting addition may cause an overflow, but this is normal.

This I/O model uses both of the Neuron timer/counters.

Neuron C Resources

The following functions are provided specifically for use with the edgelog I/O object:

- **io_edgelog_preload( )**

  Changes the maximum value for each period measurement. The maximum value may range from 1 to 65535; the default value is 65535. This function is only used for an edgelog device that is *not* declared with the **single_tc** option keyword.

- **io_edgelog_single_preload( )**

Changes the maximum value for each period measurement for an edgelog device declared with the **single_tc** option keyword. The maximum value may range from 1 to 65535; the default value is 65535.

Example for a Series 3100 device with a 10 MHz input clock: An **edgelog** input object using **clock(3)** and the default maximum period yields a 1.6 µs resolution and does not overflow until 104.86 ms elapse. Using a value of 7500 for **io_edgelog_preload( )** results in the **io_in( )** function call terminating if 12 ms elapse with no input edges.

Syntax

pin [**input**] **edgelog** [**single_tc**] [**mux | ded**] [**clock (***const-expr***)**] *io-object-name***;**

pin

Specifies a Neuron input pin for the edgelog input object. The input pin can be **IO_4** through **IO_7** if the **single_tc** option is specified, otherwise the input pin must be **IO_4**.

single_tc

Optionally specifies that a single timer/counter should be used. If this keyword is not specified, two timer/counters are used. If a single timer/counter is specified, the application can only be loaded on a device based on a Series 3100, Series 5000, or Series 6000 Smart Transceiver, a Neuron 3120 Chip, or a Neuron 3150B1 Chip (or newer).

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This option is only necessary with the **single_tc** option when the edgelog device is declared on pin **IO_4**. The multiplexed timer/counter is always used on pins **IO_5** through **IO_7**.

clock(*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on a Series 5000 device, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()** value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

In **Figure 50**, an **io_in( )** function call is executed sometime after the **IO_4** input signal is sensed as changing to high, but before it has changed back to low. The first period, Period [1], is stored as a value in the array pointed to by the buffer argument. If the **io_in( )** function call occurs within the Period [2] time frame, the data for Period [1] is lost.

Individual period measurements can be skipped if the sum of two consecutive periods is less than 104 µs (for a Series 3100 device with a 10 MHz input clock), regardless of the timer/counter clock setting. The minimum value scales with the input clock.

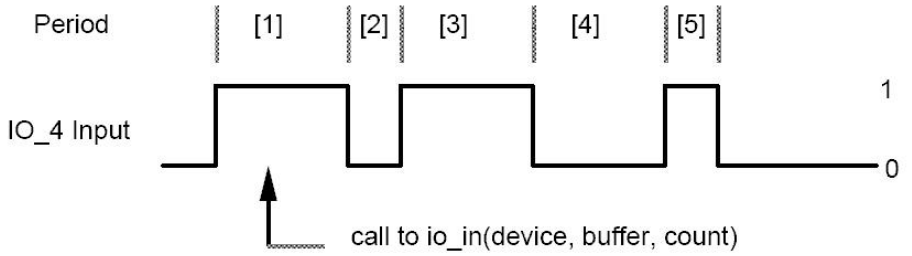


Figure 50. Call to io_in(device, buffer, count)

If the **IO_4** input pin has been at a constant level for longer than the overflow period before the call to **io_in( )** is made, the first value stored in the buffer is not the maximum value, but rather the value for the next period.

Usage

unsigned int *count***;**
unsigned long *input-buffer***[***buffer-size***];**

count = **io_in(***io-object-name***,** *input-buffer***,** *count***);**

Example

```
IO_4 input edgelog clock(7) ioTimeStream;

// The next object allows direct reading
// of time_stream level.
IO_4 input bit ioTimeStreamLevel;

unsigned int edges;
unsigned long buffer[20];
unsigned long preLoad = 0x4000;

when (reset) {
  io_edgelog_preload(preLoad);
}

when (io_changes(ioTimeStreamLevel) to 1) {
  int i;
```

```
  // Retrieve edge log
  edges = io_in(ioTimeStream, buffer, 20);
  // Correct for preload offset
  for (i = 0; i < edges; i++) {
    buffer[i] += preLoad;
  }
  // Process data
  ...
}
```

Infrared Input

The **infrared** I/O model is used to capture a data stream generated by a class of infrared remote control devices (see **Figure 51**). This class of devices generates a stream of ones and zeros by modulating an infrared emitter for an on and off cycle, each cycle representing either a one or a zero. The period of this on/off cycle determines the data bit value, a longer cycle implies a one, a shorter cycle implies a zero. The actual threshold for the on/off determination is set at the time of the call of the function. The measurements are made between the negative edges of the input bits unless the **invert** keyword is used in the I/O object declaration.

Typically, an infrared signal consists of an infrared source modulated at a carrier frequency between 38 kHz and 42 kHz. An infrared receiver/demodulator is used external to the Neuron Chip or Smart Transceiver to produce a digital sequence with the carrier removed. Upon execution of the **io_in( )** function for the infrared I/O object, the Neuron Chip or Smart Transceiver measures the cycle times and stores the data bits into a buffer passed to the **io_in( )** function.

For a Series 3100 device, a timer/counter is used to make the series of cycle time measurements. The resolution of these measurements is in units of the clock period:

period (ns) = measured_value * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, a timer/counter is used to make the series of cycle time measurements. The resolution of these measurements is in units of the clock period:

period (ns) = measured_value * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

See the **edgelog** input model for an alternate method to decode infrared inputs.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceviers.

Hardware Considerations

The input to this I/O object is the demodulated series of bits from infrared receiver circuitry. The infrared input model, based on the input data stream, generates a buffer containing the values of the bits received. The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*.

This I/O model can be used with an off-the-shelf infrared encoder/decoder chip that uses the NEC IR protocol to quickly develop an infrared interface to a Neuron Chip or Smart Transceiver. You can also use the edgelog input model for this purpose, but your application program will likely require more code.

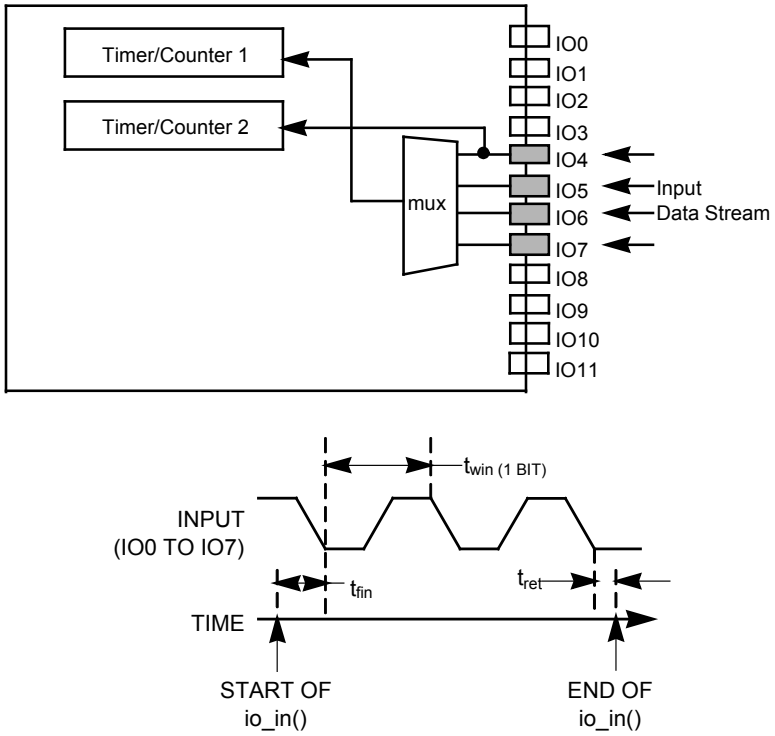


Figure 51. Infrared Input and Timing

Table 51. Infrared Input Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{fin}	Function call to start of input sampling	—	82.2 µs	—
t_{ret}	End of last valid bit to function return	*max-period*	*max-period*	—
t_{win}	Minimum input period width	—	93 µs	—
Note: *max-period* is the timeout period passed to the function at the time of the call.				

Programming Considerations

For infrared input, the **io_in( )** function requires, in addition to the *io-object-name*, four arguments:

- A pointer to a data buffer in which the series of data bits are stored

- A *bit_count* argument, which is the expected number of data bits to be received and stored
- A *max_period* argument limiting the range of the timer/counter measurement process
- A *threshold* argument, representing the half way point, in timer/counter count clocks, between a zero data period and a one data period

The value returned by the **io_in( )** function is the actual number of bits read. If less than the expected number of bits (controlled by *bit_count***)** appear at the input pin, the **io_in( )** function waits for the *max_period* period before returning. If the expected number of bits, or more, appear at the input pin, the **io_in( )** function waits for silence at the input pin before returning. Silence is defined as a lack of input cycles for the *max_period* period. If input cycles persist, the function returns after 256 input cycles occur. This data may be retrieved using the **tst_bit( )** function.

The *max_period* argument is an **unsigned long**, and is passed as the negative (two's complement) of the required value. The threshold argument is passed as the *max_period* value plus the required threshold value. The **edgelog** input object type can be used to read inputs from infrared devices that do not conform to the assumptions of the infrared input model.

Syntax

pin [**input**] **infrared** [**mux | ded**] [**invert**] [**clock (***const-expr***)**] *io-object-name***;**

pin

An I/O pin. Infrared input can specify pins **IO_4** through **IO_7**.

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin **IO_4** is the input pin.

The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used on pins **IO_5** through **IO_7**.

invert

Causes the measurement of the cycle period to be between positive input edges rather than the default, which is between negative input edges.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for infrared input is clock 6. The **io_set_clock( )** function can be used to change the clock. **Table 52** shows the clock values for a Series 3100 device with an input clock of 10 MHz, and a Series 5000 or Series 6000 device with a system clock of 80 MHz. The values in the table can be adjusted for different input clocks by scaling them inversely proportional to the change in input clock (for example, for a 20 MHz clock, divide all values in the table by 2, and for a 5 MHz clock, multiply all values in the table by 2).

Table 52. Clock Values

Clock	Range and Resolution Period	
	Series 3100 (10 MHz Clock)	**Series 5000 and Series 6000 (80 MHz Clock)**
0	0 to 13.11 ms in steps of 200 ns (0-65535)	0 to 1.639 ms in steps of 12.5 ns (0-65535)
1	0 to 26.21 ms in steps of 400 ns	0 to 3.278 ms in steps of 25 ns
2	0 to 52.42 ms in steps of 800 ns	0 to 6.555 ms in steps of 50 ns
3	0 to 104.86 ms in steps of 1.6 µs	0 to 13.11 ms in steps of 100 ns
4	0 to 209.71 ms in steps of 3.2 µs	0 to 26.21 ms in steps of 200 ns
5	0 to 419.42 ms in steps of 6.4 µs	0 to 52.42 ms in steps of 400 ns
6 (default)	0 to 838.85 ms in steps of 12.8 µs	0 to 104.86 ms in steps of 800 ns
7	0 to 1.677 s in steps of 25.6 µs	0 to 209.71 ms in steps of 1.6 µs

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int *bit-count***;**
unsigned int *input-buffer***[***buffer-size***];**
unsigned long *max-period***,** *threshold***;**

count = **io_in**(*io-object-name*, *input-buffer*, *bit-count*, *max-period*, *threshold*);

Example

This example works with a Series 3100 device and an infrared encoder/decoder chip that uses the NEC IR protocol. This encoder produces a 9 ms start bit cycle before the actual data stream. During the start bit cycle, the input signal is driven low. This start condition is typical of infrared encoders because it allows a receiver's or demodulator's automatic gain control (AGC) circuit time to adjust. It also gives the Neuron Chip or Smart Transceiver some time to catch this condition from the scheduler, and enter the **io_in( )** function. After the AGC burst, the protocol includes a 4.5 ms space, which is then followed by 32 bits for the device address and command.

The NEC protocol uses pulse distance encoding of the bits. Each pulse is a 560 µs long 38 kHz carrier burst (about 21 cycles). A logical one requires 2.250 ms to transmit, and a logical zero requires 1.125 ms to transmit. The input clock is 10

MHz, and the timer/counter clock is clock (7). This yields a 25.6 µs timer/counter clock resolution.

The *max-period* parameter is set to cause an overflow at 110% of the start cycle (the timer/counter will count *up* from this value):

$$65536 - \left(\frac{1.10 * (9 * 10^{-3})}{25.6 * 10^{-6}} \right)$$

$$= 65149$$

Given the one and zero data periods, the threshold value is:

$$65149 + \left(\frac{\left\lfloor \frac{(1.125 * 10^{-3}) + (2.25 * 10^{-3})}{2} \right\rfloor}{25.6 * 10^{-6}} \right)$$

$$= 65149 + 66$$

$$= 65215$$

This encoder always sends 32 bits, so the count will be 32, and the returned *input-buffer* will be an array of 4 bytes.

```
// This is the demodulated IR input.
// Use the non-inverted mode to read falling to falling
// input periods.
IO_4 input infrared ded clock (7) ioIr;

// This object allows the application to monitor the input
// signal before entering the io_in() function.
IO_4 input bit ioIrLevel;

unsigned int bits;
unsigned int irb[4];
. . .

when (io_changes(ioIrLevel) to 0) {
  bits = io_in(ioIr, irb, 32, 65149UL, 65149UL + 66UL);
  if (bits == 32) {
    // So far, a valid data message.
    . . .
  }
}
```

Ontime Input

For a Series 3100 device, the **ontime** I/O model measures pulsewidth or period of an input signal (the high or low period) in units of the clock period:

time_on (ns) = return_value * 2000 * 2^(clock) / input clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, the **ontime** I/O model measures pulsewidth or period of an input signal (the high or low period) in units of the clock period:

time_on (ns) = return_value * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

You can use this model to implement digital-to-analog (D/A) converters, frequency counters, or tachometers.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to measure the time for which its input is asserted. The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*. Assertion can be defined as either logic high or logic low. This model can be used as a simple analog-to-digital converter with a voltage-to-time circuit, or for measuring velocity by timing motion past a position sensor (see **Figure 47** and **Figure 52**).

The **ontime** I/O model is level sensitive. The active level of the input signal gates the clock driving the internal counter in the Neuron Chip or Smart Transceiver.

The actual active level of the input depends on whether the **invert** option is used in the declaration of the I/O object. The default is the high level.

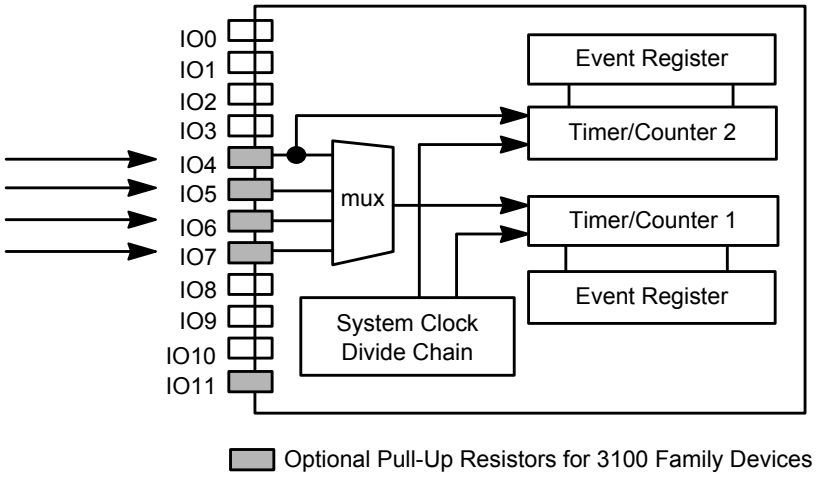


Figure 52. Ontime Input

Table 53. Ontime Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to input sample	86 µs

Symbol	Description	Typical at 10 MHz
t_{ret}	Return from function	52 µs or 22 µs
Note: If the measurement is new, t_{ret} = 52 µs. If a new time is not being returned, t_{ret} = 22 µs.		

