Parallel I/O Interface to the Neuron® Chip

January 1995 **LONWORKS® Engineering Bulletin**

The Neuron Chip parallel I/O object permits bidirectional data transfer at rates of up to 3.3Mbps. A Neuron Chip may communicate with another Neuron Chip or with any other microprocessor or microcontroller. The Microprocessor Interface Program (MIP), running on the Neuron Chip, provides an easy solution for connecting the Neuron Chip to a host processor. The MIP is available in several configurations supporting different interface schemes including interrupt, dual ported RAM, and polled.

The physical interface to the parallel I/O object is accomplished through the eleven I/O pins of the Neuron Chip. No other I/O objects of the Neuron Chip may be used in conjunction with parallel I/O. In addition to the physical interface, a token-passing, handshaking protocol is implemented by the Neuron Chip firmware as a way to establish synchronization and prevent bus contention.

The Neuron C programming language provides several built-in functions that enable the use of the parallel I/O object without the need for detailed, hardware-level knowledge of the handshaking protocol. These functions are discussed in detail in the Neuron Chip-to-Neuron Chip interface section of this document.

For increased design flexibility, the Neuron Chip provides several modes of operation for the parallel I/O object: Master, Slave A, and Slave B. The different attributes of each mode can be used to tailor the Neuron Chip for a specific application.

The master mode is the intelligent mode of the parallel I/O object. In this mode, the Neuron Chip controls the handshaking protocol between itself and the attached processor, which is in the slave mode. While in the master mode, the Neuron Chip may be interfaced to another Neuron Chip (in slave A mode), a microprocessor, or a microcontroller.

In the slave A mode, the Neuron Chip is under control of a master. The Neuron Chip appears to the host as a parallel I/O device with 8 data bits and 3 control bits.

The slave B mode is logically similar in operation to the slave A mode; however, the handshaking process and the data bus control are specifically tailored for use in a microprocessor bus environment. The Neuron Chip appears to the host as a memory mapped I/O device. This is useful when interfacing a Neuron Chip to a microprocessor or microcontroller, or when there is a need for multiple slaves on the same parallel bus (e.g., PC bus interfacing).

Figure 1 illustrates the application of the different parallel I/O modes. Although all possible interfacing scenarios are shown, not all can be considered for every application. Certain applications, such as a Neuron Chip-to-Neuron Chip connection, have only one solution (master to slave A), while interfacing a foreign

processor to the Neuron Chip can be accomplished in several ways depending on available hardware and software resources.

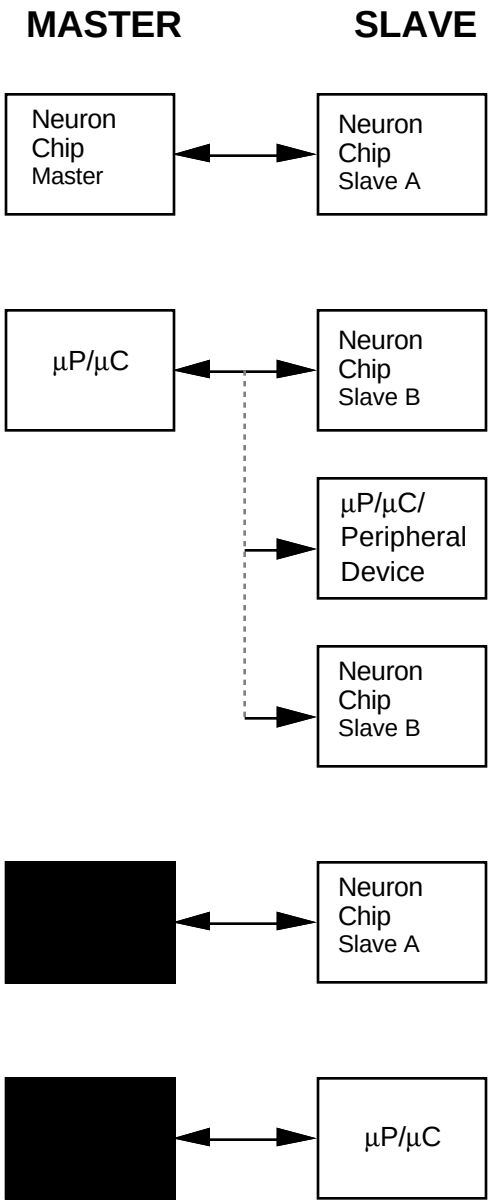

Figure 1 Possible master/slave connections for the NEURON CHIP

In a non-Neuron Chip (foreign processor) interface, it is assumed that the microprocessor or microcontroller involved has the ability to execute the token passing algorithm dictated by the attached Neuron Chip. This usually consists of a hardware interface and a software program that duplicates the actions of a Neuron Chip .

Neuron Chip Interface

The Neuron Chip parallel I/O interface consists of eight I/O and three control lines. Figure 2 shows the assignment of the Neuron Chip pins for each of the parallel I/O modes.