Programming Considerations

For **ontime** input, the data type of the return value for the **io_in( )** function is an **unsigned long**.

The state of the input pin is latched in hardware every 50 ns for a Series 3100 device with a 40 MHz input clock, or every 12.5 ns for the Series 5000 and Series 6000 devices with an 80 MHz system clock (the value scales inversely with clock speed). If no edges occur during the measuring period, an overflow condition occurs. The next call to the **io_in( )** function after the overflow occurs returns the out-of-range value (0xFFFF). The **io_update_occurs** event is not asserted as **TRUE** unless the program uses the **io_preserve_input( )** function after the **io_select( )** when using the multiplexed timer/counter, or in the reset task when using the dedicated timer/counter.

Syntax

pin [**input**] **ontime** [**mux** | **ded**] [**invert**] [**clock (***const-expr***)**] *io-object-name***;**

pin

An I/O pin. Ontime input can specify one of pins **IO_4** through **IO_7** as the input pin.

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This keyword is used only when pin **IO_4** is used as the input pin.

The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert

Causes the measurement of the low period of the input signal. By default, measurement occurs on the high period of the input signal.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the

TCCLK_* macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on a Series 5000 or Series 6000 device, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()** value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value*;

input-value = **io_in(***io-object-name***);**

Example

```
IO_4 input ontime ded clock(7) ioGateTime;
unsigned long pulseDuration;

when (io_update_occurs(ioGateTime)) {
  pulseDuration = input_value;
  // measures up to 1.677 seconds
}
```

Period Input

For Series 3100 devices, the **period** I/O model measures the total period, from edge to edge, of an input signal in units of the clock period, calculated as follows:

period (ns) = (*return-value*+n) * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7, and n = 1 for **clock(0)** or n = 0 otherwise. Also, the value *return-value* is equivalent to the *input-value* shown in *Usage*.

For Series 5000 and Series 6000 devices, the **period** I/O model measures the total period, from edge to edge, of an input signal in units of the clock period, calculated as follows:

period (ns) = (*return-value*+n) * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15, and n = 1 for **clock(0)** or n = 0 otherwise. Also, the value *return-value* is equivalent to the *input-value* shown in *Usage*.

You can use this model to implement digital-to-analog (D/A) converters, frequency counters, or tachometers. This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to measure the period from one rising or falling edge to the next corresponding edge on the input. The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*. This model is useful for instantaneous frequency or tachometer applications. Analog-to-digital conversion can be implemented using a voltage-to-frequency converter with this model (see **Figure 53**).

This I/O model is edge sensitive. The clock driving the internal counter in the Neuron Chip or Smart Transceiver is free running. The detection of active input edges stops and resets the counter each time.

The actual active edge of the input depends on whether the **invert** option is used in the declaration of the function block. The default is the negative edge.

Because the period function measures the delay between two consecutive active edges, the **invert** option has no effect on the returned value of the function for a repeating input waveform.

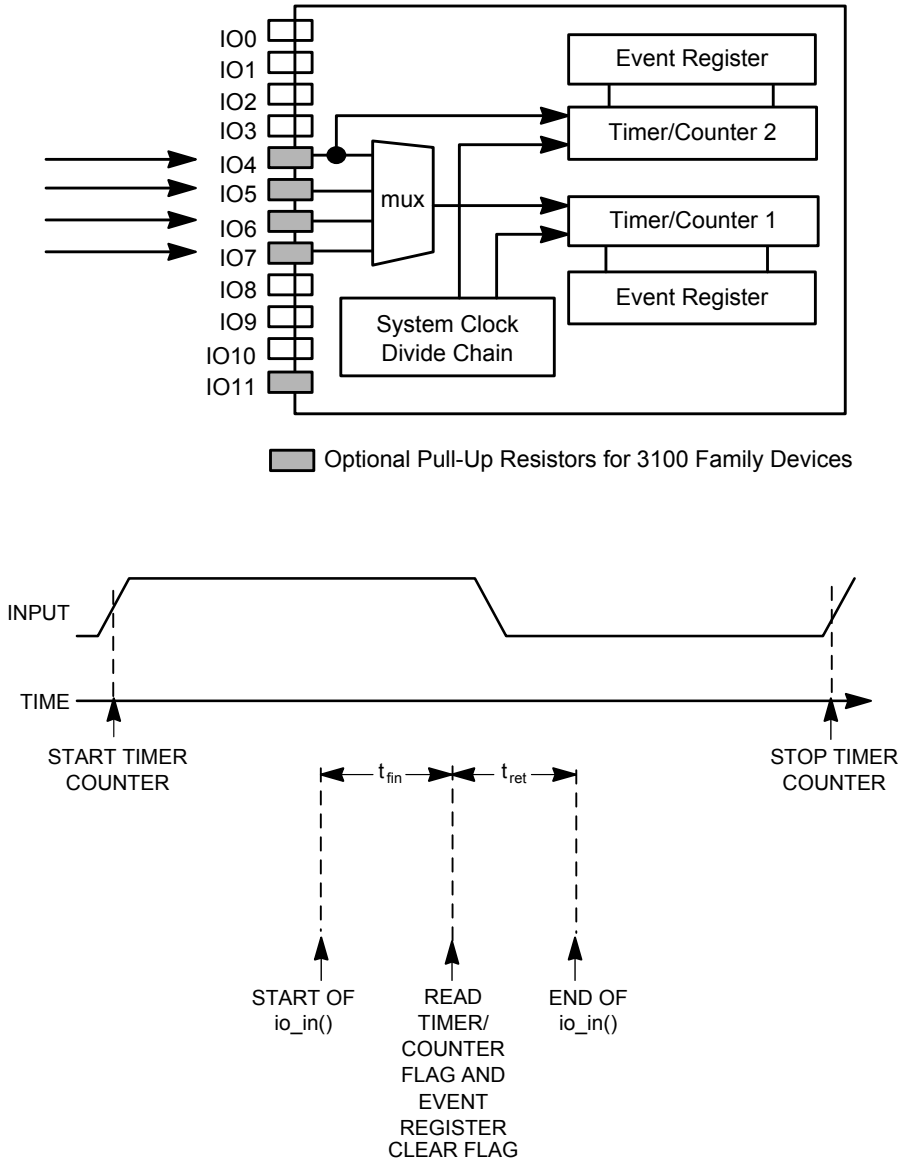


Figure 53. Period Input and Timing

Table 54. Period Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to input sample	86 µs
t_{ret}	Return from function	52 µs or 22 µs
Note: If the measurement is new, t_{ret} = 52 µs. If a new time is not being returned, t_{ret} = 22 µs.		

Programming Considerations

For *period-input*, the data type of the *return-value* for the **io_in( )** function is an **unsigned long**.

The input is latched every 50 ns for a Series 3100 device with a 40 MHz input clock, or every 12.5 ns for Series 5000 and Series 6000 devices with an 80 MHz system clock. This value scales inversely with the input clock speed. If no edges occur during the measuring period, an overflow condition occurs. The next **io_in( )** function call after the overflow has occurred returns the out-of-range value of 0xFFFF. The **io_update_occurs** event is not asserted as TRUE unless the program uses the **io_preserve_input( )** function after the **io_select( )** when using the multiplexed timer/counter, or in the reset task when using the dedicated timer/counter.

Syntax

pin [**input**] **period** [**mux** | **ded**] [**invert**] [**clock (***const-expr***)**] *io-object-name***;**

pin

An I/O pin. Period input can specify pins **IO_4** through **IO_7**.

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This keyword only applies, and must be used, when pin **IO_4** is the input pin.

The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert

Causes the measurement of time between positive edges and typically has no effect. By default, period input measures the time between negative edges.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on a Series 5000 or Series 6000 device, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()** value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value***;**

input-value = **io_in(***io-object-name***);**

Example

```
IO_4 input period mux clock(7) ioPeriod;

// END OF PERIOD:
when (io_update_occurs(ioPeriod)) {
  unsigned short timegap; // in tenths of a second

  // convert to tenths of sec
  timegap = (unsigned short)(io_in(ioPeriod) / 3906);
}
```

Pulsecount Input

The **pulsecount** I/O model counts the number of input edges at the input pin over a period of 0.8388608 seconds. You can use this model to perform average frequency measurements, implement tachometers, or control devices that require a precision count of pulses, such as stepper motors.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to count the number of input edges (up to 65535) in a fixed time (0.8388608 second) at all allowed input clock rates. Edges can be defined as rising or falling.

This I/O model is edge sensitive. The clock driving the internal counter in the Neuron Chip or Smart Transceiver is the actual input signal. The counter is reset automatically every 0.839 second.

The internal counter increments with every occurrence of an active input edge. Every 0.839 second, the content of the counter is saved and the counter is then reset to 0. This sequence is repeated indefinitely.

The actual active edge of the input depends on whether the **invert** option is used in the declaration of the function block. The default is the negative edge.

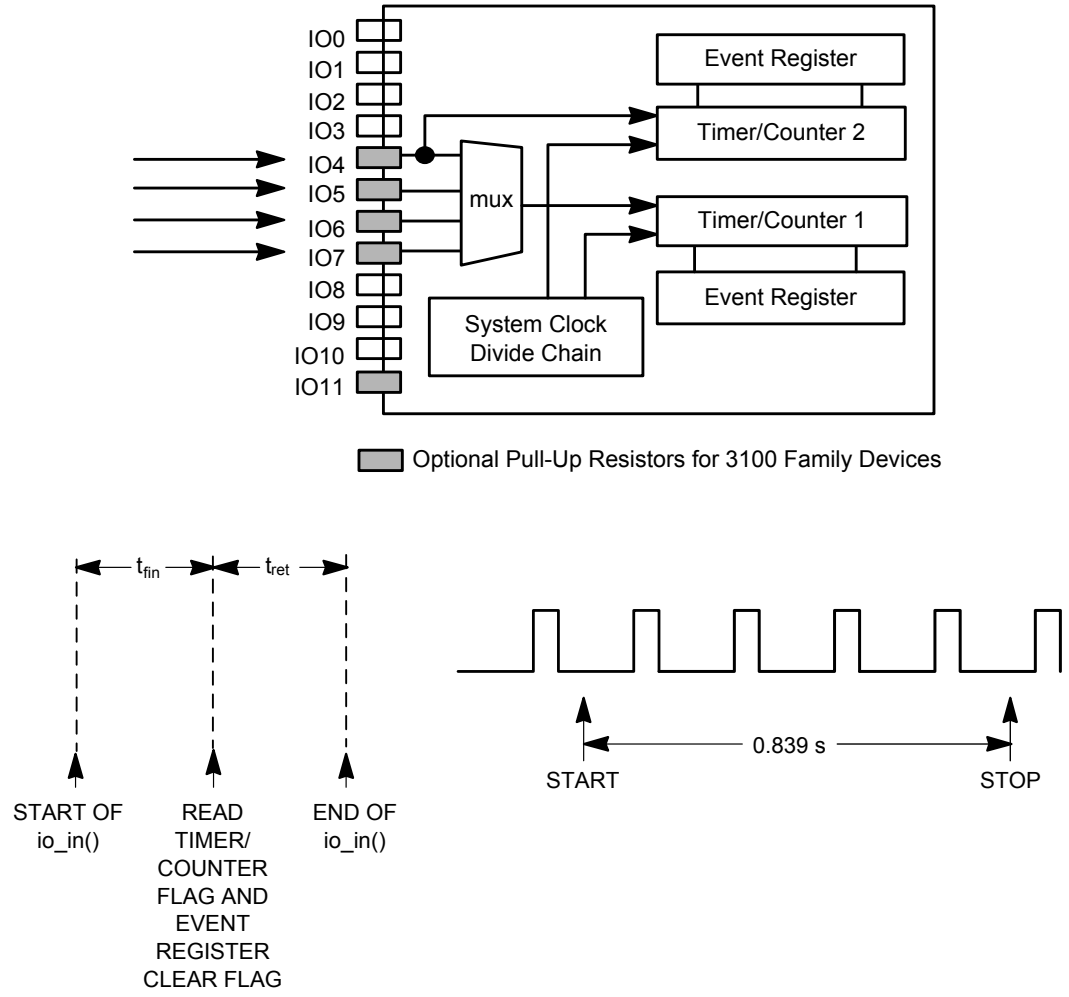


Figure 54. Pulsecount Input and Timing

Table 55. Pulsecount Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to input sample	86 µs
t_{ret}	Return from function	52 µs or 22 µs
Note: If the measurement is new, t_{ret} = 52 µs. If a new time is not being returned, t_{ret} = 22 µs.		

Programming Considerations

For pulsecount input, the data type of the return value for the **io_in( )** function is an **unsigned long**.

The input is latched every 50 ns for a Series 3100 device with a 40 MHz input clock, or every 12.5 ns for Series 5000 and Series 6000 devices with an 80 MHz system clock. This value scales inversely with the input clock. The value of a pulsecount input object is updated every 0.8388608 seconds and the **io_update_occurs** event becomes **TRUE**.

If no edges occur during the measuring period, an overflow condition occurs. The next **io_in( )** function call after the overflow has occurred will return the out-of-range value of 0xFFFF. The **io_update_occurs** event is not asserted as **TRUE** unless the program uses the **io_preserve_input( )** function after **io_select( )** when using the multiplexed timer/counter, or in the reset task when using the dedicated timer/counter.

Syntax

pin **input pulsecount** [**mux** | **ded**] [**invert**] *io-object-name***;**

pin

An I/O pin. Pulsecount input can specify pins **IO_4** through **IO_7**.

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This keyword is used only when pin **IO_4** is used as the input pin.

The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert

Causes positive edges to be counted. Typically this keyword has no effect because the number of positive edges equals the number of negative edges. By default, pulsecount input counts the number of negative input edges.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value***;**

input-value = **io_in(***io-object-name***);**

Example

```
IO_7 input pulsecount ioTotalTicks;
unsigned long ticks;

when (io_update_occurs(ioTotalTicks)) {
  ticks = input_value;
  // for up to 65535 ticks per 0.839 seconds
}
```

Quadrature Input

The **quadrature** I/O model is used to read a shaft or positional encoder input on two adjacent pins. You can use this model to monitor input data from shaft encoders for low-cost angular position input.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to count transitions of a binary Gray code input on two adjacent input pins. The Gray code is generated by peripheral devices such as shaft encoders and optical position sensors, which generate the bit pattern (00,01,11,10,00, …) for one direction of motion and the bit pattern (00,10,11,01,00, …) for the opposite direction. Reading the value of a quadrature object gives the arithmetic net sum of the number of transitions since the last time it was read (-16384 to 16383).

For Series 3100 devices, the maximum frequency of the input is one-quarter of the input clock rate, for example 2.5 MHz for a 10 MHz Smart Transceiver input clock. For Series 5000 devices, the maximum frequency of the input is one-half of the system clock rate, for example 5 MHz with a 10 MHz system clock.

Quadrature devices can be connected to timer/counter 1 through pins IO6 and IO7, and timer/counter 2 through pins IO4 and IO5 (see **Figure 47** and **Figure 55**). If the second input transitions low while the first input is low, and high while the first input is high, the counter counts up. Otherwise, the count is down.