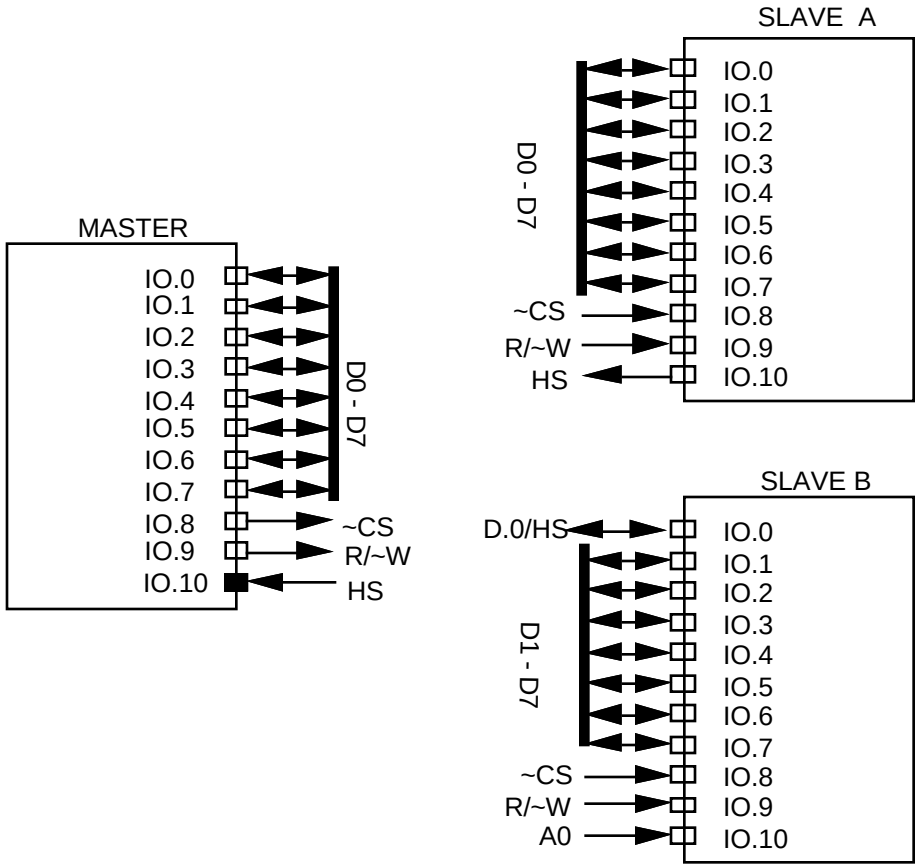

Figure 2 Pin assignments for the three modes of parallel I/O

The ~CS line is always driven by the master and, when active, signifies that a byte transfer operation is currently in progress. A low pulse on this line strobes the data into either the master or slave. The ~CS line is asynchronous and should be kept as noise-free as possible.

The type of data transfer actually taking place, either a read or a write (with respect to the master), is assessed by the level of the R/~W line at the time the ~CS line is pulsed low. The R/~W line is driven by the master, and determines the direction of the bi-directional bus drivers on a Neuron Chip slave.

The HS (handshake) line is always driven by the slave. It informs the master that the slave is busy. In effect, the HS line can be treated as a slave-busy signal. When high, it is the slave's turn to perform an action (read or write command and data);

otherwise, it is the master's turn to access the bus. Note that the state of ~HS could change before ~CS has become inactive.

The A0 pin, driven by the master and only available on the slave B mode, is the address pin that selects between the data register or the control register containing the HS bit. The HS bit is the least significant bit of the control register (D0 line). The remaining bits of the control register are unused. The explicit HS polling required by the master is what separates the microprocessor bus-compatible slave B mode from the slave A mode.

It is possible for the master device to come online and poll the HS line before the Neuron Chip Slave has had a chance to set the proper level on this line. To prevent the master from reading invalid data on the HS line, it is recommended that this line be pulled high through a 10k Ω pull-up resistor for a slave A Neuron Chip. For a slave B Neuron Chip, the D0 line should be pulled high.

Handshake Protocol

The handshake protocol implemented by the Neuron Chip firmware permits coexistence of multiple devices on a common bus. At any given time, only one device is given the option of writing to the bus. A virtual write token is passed alternately between the master and the slave on the bus in an infinite, ping-pong fashion (there is one exception to this when the Neuron Chip is interfaced to a host processor which will be discussed later). The owner of the token has the option of writing data, or alternatively, passing the token without any data.

Figure 3 illustrates the token passing operation between a master and a slave.