A call to the **io_in( )** function returns the current value of the quadrature count since the last read operation. The counter is then reset and ready for the next series of input transitions. The count returned is a 16-bit signed binary number, capped at ±16K.

The number shown in the diagram above is the minimum time allowed between consecutive transitions at either input of the quadrature function block. For more information, see the, *Neuron Chip Quadrature Input Function Interface* engineering bulletin (part number 005-0003-01).

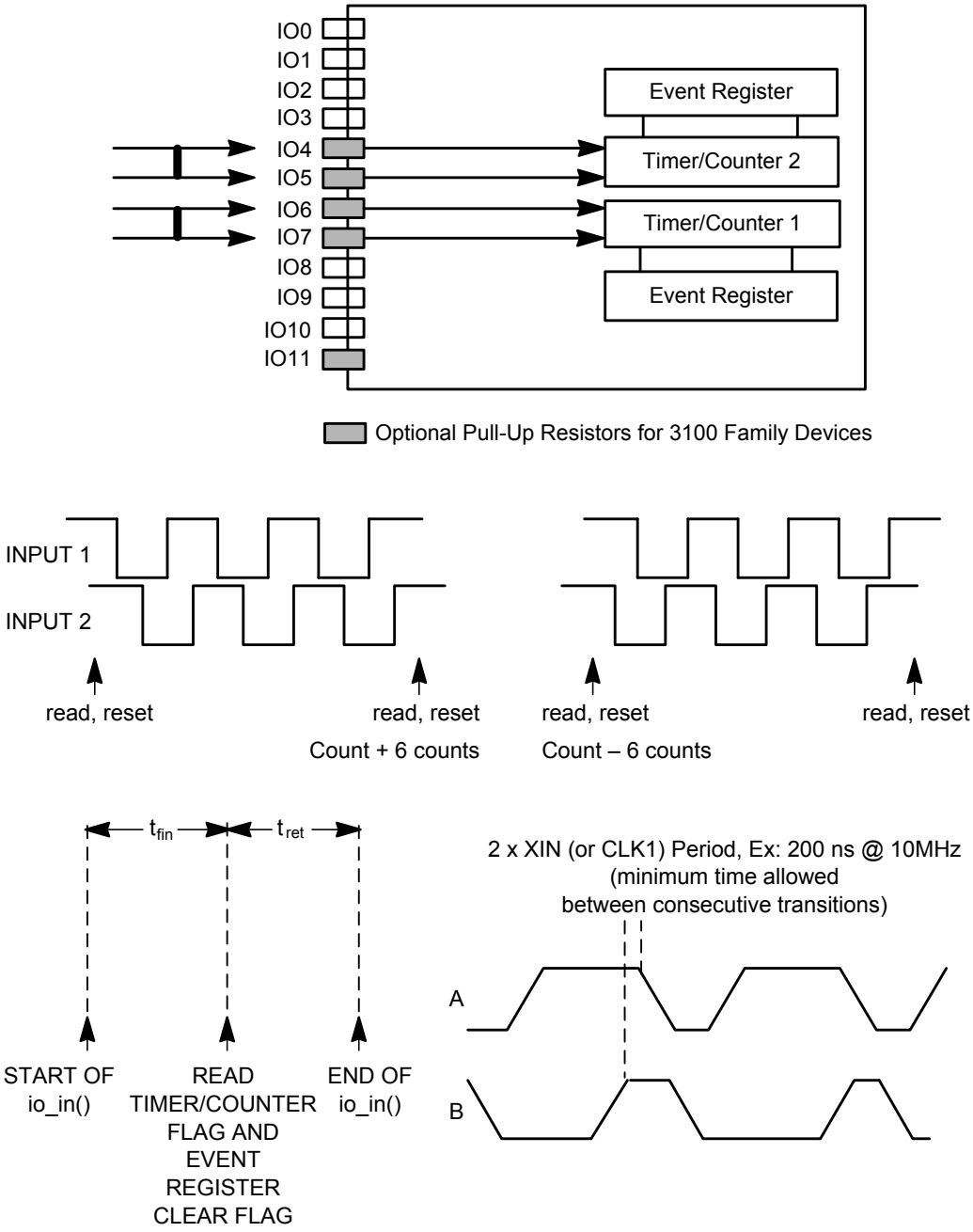


Figure 55. Quadrature Input and Timing

Table 56. Quadrature Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to input sample	90 µs
t_{ret}	Return from function	88 µs

Programming Considerations

A **signed long** value is returned from the **io_in( )** function, based on the change since the last input. The input is sampled every 50 ns for a Series 3100 device with a 40 MHz input clock, or every 12.5 ns for Series 5000 and Series 6000 devices with an 80 MHz system clock. This value scales inversely with the input clock speed.

For Series 3100 devices, add a **#pragma enable_io_pullups** directive to enable the Neuron Chip's or Smart Transceiver's built-in pull-up resistors.

For more information on quadrature input, see the *Neuron Chip Quadrature Input Function Interface* engineering bulletin (part number 005-0003-01).

Syntax

pin [**input**] **quadrature** *io-object-name***;**

pin

An I/O pin. Quadrature input requires two adjacent pins. The pin specification denotes the lower-numbered pin of the pair. The pin can be **IO_4** (which uses the dedicated timer/counter) or **IO_6** (which uses the multiplexed timer/counter).

Figure 56 shows the use of the two signal inputs A and B. Both edges of input A are counted. Input B indicates whether input A is moving in a positive or a negative direction.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

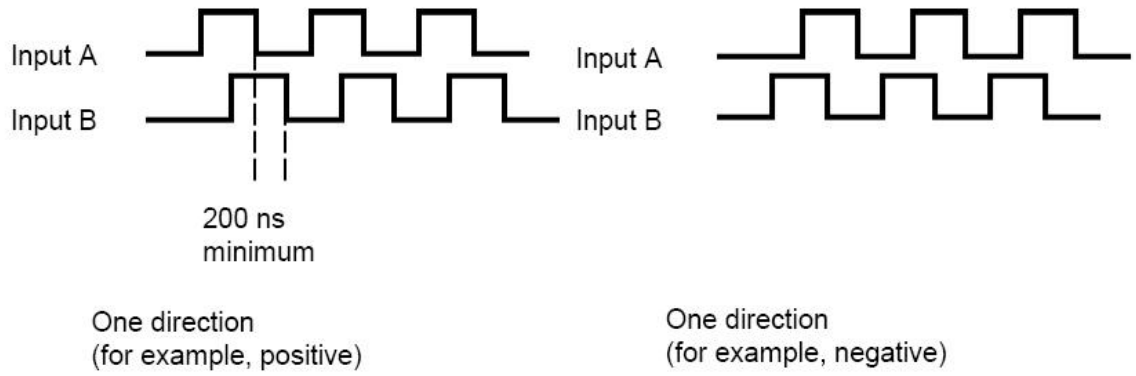


Figure 56. Quadrature Input

Usage

long *input-value***;**

input-value **= io_in(***io-object-name***);**

Example

```
IO_4 input quadrature ioDial;

long angle = 0;

when (io_update_occurs(ioDial)) {
  angle += input_value; // integrate angle in software
}
```

Totalcount Input

The **totalcount** I/O model counts the number of input edges at the input pin since the last **io_in( )** operation, or since initialization. Thus, this model can count external events, such as contact closures where it is important to keep an accurate running total (see **Figure 47** and **Figure 57**).

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to count either rising or falling input edges, but not both. Reading the value of a totalcount model gives the number of transitions since the last time it was read (0 to 65535). Maximum frequency of the input is one-quarter of the input clock rate for a Series 3100 device, or one-half of the system clock rate for Series 5000 and Series 6000 devices. For example, 2.5 MHz for a Series 3100 device at a maximum of 10 MHz input clock.

A call to the **io_in( )** function returns the current value of the totalcount value corresponding to the total number of active clock edges since the last call. The counter is then reset, and ready for the next series of input transitions.

The actual active edge of the input depends on whether the **invert** option is used in the declaration of the I/O object. The default is the negative edge.

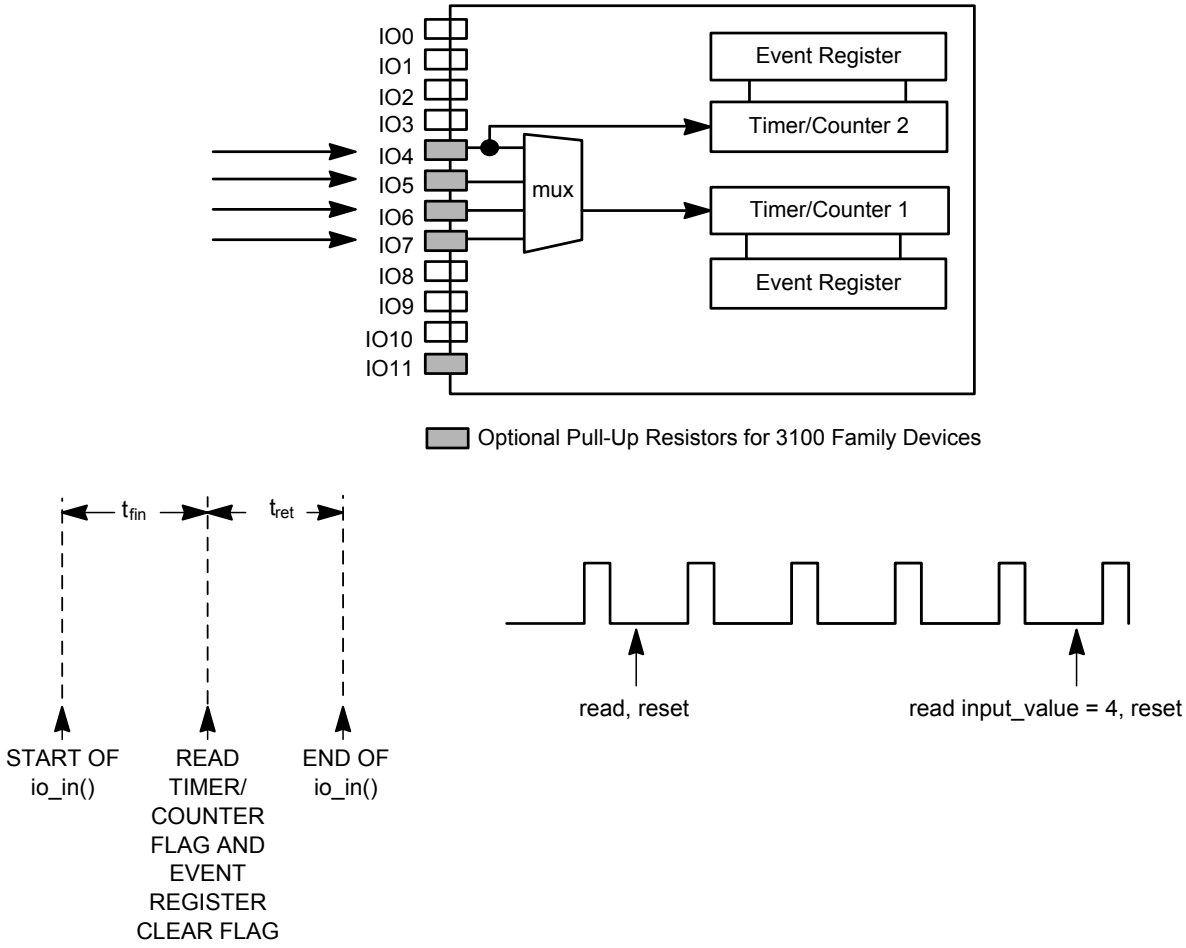


Figure 57. Totalcount Input and Timing

Table 57. Totalcount Input Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fin}	Function call to input sample	92 µs
t_{ret}	Return from function	61 µs

Programming Considerations

For totalcount input, the data type of **return_value** for the **io_in( )** function is an **unsigned long**.

The minimum duration for a high or low input signal for this I/O object is 50 ns for a Series 3100 device with a 40 MHz input clock, or 12.5 ns for Series 5000 and Series 6000 devices with an 80 MHz system clock. This value scales inversely with the input clock speed.

Syntax

pin [**input**] **totalcount** [**mux** | **ded**] [**invert**] *io-object-name***;**

pin

An I/O pin. Totalcount input can specify pins **IO_4** through **IO_7**.

mux | ded

Specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This keyword is used only when pin **IO_4** is used as the input pin.

The **mux** keyword assigns the I/O object to the multiplexed timer/counter. The **ded** keyword assigns the I/O object to the dedicated timer/counter. The multiplexed timer/counter is always used for pins **IO_5** through **IO_7**.

invert

Causes positive edges to be counted. By default, totalcount input counts the number of negative input edges.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input-value***;**

input-value = **io_in(***io-object-name***);**

Example

```
IO_4 input totalcount ded ioEventCount;
unsigned long events = 0;
mtimer repeating tick = 100;

when (timer_expires(tick)) {
  events += io_in(ioEventCount);
  // this sums up all events since initialization-time
}
```

6

Timer/Counter Output Models

This chapter describes timer/counter output models. Timer/counter I/O models use a timer/counter circuit in the Neuron Chip or Smart Transceiver. Each Neuron Chip and each Smart Transceiver has two timer/counter circuits: One whose input can be multiplexed, and one with a dedicated input.

Edgedivide Output

The **edgedivide** I/O model is used to control an output pin by toggling its logic state every **output_value** negative edges on an input pin. This toggling results in a acts as a frequency divider by providing an output frequency on either pin IO0 or IO1: divide-by-n*2 counter, where *n* is the value defined by the **output_value** argument.

The output frequency is a divided-down version of the input frequency applied on pins IO4 – IO7. This object is useful for any divide-by-n operation, where *n* is passed to the timer/counter object through the application program and can be from 1 to 65 535. The value of 0 forces the output to the off level and halts the timer/counter.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

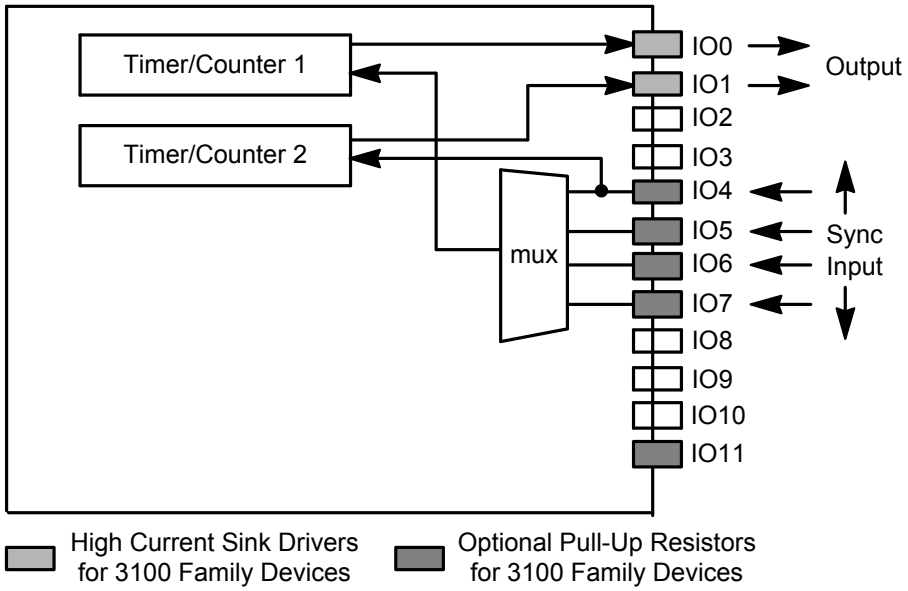
A new divide value does not take effect until after the output toggles, with two exceptions:

- If the output is initially disabled, the new (non-zero) output starts immediately after t_{fout}
- For a new divide value of 0, the output is disabled immediately

Normally, the negative edges of the input sync pulses are the active edge. Using the **invert** keyword in the object declaration makes the positive edge active.

The initial state of the output pin is logic 0 by default. This initial state can also be changed to logic 1 through the object declaration.

Figure 58 shows the pinout and timing information for this output model.



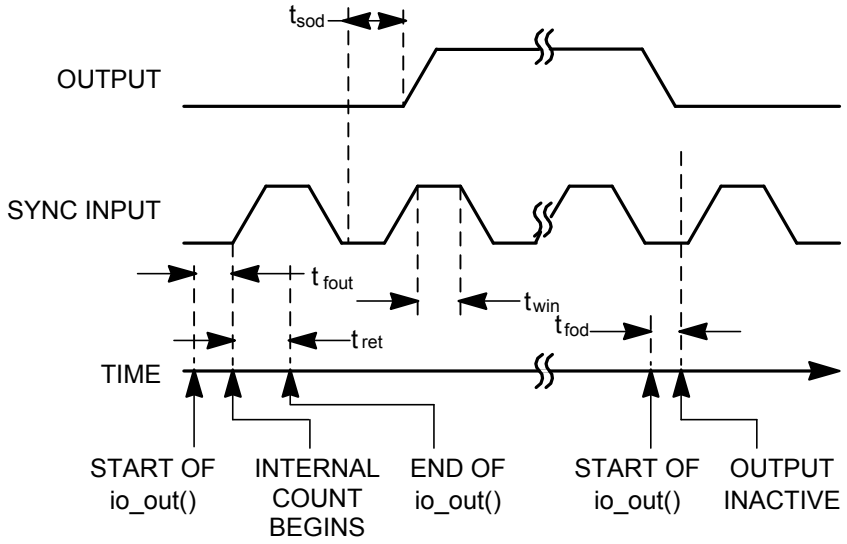


Figure 58. Edgedivide Output and Timing

Table 58. Edgedivide Output Latency Values for Series 3100 Devices

Symbol	Description	Minimum	Typical	Maximum
t_{fout}	Function call to start of timer	—	96 µs	—
t_{fod}	Function to output disable	—	82.2 µs	—
t_{sod}	Active sync edge to output toggle	550 ns	—	750 ns
t_{win}	Sync input pulse width (10 MHz)	200 ns	—	—

Symbol	Description	Minimum	Typical	Maximum
t_{ret}	Return from function	—	13 µs	—

Programming Considerations

For **edgedivide** output, the data type of the output value for **io_out( )** is an **unsigned long**. Following reset of the Neuron Chip or Smart Transceiver, the divider is disabled until the first call to the **io_out( )** function. The first call to the **io_out( )** function for the **edgedivide** output model sets the output pin high and starts the divider. When the divider is running, the function call to **io_out( )** only sets the value used for the divider and does not affect the state of the output pin. However, when the output value is 0, the output signal is forced to a low state and the divider is halted.

Syntax

pin **[output] edgedivide sync (***pin-nbr***)** [**invert**] *io-object-name*
[**=** *initial-output-level*]**;**

pin

An I/O pin. Edgedivide output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the multiplexed timer/counter is used and the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the dedicated timer/counter is used and the sync pin must be **IO_4**.

sync (*pin-nbr***)**

Specifies the sync pin, which is the counting input signal. By default, the divider counts negative edges.

invert

This keyword causes positive edges at the sync pin input to be counted rather than the default negative edges.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_0 output edgedivide sync(IO_4) ioDivider;
...

when (reset) {
  // There is a 60Hz signal at pin IO_4.
  // Set up the divider to produce
  // a change on pin IO_0 once a minute.
  io_out(ioDivider, 3600UL);
}
```

Frequency Output

For Series 3100 devices, the **frequency** I/O model produces a repeating square wave output signal whose period is a function of **output_value** and the selected clock value:

period (ns) = (output_value+n) * 4000 * 2^(clock)/ input_clock (MHz)

where clock ranges from 0..7, and n = 1 for **clock(0)** or n = 0 otherwise.

For Series 5000 and Series 6000 devices, the **frequency** I/O model produces a repeating square wave output signal whose period is a function of **output_value** and the selected clock value:

period (ns) = (output_value+n) * 4000 * 2^(value)/ 10 MHz

where value ranges from 0..15, and n = 1 for **clock(0)** or n = 0 otherwise.

You can use this I/O model for frequency synthesis to drive an audio transducer or to drive a frequency-to-voltage converter to generate an analog output.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to generate a continuous square wave of 50% duty cycle. Writing a new frequency value to the device takes effect at the end of the current cycle. This object is useful for frequency synthesis to drive an audio transducer, or to drive a frequency to voltage converter to generate an analog output (see **Figure 59** and **Figure 60**).

The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*.

A new frequency output value does not take effect until the end of the current cycle, with two exceptions:

- If the output is disabled, the new (non-zero) output starts immediately after t_{fout}
- For a new output value of zero, the output is disabled immediately and not at the end of the current cycle

A disabled output is a logic zero by default unless the **invert** keyword is used in the I/O object declaration. The resolution and range for this object scale with Neuron Chip or Smart Transceiver input clock rate.

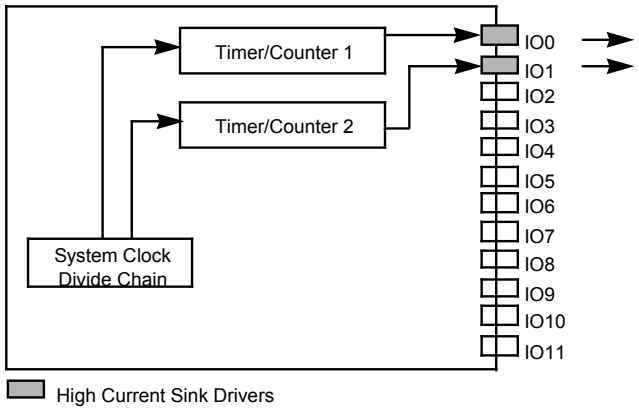


Figure 59. Frequency Output

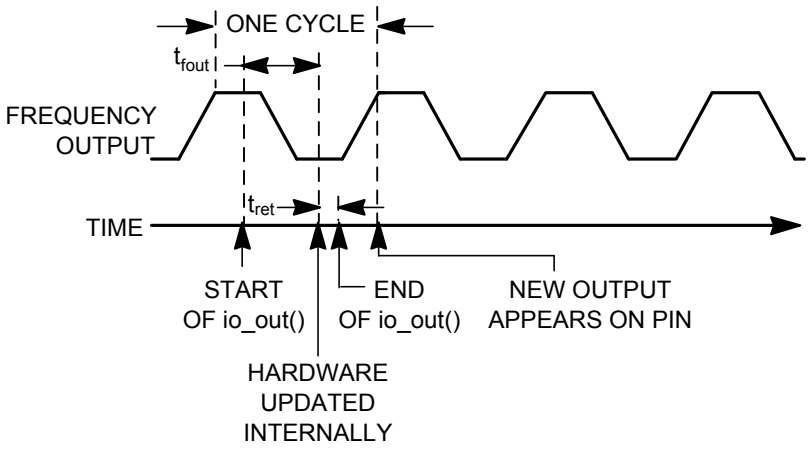


Figure 60. Frequency Output Timing

Table 59. Frequency Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to output update	96 µs
t_{ret}	Return from function	13 µs

Programming Considerations

For frequency output, the data type of **output_value** for **io_out( )** is an **unsigned long**. An **output_value** of 0 forces the output signal to a low state (unless the **invert** keyword is used in the declaration).

Syntax

pin [**output**] **frequency** [**invert**] [**clock (***const-expr***)**] *io-object-name* [**=***initial-output-level*]**;**

pin

Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).

invert

This keyword inverts the output for an output value of 0. The default output for 0 is low.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on a Series 5000 or Series 6000 device, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()** value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_1 output frequency clock(3) ioAlarm;
...
```

```
when (...) {
  io_out(ioAlarm, 100);
  // outputs 3.125kHz signal at clock(3)
}

when (...) {
  io_out(ioAlarm, 50);
  // outputs 6.25kHz signal at clock(3)
}

when (...) {
  io_out(ioAlarm, 0);
  // output signal is stopped
}
```

Infrared Pattern Output

An **infrared_pattern** I/O model produces a series of timed repeating square wave output signals. The frequency of the square wave output is controlled by the application. Normally, this frequency is the modulation frequency used for infrared transmission.

This I/O model is useful for driving an infrared LED to provide infrared control of devices that support infrared remote control. For example, for a Series 3100 device with a 10 MHz input clock, a **clock(1)** configuration and an *output-frequency* value of 33 results in a 37.878 kHz (38 kHz) modulation signal.

This model applies to 3120 Power Line Smart Transceivers, 3150 Power Line Smart Transceivers, 3170 Power Line Smart Transceivers, Series 5000 Neuron Processors and Smart Transceiver, and to Series 6000 Neuron Processors and Smart Transceiver.

Hardware Considerations

The pattern of the modulation frequency is controlled by the application, which specifies how long the output is active and how long the output is idle. This pattern is then repeated to produce a sequence of frequency output bursts separated by idle periods.

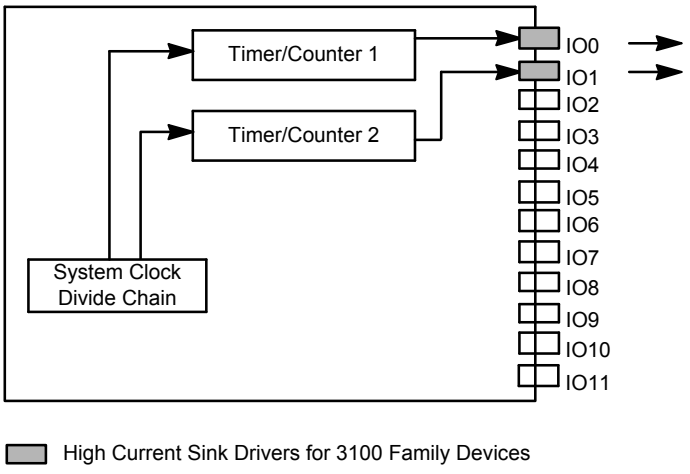


Figure 61. Infrared Pattern Output

Programming Considerations

The frequency of the square wave output is controlled by the *clock-expr* setting and by the **unsigned long** *output-frequency* value passed to the **io_out( )** function. The pattern of this modulation frequency is controlled by an array of **unsigned long** timing values, also passed to the **io_out( )** function:

- The first value in this array controls the length of the first burst of modulation frequency signal output—the output is active for this period.
- The second value in this array controls the length of an absence of the modulation frequency signal—the output is idle for this period.

This pattern is then repeated by subsequent values in the array to produce a sequence of frequency output bursts separated by idle periods. This array is similar to the array generated by the **edgelog** input model.

The values in the *timing-table* array control the on and off time of the modulation frequency output. This timing also is a product of the Neuron input clock, and is:

On/off period (μs) = (25.2 * value + 29.4) * *S*

In the formula above, the scaling factor *S* is determined by the input clock, as shown in **Table 60**.

Table 60. Determining S

S	Input Clock Rate (Series 3100)	System Clock Rate (Series 5000)
0.063	—	80 MHz
0.125	—	40 MHz
0.25	40 MHz	20 MHz
0.5	20 MHz	10 MHz
1	10 MHz	5 MHz
1.5259	6.5536 MHz	—
2	5 MHz	—
4	2.5 MHz	—
8	1.25 MHz	—
16	625 kHz	—

The square wave output state is always toggled, between idle (off) and active (on), at the end of the **io_out( )** function. Typically, the number of elements in the *timing-table* should always be an odd number, which will result in the output being toggled to idle (turned off) at the end of the **io_out( )** function. The last

element of the *timing-table* controls the last active period before toggling to idle (off) and returning from the **io_out( )** function. If the number of elements in the timing table is even, the output will be toggled on at the end of the **io_out( )** function, which is typically not the desired behavior.

Syntax

pin [**output**] **infrared_pattern** [**invert**] [**clock(***clock-expr***)**] *io-object-name* [**=** *initial-output-level*] **;**

pin

Specifies a Neuron output pin. The value can be **IO_0** or **IO_1**.

invert

Set this option to specify that the output pin is idle at 1. Otherwise, the output pin is idle at 0.

clock(*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for **infrared_pattern** output is **clock(0)**. You can use the **io_set_clock( )** function to change the clock at run time. **Table 61** shows the clock values for a Series 3100 device with a 10 MHz input clock and Series 5000 and Series 6000 devices with an 80 MHz system clock (the values scale inversely proportional to the input or system clock).

Table 61. Clock Values

Clock	Range and Resolution Period	
	Series 3100 (10 MHz Clock)	**Series 5000 (80 MHz Clock)**
0 (default)	800 ns to 26.21 ms in steps of 400 ns (1-65535)	12.8 µs to 419.36 ms in steps of 25 ns (1-65535)
1	0 to 52.42 ms in steps of 800 ns (0-65535)	0 to 838.72 ms in steps of 50 ns (0-65535)
2	0 to 104.86 ms in steps of 1.6 µs	0 to 1.677 sec in steps of 100 ns
3	0 to 209.71 ms in steps of 3.2 µs	0 to 3.355 sec in steps of 200 ns
4	0 to 419.42 ms in steps of 6.4 µs	0 to 6.71 sec in steps of 400 ns
5	0 to 838.85 ms in steps of 12.8 µs	0 to 13.42 sec in steps of 800 ns
6	0 to 1.677 sec in steps of 25.6 µs	0 to 26.84 sec in steps of 1.6 µs
7	0 to 3.355 sec in steps of 51.2 µs	0 to 53.68 sec in steps of 3.2 µs

io-object-name

Specifies a name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

unsigned *count*;
unsigned long *output-frequency*, *timing-table*[*count*];

io_out(*io-object-name*, *output-frequency*, *timing-table*, *count*);
(There is no return value for the function.)

Example

```
IO_0 output infrared_pattern ioIrOut;

unsigned long timing [5]={395,395,783,783,395};
unsigned long frequency = 62;

when (...) {
  io_out(ioIrOut, frequency, timing, 5);
}
```

Oneshot Output

The **oneshot** I/O model produces output pulses of a specified period or duty cycle. That is, for Series 3100 devices, it can produce a single output pulse whose duration is a function of the output value and the selected clock value, calculated as follows:

duration (ns) = output_value * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, it can produce a single output pulse whose duration is a function of the output value and the selected clock value, calculated as follows:

duration (ns) = output_value * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

You can use this I/O model to implement digital-to-analog (D/A) converters or to control any device with a pulsewidth modulated input.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to generate a single pulse of programmable duration. The asserted state can be either logic high or logic low. Retriggering the oneshot before the end of the pulse causes it to continue for the new duration. The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range*. This object is useful for generating a time delay without intervention of the application processor (see **Figure 62**).

While the output is still active, a subsequent call to this function cause the update to take effect immediately, extending the current cycle. This is, therefore, a retriggerable oneshot function.

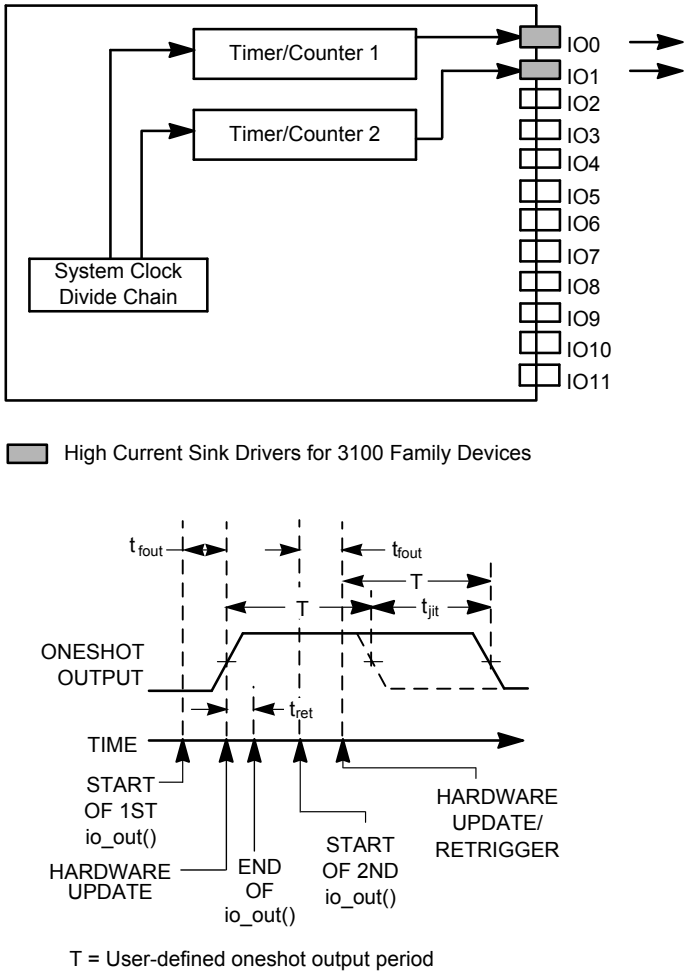


Figure 62. Oneshot Output and Timing

Table 62. Oneshot Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to output update	96 µs
t_{ret}	Return from function	13 µs
t_{jit}	Output duration jitter	—
Note: The maximum value for t_{jit} is 1 timer/counter clock period.		

Programming Considerations

The **oneshot** I/O model can be retriggered. A call to the **io_out( )** function for a oneshot object starts a new pulse, even if one is currently in progress.

For **oneshot** output, the data type of the output value for the **io_out( )** function is an **unsigned long**. An output value of zero (0) forces the output to a low state.

Syntax

pin [**output**] **oneshot** [**invert**] [**clock (***const-expr***)**] *io-object-name*
[**=***initial-output-level*]**;**

pin

Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).

invert

Causes the output to be inverted, producing a signal that is normally high with low pulses. The default is normally low with high pulses.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on Series 5000 or Series 6000 devices, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()**

value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_0 output oneshot ioFlash;
unsigned long pulse = 39062;  // 1 second pulse

mtimer repeating flashTimer;

when (...) {
  // start timer, flash every 2 secs
  flashTimer = 2000;
}

when (timer_expires(flashTimer)) {
  // outputs a 1 sec pulse
  io_out(ioFlash, pulse);
}
```

Pulsecount Output

For Series 3100 devices, the **pulsecount** I/O model produces a sequence of pulses whose period is a function of the clock period, calculated as follows:

period (ns) = 256 * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, the **pulsecount** I/O model produces a sequence of pulses whose period is a function of the clock period, calculated as follows:

period (ns) = 256 * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

This I/O model can perform average frequency measurements and implement tachometers. You can use the pulsecount input to control devices that require a precision count of pulses, such as stepper motors.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to generate a series of pulses. The number of pulses output is in the range 0 to 65535, and the output waveform is a square wave with 50% duty cycle. This function suspends the current application context until the pulse train is complete. See *Timer/Counter Square Wave Output* for the frequency of the waveform for various clock select values. This model is useful for external counting devices that can accumulate pulse trains, such as stepper motors (see **Figure 63**).

The **io_out( )** function does not return until all output pulses have been produced. t_{fout} is the time from function call to first output pulse. Therefore, the calling of this function ties up the application processor for a period of N x (pulse period) + t_{fout} + t_{ret}, where N is the number of specified output pulses.

The polarity of the output depends on whether the **invert** option is used in the declaration of the function block. The default is low with high pulses.

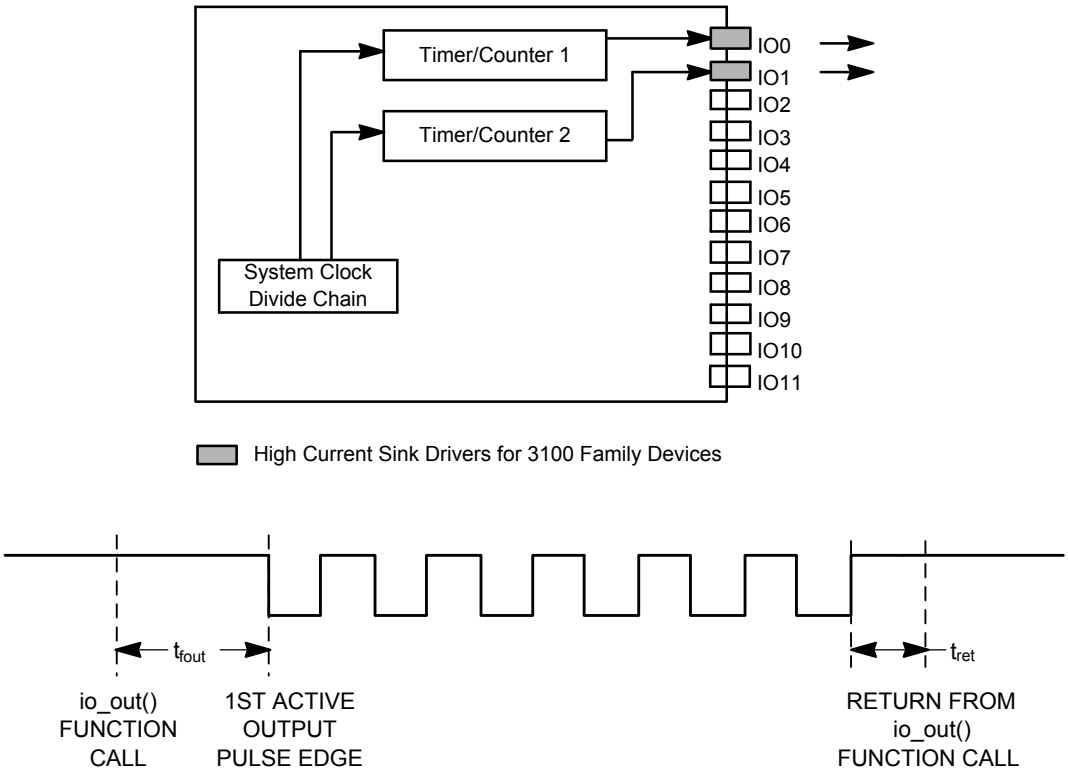


Figure 63. Pulsecount Output and Timing

Table 63. Pulsecount Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to first active output pulse edge	115 µs
t_{ret}	Return from function	5 µs

Programming Considerations

The **output_value** determines the number of pulses output. When this I/O model is used, the **io_out( )** function call does not return until all pulses have been produced. This process ties up the application processor for the duration of the pulsecount.

For pulsecount output, the data type of the output value for the **io_out( )** function is an **unsigned long**. An output value of 0 forces the output signal to its normal state.

Syntax

pin **output pulsecount** [**invert**] [**clock (***const-expr***)**] *io-object-name* [**=***initial-output-level*]**;**

pin

Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).

invert

Causes the signal to be inverted, normally high with low pulses. By default, the signal is normally low with high pulses.

clock (*const*-expr**)**

Specifies a clock in the range 1 to 7, where 1 is the fastest clock and 7 is the slowest clock. The default clock for pulsecount output is clock 7. The **io_set_clock( )** function can be used to change the clock.

Specifying clock 0 for the **io_set_clock( )** function results in an unspecified number of counts, because 0 is not a valid clock for pulsecount output.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of how the **io_set_clock( )** function affects the resolution and range of certain timer/counter I/O models.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

unsigned long *output-value***;**

io_out(*io-object-name*, *output-value***);**

Example

```
IO_1 output pulsecount ioTrainOut;

when (...) {
  // will produce 100 pulses on pin 1
  // each pulse of period 6.554 milliseconds
  io_out(ioTrainOut, 100);
}
```

Pulsewidth Output

For Series 3100 devices, the **pulsewidth** I/O model produces output pulses of a specified period or duty cycle to create a repeating waveform whose duty cycle is a function of *output-value* and whose period is a function of the clock period, calculated as follows:

pulsewidth (ns) = *output-value* * 2000 * 2^(clock) / input_clock (MHz)

total_period (ns) = 256 * 2000 * 2^(clock) / input_clock (MHz)

where clock ranges from 0..7

For Series 5000 and Series 6000 devices, the **pulsewidth** I/O model produces output pulses of a specified period or duty cycle to create a repeating waveform whose duty cycle is a function of *output-value* and whose period is a function of the clock period, calculated as follows:

pulsewidth (ns) = *output-value* * 2000 * 2^(value) / 10 MHz

total_period (ns) = 256 * 2000 * 2^(value) / 10 MHz

where value ranges from 0..15

You can use this I/O object to implement digital-to-analog (D/A) converters or to control any device with a pulsewidth-modulated input.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to generate a pulsewidth modulated repeating waveform. In pulsewidth short function, the duty cycle ranges from 0% to 100% (0/256 to 255/256) of a cycle, in steps of about 0.4% (1/256). See *Timer/Counter Square Wave Output* for the frequency of the waveform for various clock values.

In pulsewidth long function, the duty cycle ranges from 0% to almost 100% (0/65536 to 65535/65536) of a cycle in steps of 15.25 ppm (1/65536). See *Timer/Counter Pulsetrain Output* for the frequency of the waveform for various clock values. The asserted state of the waveform can be either logic high or logic low. Writing a new pulsewidth value to the device takes effect at the end of the current cycle. A pulsewidth modulated signal provides a simple means of digital-to-analog conversion (see **Figure 64**).

The new output value does not take effect until the end of the current cycle, with two exceptions:

- If the output is disabled, the new (non-zero) output starts immediately after t_{fout}
- For a new output value of zero, the output is disabled immediately and not at the end of the current cycle

A disabled output is a logic 0 by default, unless the **invert** keyword is used in the I/O object declaration.

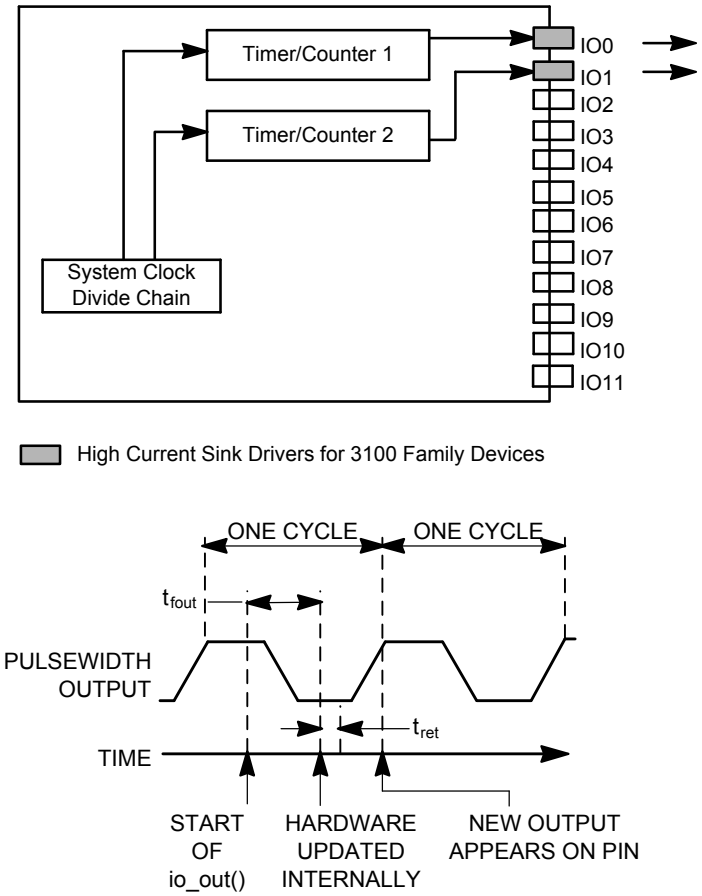


Figure 64. Pulsewidth Output and Timing

Table 64. Pulsewidth Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to output update	101 µs
t_{ret}	Return from function	13 µs

Programming Considerations

For 8-bit pulsewidth output, the data type of *output-value* for the **io_out( )** function is an **unsigned short**. An *output-value* of 0 results in a 0% duty cycle. A value of 255 (the maximum value allowed) results in a 100% duty cycle. The duty cycle of the pulse train is (*output-value*/256), except when *output-value* is 255; in that case, the duty cycle is 100%.

For 16-bit pulsewidth output, the data type of *output-value* for the **io_out( )** function is an **unsigned long**. An *output-value* of 0 results in a 0% duty cycle. A value of 65535 (the maximum value allowed) results in a 99.998% duty cycle. The duty cycle of the pulse train is (*output-value*/65536).

Important: Do not use an *output-value* of 1 in combination with **clock(0)**.

Syntax

pin [**output**] **pulsewidth** [**short** | **long**] [**invert**] [**clock (***const-expr***)**] *io-object-name*

[**=***initial-output-level*]**;**

pin

Specifies either pin **IO_0** (using the multiplexed timer/counter) or **IO_1** (using the dedicated timer/counter).

short | long

Resolution of the data value: **short** specifies 8-bit pulsewidth output, **long** specifies 16-bit. If neither of these options is specified, the **pulsewidth** I/O object defaults to the 8-bit (short) mode.

invert

Causes the output signal to be inverted, normally high for a 0% duty cycle. By default, the output signal is normally low for a 0% duty cycle.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for pulsewidth output is clock 0. The **io_set_clock( )** function can be used to change the clock.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of how the **io_set_clock( )** function affects the resolution and range of certain timer/counter I/O models.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

unsigned int *output-value***;** // for 8-bit output
unsigned long *output-value***;** // for 16-bit output

io_out(*io-object-name***,** *output-value***);**

Example

```
IO_1 output pulsewidth clock(7) ioDimmer;

mtimer repeating tick;
unsigned short brightness;

when (...) {
  tick = 10; // start clock for fading
  brightness = 255; // start brightness for fading
}

when (timer_expires(tick_timer)) {
  io_out(ioDimmer, --brightness);
  if (brightness == 0) {
    tick = 0;
    // turn off the timer
  }
}
```

Stretched Triac Output

The **stretchedtriac** I/O model is used to control the delay of an output pulse signal with respect to an input trigger signal. For control of AC circuits using a **stretchedtriac** I/O object, the sync input is typically a zero-crossing signal, and the pulse output is the triac trigger signal. The output pulse width is programmatically controlled, normally active low, but it can be inverted. The pulse width is independent of the Neuron input clock.

You can use this I/O model to control AC circuits that use a triac device, such as lamp dimmers.

This model applies to Series 5000 and Series 6000 Neuron Processors and Smart Transceivers.

For Series 3100 devices, see *Triac Output*.

Comparing Stretched Triac Output to Triac Output

Figure 65 shows basic triac operation for a Series 3100 device using the **triac** I/O object. For a Series 3100 device, the turn-on pulse has a fixed duration of 25.6 µs, which is sufficient to control many triac devices.

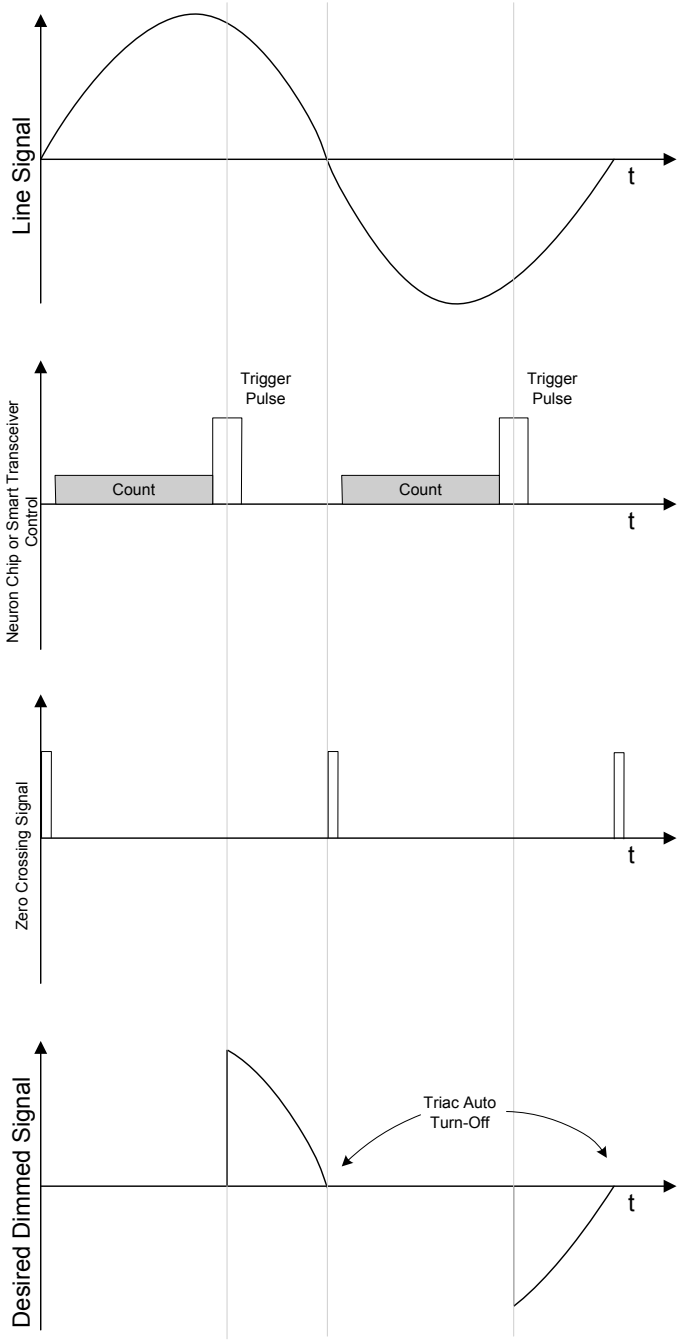


Figure 65. Series 3100 Triac Output

Figure 66 shows basic triac operation for a Series 5000 and Series 6000 device using the **stretchedtriac** I/O object. For Series 5000 and Series 6000 devices, the turn-on pulse has a programmatically controlled duration. By increasing the

turn-on pulse duration, the **stretchedtriac** I/O object can control triac devices that run under highly inductive loads.

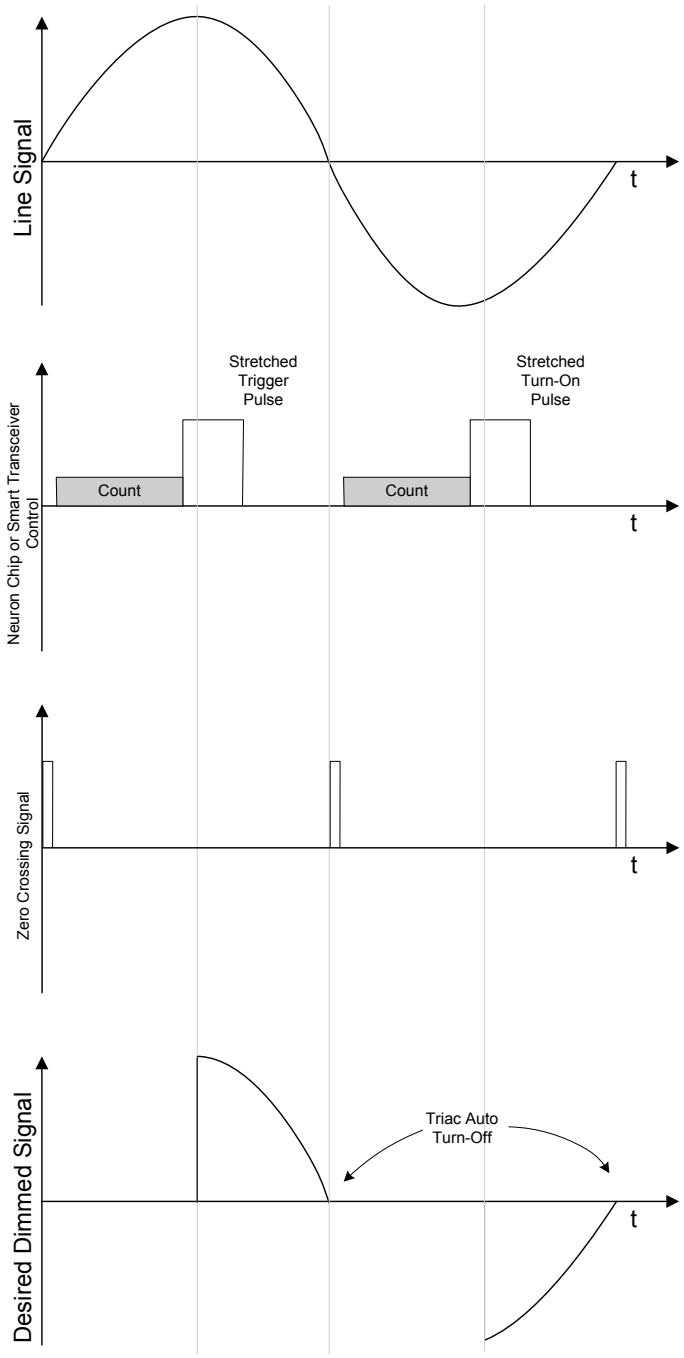


Figure 66. Series 5000 and Series 6000 Stretched Triac Output

Hardware Considerations

On a Smart Transceiver, a timer/counter can be configured to control the delay of an output signal with respect to a synchronization input. This synchronization

can occur on the rising edge, the falling edge, or both the rising and falling edges of the input signal. For control of AC circuits using a triac device, the sync input is typically a zero-crossing signal, and the pulse output is the triac trigger signal.

The output gate pulse is asserted after the control period and is deasserted at or near the next sync input point. Although the input trigger signal (zero crossing) is asynchronous relative to the internal clock, there is minimal jitter, t_{jit}, associated with the output gate pulse.

The actual active edge of the sync input and the triac gate output can be set by using the **clockedge** or **invert** parameters, respectively.

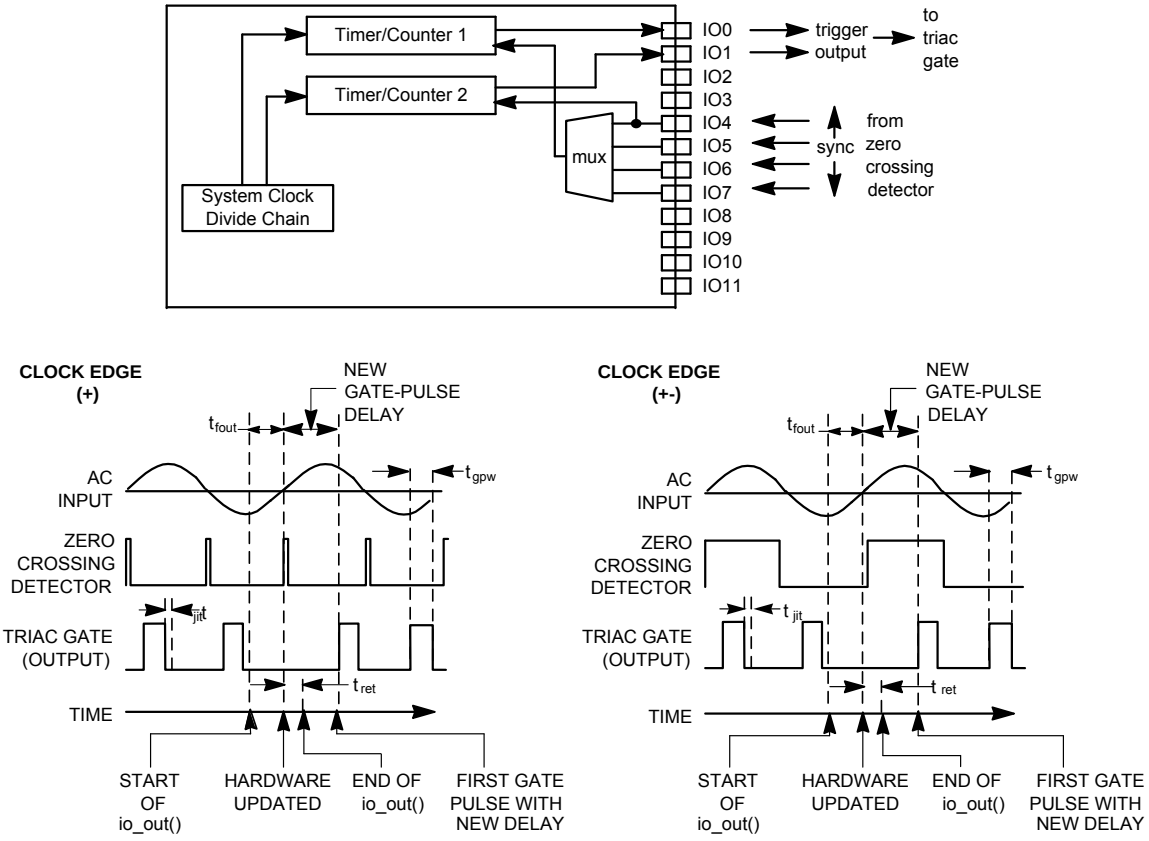


Figure 67. Stretched Triac Output and Timing

The hardware update does not happen until the occurrence of an external active sync clock edge. The internal timer is then enabled, and a triac gate pulse is generated after the user-defined period has elapsed. This sequence is repeated indefinitely until another update is made to the triac gate pulse delay value.

t_{fout} (min) refers to the delay from the initiation of the function call to the first sampling of the sync input. In the absence of an active sync clock edge, the input is repeatedly sampled for 10 ms (1/2 wave of a 50 Hz line cycle time), t_{fout} (max), during which the application processor is suspended.

Programming Considerations

Execution of this I/O object type is synchronized with the sync pin input and might not return for up to 10 ms. That is, the application program could be delayed for as long as 10 ms. Because of this synchronization, the frequency of

the sync pin input (and the frequency of the AC circuit being controlled) is limited to the 50-60 Hz range.

Syntax

pin [**output**] **stretchedtriac sync (***pin-nbr***)** [**clockedge (+) | (-) | (+-)**] **frequency**(*value*) *io-object-name***;**

pin

An I/O pin. Stretched triac output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the sync pin must be **IO_4**.

sync (*pin-nbr***)**

Specifies the sync pin, which is the input trigger signal.

clockedge (+) | (-) | (+-)

(+) Causes the sync input to be positive-edge sensitive.

(-) Causes the sync input to be negative-edge sensitive. This is the default.

(+-) Causes the sync input to be both positive- and negative-edge sensitive.

frequency(*value***)**

Specifies the frequency of the sync pin input. Valid values range from 40 to 70, with the most usual values' being 50 (for 50 Hz input) or 60 (for 60 Hz input).

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned short *control-value***;**

io_out(*io-object-name***,** *control-value***);**

The **io_out( )** function for a **stretchedtriac** I/O object uses a *control-value* parameter to specify duration of the triac trigger pulse. This 8-bit control value allows you to stretch the triac trigger pulse to any width within the half-cycle (between zero crossings). The control value is related to the number of clock cycles between zero crossings, and depends on the AC frequency (50 or 60 Hz) that the triac is controlling. Valid values are:

- 0-193 specifies an amount of stretching for 50 Hz control frequency
- 0-160 specifies an amount of stretching for 60 Hz control frequency

The 193 or 160 values represent a minimum stretching of the triac control signal (the triac is pulsed on for the minimum time before the control signal's zero crossing – for example, a light is almost completely dim). A 0 value represents maximum stretching of the triac control signal.

The **io_set_terminal_count( )** function allows the application to change the terminal count for the stretched triac I/O object at runtime. This function allows a device to:

- Have a single application for both 50 Hz and 60 Hz power domains
- Operate at a non-standard power line frequency
- Provide higher-than-typical tolerances to changes in frequency

The application can determine the current values for frequency at runtime, and use this function to adjust the triac on-time as needed.

Example

```
IO_0 output stretchedtriac sync (IO_5) frequency(60)
ioTriac;

when (...) {
  io_out(ioTriac, 160);  // full on
}

when (...) {
  io_out(ioTriac, 80);  // half on
}

when (...) {
  io_out(ioTriac, 0);   // full off
}
```

Triac Output

The **triac** I/O model is used to control the delay of an output pulse signal with respect to an input trigger signal. For control of AC circuits using a **triac** I/O object, the sync input is typically a zero-crossing signal, and the pulse output is the triac trigger signal. The output pulse is 25 μs wide, normally low. The pulse width is independent of the Neuron input clock.

You can use this I/O model to control AC circuits that use a triac device, such as lamp dimmers.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

For applications that drive inductive loads, or applications that operate with a wide range of power line frequencies, see *Stretched Triac Output*.

Hardware Considerations

On a Smart Transceiver, a timer/counter can be configured to control the delay of an output signal with respect to a synchronization input. This synchronization can occur on the rising edge, the falling edge, or both the rising and falling edges of the input signal. For control of AC circuits using a triac device, the sync input is typically a zero-crossing signal, and the pulse output is the triac trigger signal. The resolution and range of the timer/counter period options is described in *Timer/Counter Resolution and Maximum Range* (see **Figure 68**).

The output gate pulse is gated by an internal clock with a constant period of 25.6 µs for a Series 3100 device at 10 MHz (39.062 µs at 6.5536 MHz). Because the input trigger signal (zero crossing) is asynchronous relative to this internal clock, there is a jitter, t_{jit}, associated with the output gate pulse.

The actual active edge of the sync input and the triac gate output can be set by using the **clockedge** or **invert** parameters, respectively.

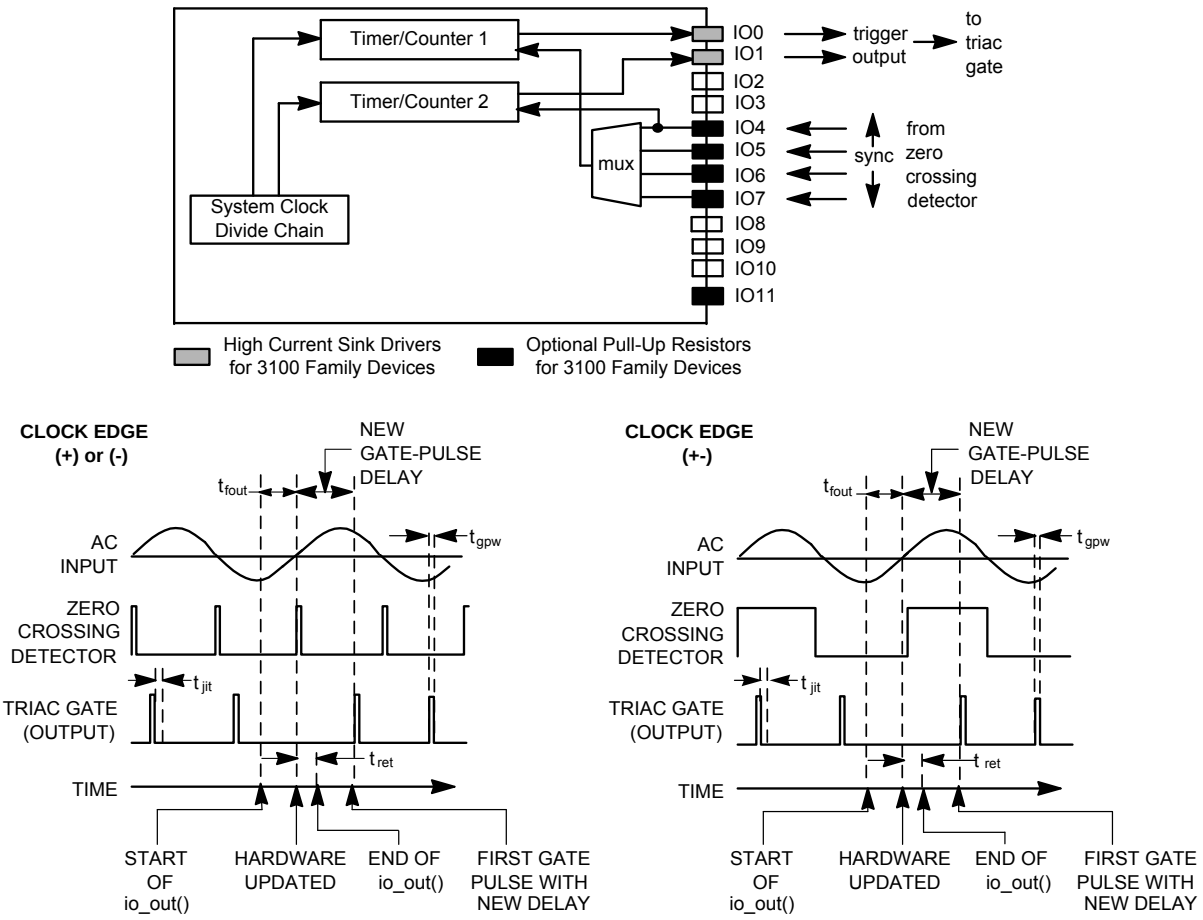


Figure 68. Triac Output and Timing

The hardware update does not happen until the occurrence of an external active sync clock edge. The internal timer is then enabled, and a triac gate pulse is generated after the user-defined period has elapsed. This sequence is repeated indefinitely until another update is made to the triac gate pulse delay value.

t_{fout} (min) refers to the delay from the initiation of the function call to the first sampling of the sync input. In the absence of an active sync clock edge, the input is repeatedly sampled for 10 ms (1/2 wave of a 50 Hz line cycle time), t_{fout} (max), during which the application processor is suspended.

Programming Considerations

Execution of this I/O model is synchronized with the sync pin input and might not return for up to 10 ms. That is, the current application context could be delayed for as long as 10 ms. Because of this synchronization, the frequency of

the sync pin input (and the frequency of the AC circuit being controlled) is limited to the 50-60 Hz range.

When using the pulse output configuration, an output value of 65535 (the overrange value) assures that no output pulse is generated. This is the equivalent of an OFF state. When using the level output configuration, there is always some amount of output signal; use an output value that is about 95% of the half-cycle period to approximate the OFF state.

Syntax

pin [**output**] **triac** [**pulse**] **sync (***pin-nbr***)** [**invert**] [**clock (***const-expr***)**] [**clockedge (+) | (-) | (+-)**] *io-object-name***;**

pin

An I/O pin. Triac output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the sync pin must be **IO_4**.

sync (*pin-nbr***)**

Specifies the sync pin, which is the input trigger signal.

invert

Causes the output signal to be inverted, normally high. The default output signal is normally low.

clock (*const-expr***)**

Specifies a clock in the range 0 to 7, where 0 represents the fastest clock and 7 represents the slowest clock. The default value is clock 0.

You can change resolution for the timer base clock frequency by calling the **io_set_clock( )** function with a clock value in the range 0..7 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

For an application running on a Series 5000 or Series 6000 device, you can specify an increased resolution for the timer base clock frequency by calling the **io_set_clock()** function with a clock value in the range 0..15 (using one of the **TCCLK_*** macros defined in **<echelon.h>**). This function overrides the resolution value specified for **clock()** within the I/O object declaration.

See Appendix A, *Timer/Counter Periods and Resolution*, for a description of the timer resolution and maximum range for each specification of the **clock()** value or each value of the **TCCLK_*** macros. See the *Neuron C Reference Guide* for information about the **io_set_clock()** function.

clockedge (+) | (-) | (+-)

(+) Causes the sync input to be positive-edge sensitive.

(-) Causes the sync input to be negative-edge sensitive. This is the default.

(+-) Causes the sync input to be both positive- and negative-edge sensitive (valid for all Neuron 3120xx Chips, all models of Neuron 3150 Chips except minor model 0, all Smart Transceivers, Series 5000 and Series 6000 devices). Can be used with pulse mode only.

Note: The **clockedge (+-)** option does not work with minor model 0 of Neuron 3150 Chips. When using a Neuron 3150 Chip, a 3150 Smart Transceiver, or a LonBuilder emulator, the compiler inserts code in the application that checks for the availability of this feature. This code logs an error if the chip does not support the feature.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

[**pulse**]

Specifies that the output signal produces a 25 µs pulse at the delay point.

The output pulse is generated by an internal clock with a constant period of 25.6 µs (independent of the Neuron input clock). Because the input sync edge is asynchronous relative to the internal clock, there is a jitter associated with the pulse output relative to the input sync edge. This jitter spans a period of 25.6 µs.

Usage

unsigned long *output-value***;**

io_out(*io-object-name***,** *output-value***);**

Example 1

```
IO_0 output triac sync (IO_5) ioTriac;

when (...) {
  io_out(ioTriac, 325); // delay pulse by 8.3 ms
}

when (...) {
  io_out(ioTriac, 650); // delay pulse by 16.6 ms
}

when (...) {
  io_out(ioTriac, 0); // full on
}
```

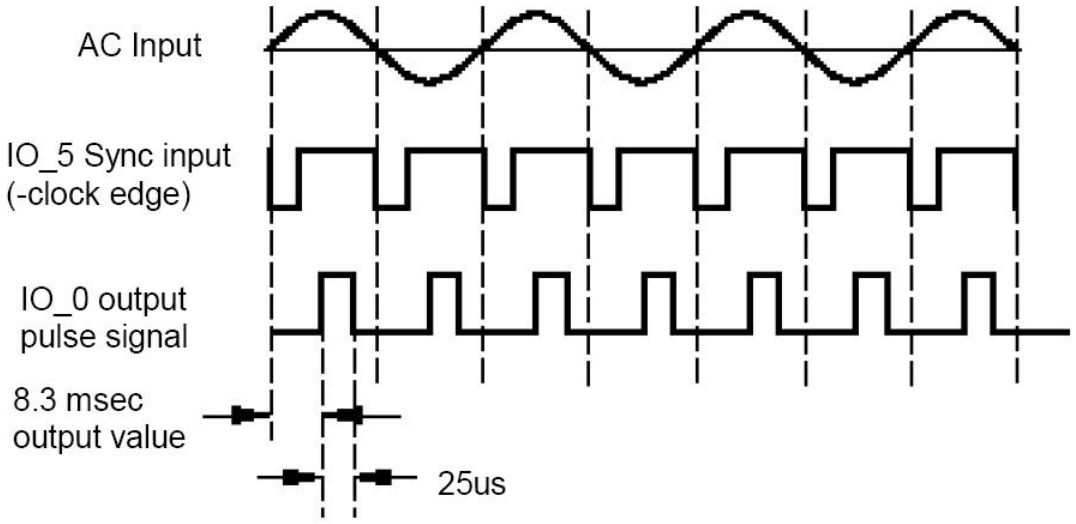


Figure 69. Triac Output, Example 1

Example 2

This example does not apply to model 0 Neuron 3150 Chips.

```
IO_1 output triac sync (IO_4) clockedge (+-) io_dimmer_2;
...
io_out(io_dimmer_2,325);
```

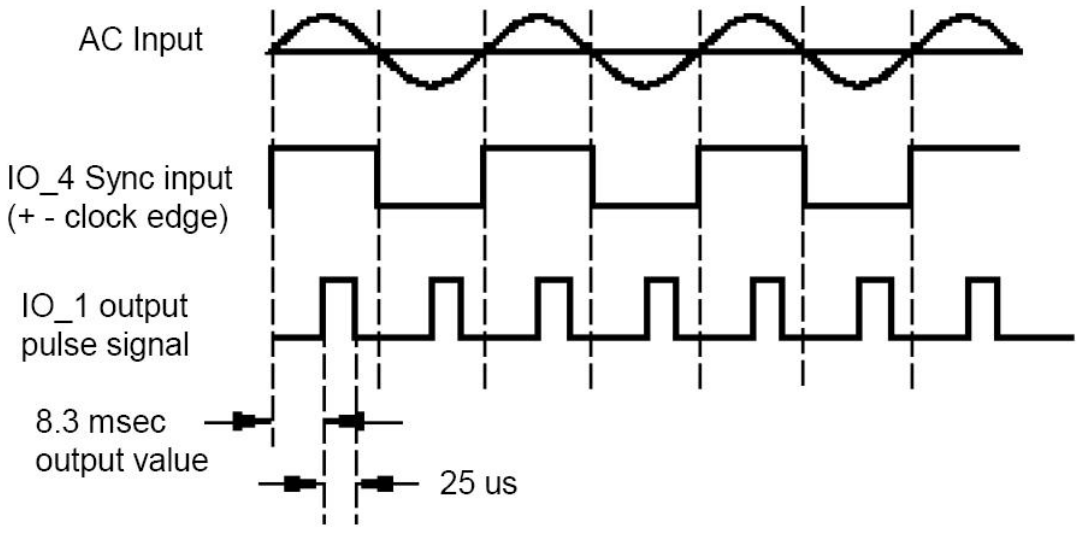


Figure 70. Triac Output, Example 2

Triggered Count Output

You can use this I/O model to control stepper motors or position actuators that provide position feedback in the form of a pulse train.

This model applies to Series 3100 Neuron Chips and Smart Transceivers, to Series 5000 Neuron Processors and Smart Transceivers, and to Series 6000 Neuron Processors and Smart Transceivers.

Hardware Considerations

A timer/counter can be configured to generate an output pulse that is asserted under program control, and de-asserted when a programmable number of input edges (up to 65535) has been counted on an input pin (IO4 – IO7). Assertion can be either logic high or logic low. This object is useful for controlling stepper motors or positioning actuators which provide position feedback in the form of a pulse train. The drive to the external device is enabled until it has moved the required distance, and then the device is disabled (see **Figure 71**).

The active output level depends on whether the **invert** option is used in the declaration of the function block. The default is high.

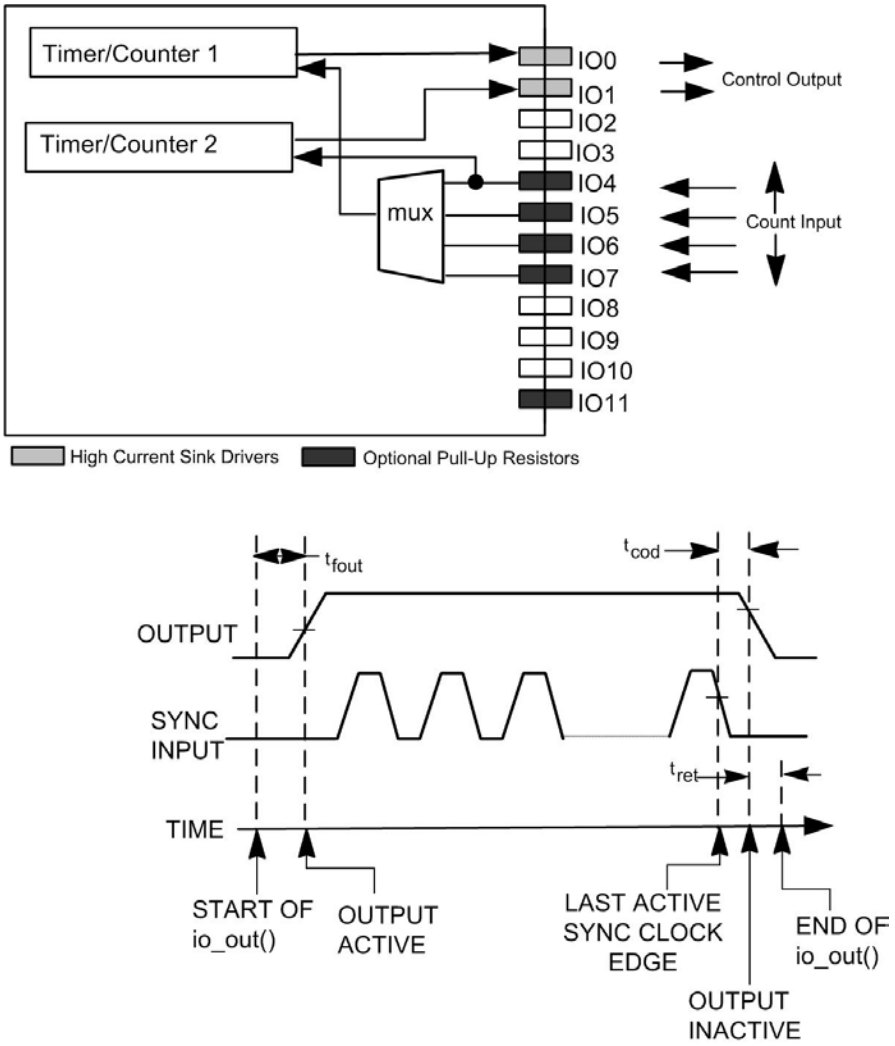


Figure 71. Triggered Count Output and Timing

Table 65. Triggered Count Output Latency Values for Series 3100 Devices

Symbol	Description	Typical at 10 MHz
t_{fout}	Function call to output update	109 µs
t_{cod}	Last negative sync Clock edge to output inactive	min 550 ns max 750 ns
t_{ret}	Return from function	7 µs

Programming Considerations

The **triggeredcount** I/O model is used to control an output pin to the active state and keep it active until *output-value* negative edges are counted at the

input sync pin. After *output-value* edges have counted off, the output pin returns to the low state.

For triggeredcount output, the data type of *output-value* for the **io_out( )** function is an **unsigned long**. An *output-value* of 0 forces the output signal to an inactive state.

Syntax

pin [**output**] **triggeredcount sync (***pin-nbr***)** [**invert**] *io-object-name* [**=***initial-output-level*] **;**

pin

An I/O pin. Triggeredcount output can specify pins **IO_0** or **IO_1**. If **IO_0** is specified, the multiplexed timer/counter is used and the sync pin can be **IO_4** through **IO_7**. If **IO_1** is specified, the dedicated timer/counter is used and the sync pin must be **IO_4**.

sync (*pin-nbr***)**

Specifies the sync pin, which is the counting input signal with low pulses.

invert

Causes the output signal to be inverted, normally high. By default, the output signal is normally low with high pulses.

io-object-name

A user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial-output-level

A constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default initial state is 0.

In **Figure 72**, an **io_out( )** function call is executed with a count argument of 11. After 11 negative edges at the input pin, the output goes low. The delay from the last input edge to the output falling edge is 50 ns or less for a Series 3100 device with an input clock of 40 MHz, or 12.5 ns or less for Series 5000 and Series 6000 devices with an 80 MHz system clock.

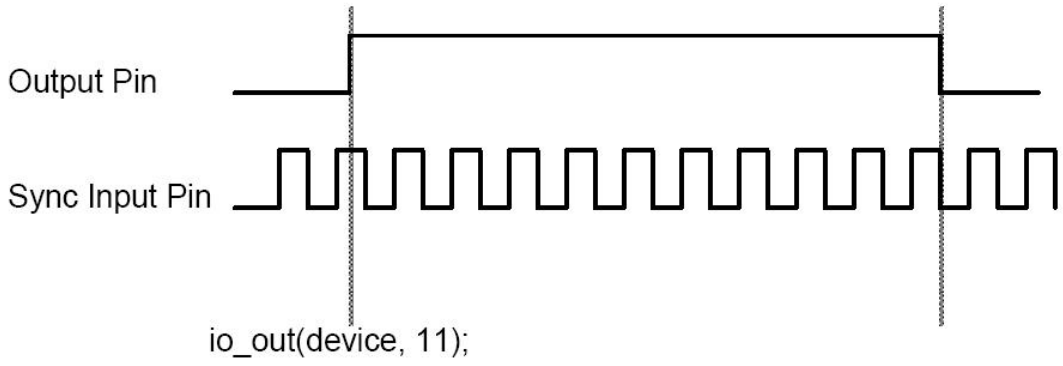


Figure 72. Triggered Count Output

Usage

unsigned long *output-value*;

io_out(*io-object-name*, *output-value***)**;

Example

```
IO_0 output triggeredcount sync (IO_4) ioCascader;

when (...) {
  // 1 big output pulse for 10 input pulses
  io_out(ioCascader, 10);
}
```

A

Timer/Counter Periods and Resolution

This appendix describes resolution, range, rate, frequency, and period information that is common to several timer/counter I/O models.

Timer/Counter Resolution and Maximum Range

Various combinations of I/O pins can be configured as basic inputs or outputs. The application program can optionally specify the initial values of basic outputs. Pins configured as outputs can also be read as inputs, returning the value last written.

The gradient behavior of the timing numbers for different Neuron Chip or Smart Transceiver pins for some of the I/O models is due to the shift-and-mask operation performed by the Neuron system firmware.

Series 3100 Resolution and Range

For dualslope input, edgelog input, ontime input, and period input, and infrared input, the timer/counter returns a value (or a table of values, in the case of edgelog input) in the range 0 to 65535, representing elapsed times from 0 up to the maximum range given in **Table 66**.

For oneshot output, frequency output, and triac output, the timer/counter can be programmed with a number in the range 0 to 65535. This number represents the waveform ontime for oneshot output, the waveform period for frequency output, and the control period from sync input to pulse/level output for the stretched triac and triac output.

Table 66 gives the range and resolution for these timer/counter objects at 10 MHz (the values scale for other clock rates; that is, for a 20 MHz device, divide the values in the table by 2). The clock select value is specified with the **clock** keyword in the declaration of the I/O object in the Neuron C application program, and can be modified at runtime by using the **io_set_clock()** function.

Table 66. Series 3100 Timer/Counter Resolution and Maximum Range at 10 MHz

Clock Select	Dualslope, Edgelog, Ontime, and Period Inputs; Oneshot and Triac Outputs		Frequency Output	
	Resolution	Maximum Range	Resolution	Maximum Range
0	0.2 µs	13.1 ms	0.4 µs	26.2 ms
1	0.4 µs	26.2 ms	0.8 µs	52.4 ms
2	0.8 µs	52.4 ms	1.6 µs	105 ms
3	1.6 µs	105 ms	3.2 µs	210 ms
4	3.2 µs	210 ms	6.4 µs	419 ms
5	6.4 µs	419 ms	12.8 µs	839 ms
6	12.8 µs	839 ms	25.6 µs	1678 ms

7	25.6 µs	1678 ms	51.2 µs	3355 ms

To Calculate for Other Clock Rates:

- Resolution $= \dfrac{2^{(ClockSelect+n)}}{InputClock}$

ClockSelect = 0..7
n = 1 for dualslope, edgelog, oneshot output, ontime, period input, and triac output
n = 2 for frequency output
InputClock in MHz
Resolution in seconds

- Maximum Range = 65535 x *Resolution*

Resolution in seconds
Maximum Range in seconds

For each of the timer/counter I/O models, the range and resolution in **Table 66** should be read as:

- Range of 13.1 ms in steps of 0.2 µs
- Range of 26.2 ms in steps of 0.4 µs
- And so on

Series 5000 and Series 6000 Resolution and Range

For dualslope input, edgelog input, ontime input, and period input, and infrared input, the timer/counter returns a value (or a table of values, in the case of edgelog input) in the range 0 to 65535, representing elapsed times from 0 up to the maximum range given in **Table 67**.

For oneshot output, frequency output, and triac output, the timer/counter can be programmed with a number in the range 0 to 65535. This number represents the waveform ontime for oneshot output, the waveform period for frequency output, and the control period from sync input to pulse/level output for the triac output. The clock select value is specified with the **clock** keyword in the declaration of the I/O object in the Neuron C application program, and can be modified at runtime by using the **io_set_clock()** function.

The resolution and range values in **Table 67** apply to all system clock settings, that is, they do not scale with changes to the system clock.

If you specify an alternate clock value (using the **io_set_clock()** function with a **TCCLK_*** macro value) that is lower than your device's system clock setting, the resolution and range values reflect the expected values specified in **Table 67**.

However, you cannot specify a value that defines a clock rate that is higher than one-half of the device's system clock. For example, if your system clock rate is 20 MHz, you can specify any **TCCLK_*** macro that defines a 10 MHz or lower clock rate (that is, you cannot specify **TCCLK_40MHz** or **TCCLK_20MHz** – no error is issued, but the effective value used in this case is **TCCLK_10MHz**).

Table 67. Series 5000 and Series 6000 Timer/Counter Resolution and Maximum Range

TCCLK Macro	Dualslope, Edgelog, Ontime, and Period Inputs; Oneshot and Triac Outputs		Frequency Output	
	Resolution	Maximum Range	Resolution	Maximum Range
TCCLK_40MHz	25 ns	1.64 ms	50 ns	3.28 ms
TCCLK_20MHz	50 ns	3.28 ms	0.1 µs	6.56 ms
TCCLK_10MHz	0.1 µs	6.56 ms	0.2 µs	13.1 ms
TCCLK_5MHz	0.2 µs	13.1 ms	0.4 µs	26.2 ms
TCCLK_2500kHz	0.4 µs	26.2 ms	0.8 µs	52.4 ms
TCCLK_1250kHz	0.8 µs	52.4 ms	1.6 µs	105 ms
TCCLK_625kHz	1.6 µs	105 ms	3.2 µs	210 ms
TCCLK_312k5Hz	3.2 µs	210 ms	6.4 µs	419 ms
TCCLK_156k2Hz	6.4 µs	419 ms	12.8 µs	839 ms
TCCLK_78k12Hz	12.8 µs	839 ms	25.6 µs	1678 ms
TCCLK_39k06Hz	25.6 µs	1678 ms	51.2 µs	3355 ms
TCCLK_19k53Hz	51.2 µs	3355 ms	102.4 µs	6711 ms
TCCLK_9k77Hz	102.4 µs	6711 ms	204.8 µs	13 422 ms
TCCLK_4k88Hz	204.8 µs	13 422 ms	409.6 µs	26 843 ms
TCCLK_2k44Hz	409.6 µs	26 843 ms	819.2 µs	53 686 ms
TCCLK_1k22Hz	819.2 µs	53 686 ms	1638.4 µs	107 373 ms

For each of the timer/counter I/O models, the range and resolution in **Table 67** should be read as:

- Range of 13.1 ms in steps of 0.2 µs
- Range of 26.2 ms in steps of 0.4 µs
- And so on

Timer/Counter Square Wave Output

The following sections list the possible choices for pulsetrain repetition frequencies for pulsewidth short output and pulsecount output.

Series 3100 Square Wave Output

For pulsewidth short output and pulsecount output, **Table 68** lists the possible choices for pulsetrain repetition frequencies for a Series 3100 device. Pulsecount cannot be used with clock select 0. The table lists the values for a 10 MHz input clock (the values scale for other clock rates).

Table 68. Series 3100 Timer/Counter Square Wave Output at 10 MHz

Clock Select	System Clock Divider	Repetition Rate	Repetition Period or Pulse Period	Resolution of Pulse
0	÷ 1 (5 MHz)	19531 Hz	51.2 µs	0.2 µs
1	÷ 2 (2.5 MHz)	9766 Hz	102.4 µs	0.4 µs
2	÷ 4 (1.25 MHz)	4883 Hz	204.8 µs	0.8 µs
3	÷ 8 (625 kHz)	2441 Hz	409.6 µs	1.6 µs
4	÷ 16 (312.5 kHz)	1221 Hz	819.2 µs	3.2 µs
5	÷ 32 (156.25 kHz)	610 Hz	1638.4 µs	6.4 µs
6	÷ 64 (78.125 kHz)	305 Hz	3276.8 µs	12.8 µs
7	÷ 128 (39.06 kHz)	153 Hz	6553.6 µs	25.6 µs

To Calculate for Other Clock Rates:

- Period $= 512 \times \left(\frac{2^{(ClockSelect)}}{InputClock} \right)$

  ClockSelect = 0..7
  InputClock in MHz
  Period in seconds

- Frequency = 1 / *Period*
  Frequency in Hertz
  Period in seconds

Series 5000 and Series 6000 Square Wave Output

For pulsewidth short output and pulsecount output, **Table 69** lists the possible choices for pulsetrain repetition frequencies for a Series 5000 device.

The resolution and range values in **Table 69** apply to all system clock settings, that is, they do not scale with changes to the system clock.

If you specify an alternate clock value (using the **io_set_clock()** function with a **TCCLK_*** macro value) that is lower than your device's system clock setting, the resolution and range values reflect the expected values specified in **Table 69**.

However, you cannot specify a value that defines a clock rate that is higher than one-half of the device's system clock. For example, if your system clock rate is 20 MHz, you can specify any **TCCLK_*** macro that defines a 10 MHz or lower clock rate (that is, you cannot specify **TCCLK_40MHz** or **TCCLK_20MHz** – no error is issued, but the effective value used in this case is **TCCLK_10MHz**).

Table 69. Series 5000 and Series 6000 Timer/Counter Square Wave Output

TCCLK Macro	Repetition Rate	Repetition Period or Pulse Period	Resolution of Pulse
TCCLK_40MHz	156.2 kHz	6.4 µs	25 ns
TCCLK_20MHz	78.1 kHz	12.8 µs	50 ns
TCCLK_10MHz	39.1 kHz	25.6 µs	0.1 µs
TCCLK_5MHz	19531 Hz	51.2 µs	0.2 µs
TCCLK_2500kHz	9766 Hz	102.4 µs	0.4 µs
TCCLK_1250kHz	4883 Hz	204.8 µs	0.8 µs
TCCLK_625kHz	2441 Hz	409.6 µs	1.6 µs
TCCLK_312k5Hz	1221 Hz	819.2 µs	3.2 µs
TCCLK_156k2Hz	610 Hz	1638.4 µs	6.4 µs
TCCLK_78k12Hz	305 Hz	3276.8 µs	12.8 µs
TCCLK_39k06Hz	153 Hz	6553.6 µs	25.6 µs
TCCLK_19k53Hz	76.3 Hz	13.11 ms	51.2 µs
TCCLK_9k77Hz	38.1 Hz	26.21 ms	102.4 µs
TCCLK_4k88Hz	19.1 Hz	52.43 ms	204.8 µs
TCCLK_2k44Hz	9.5 Hz	104.86 ms	409.6 µs
TCCLK_1k22Hz	4.77 Hz	209.72 ms	819.2 µs

Timer/Counter Pulsetrain Output

The following sections list the possible choices for pulsetrain repetition frequencies for pulsewidth short output and pulsecount output.

Series 3100 Pulsetrain Output

For pulsewidth long output, **Table 70** lists the possible choices for pulsetrain repetition frequencies. The table lists the values for a Series 3100 device with a 10 MHz input clock (the values scale for other clock rates).

Table 70. Series 3100 Timer/Counter Pulsetrain Output at 10 MHz

Clock Select	Frequency	Period
0	76.3 Hz	13.1 ms
1	38.1 Hz	26.2 ms
2	19.1 Hz	52.4 ms
3	9.54 Hz	105 ms
4	4.77 Hz	210 ms
5	2.38 Hz	419 ms
6	1.19 Hz	839 ms
7	0.60 Hz	1678 ms

To Calculate for Other Clock Rates:

- Period $= 131072 \times \left(\frac{2^{(ClockSelect)}}{InputClock} \right)$

  ClockSelect = 0..7
  InputClock in MHz
  Period in seconds

- Frequency = 1 / *Period*
  Frequency in Hertz
  Period in seconds

Series 5000 and Series 6000 Pulsetrain Output

For pulsewidth long output, **Table 71** lists the possible choices for pulsetrain repetition frequencies.

The resolution and range values in **Table 71** apply to all system clock settings, that is, they do not scale with changes to the system clock.

If you specify an alternate clock value (using the **io_set_clock()** function with a **TCCLK_*** macro value) that is lower than your device's system clock setting, the resolution and range values reflect the expected values specified in **Table 71**.

However, you cannot specify a value that defines a clock rate that is higher than one-half of the device's system clock. For example, if your system clock rate is 20 MHz, you can specify any **TCCLK_*** macro that defines a 10 MHz or lower clock rate (that is, you cannot specify **TCCLK_40MHz** or **TCCLK_20MHz** – no error is issued, but the effective value used in this case is **TCCLK_10MHz**).

Table 71. Series 5000 Timer/Counter Pulsetrain Output

TCCLK Macro	Frequency	Period
TCCLK_40MHz	610.4 Hz	1.64 ms
TCCLK_20MHz	305.2 Hz	3.28 ms
TCCLK_10MHz	152.6 Hz	6.56 ms
TCCLK_5MHz	76.3 Hz	13.1 ms
TCCLK_2500kHz	38.1 Hz	26.2 ms
TCCLK_1250kHz	19.1 Hz	52.4 ms
TCCLK_625kHz	9.54 Hz	105 ms
TCCLK_312k5Hz	4.77 Hz	210 ms
TCCLK_156k2Hz	2.38 Hz	419 ms
TCCLK_78k12Hz	1.19 Hz	839 ms
TCCLK_39k06Hz	0.60 Hz	1678 ms
TCCLK_19k53Hz	0.30 Hz	3.4 sec
TCCLK_9k77Hz	0.15 Hz	6.7 sec
TCCLK_4k88Hz	0.075 Hz	13.4 sec
TCCLK_2k44Hz	0.037 Hz	26.8 sec
TCCLK_1k22Hz	0.019 Hz	53.7 sec

Index

I

U

V

W