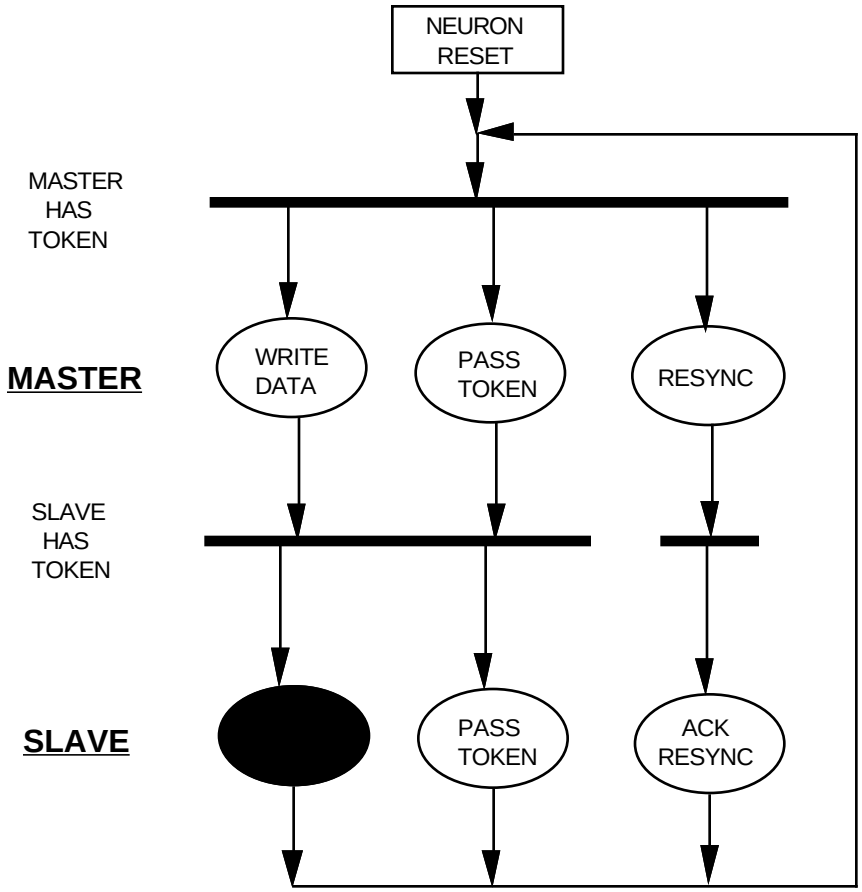

Figure 3 Handshake protocol sequence between master and slave

Multiple slaves (slave B) on a common bus, with multiple write tokens, can also be supported by the token-passing protocol. In such a case, the master must keep track of all outstanding write tokens and accordingly direct bus traffic. This is a special application of the parallel I/O object and will not be addressed in this document.

Once in possession of the write token, a device may perform one of several operations (as shown in figure 3): write data, pass token, resynchronize (master only), or acknowledge resynchronization (slave only).

The sequence of events for each of the above operations is the same every time, for either the master or the slave (A or B). However, the degree to which the user is exposed to the underlying token-passing operations is varied depending on the actual device involved. Built-in tools within the Neuron C language allow for straightforward software coding of the Neuron Chip. This translates to a transparent token-passing protocol, which in turn results in program simplicity and a lower probability of communication errors.

On the other hand, if a Neuron Chip is interfaced to a non-Neuron processor (host processor), the responsibility of token passing falls in the hands of the host processor. Although the software program on the Neuron Chip side is still simple,

the user must now explicitly implement the token-passing protocol on the host processor side. The following section describes the token-passing protocol in detail.

Protocol Commands

The byte format of the command options available to the token holder are further described in figure 4.

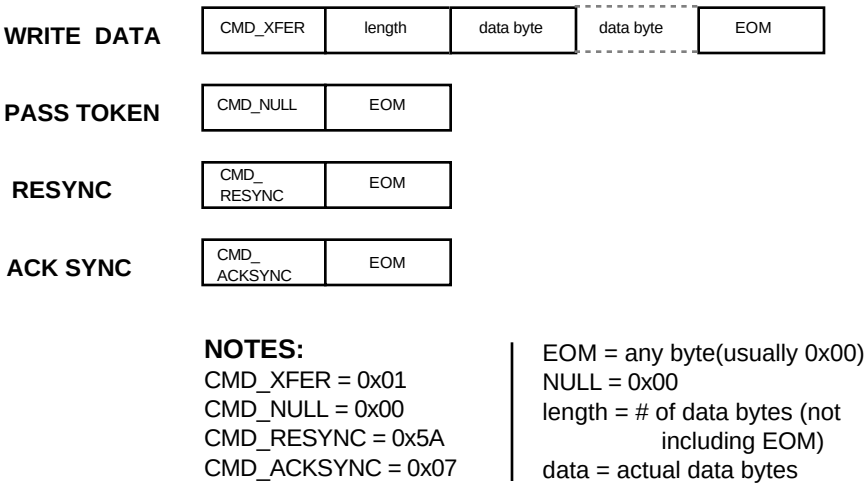

Figure 4 Possible alternatives available to the token holder

These commands are the building blocks on which all communication between a Neuron Chip parallel I/O and the outside world are based. Only one of the above commands can be performed by the token holder at any given time. Upon completion of the command, the token is passed to the other device. The attached device now has the opportunity to execute a command. The write token is thus passed back and forth between the master and slave indefinitely.

Each command is made up of a fixed sequence of read and write operations to the bus by both the master and the slave. These operations define the actual handshaking process required by each command. The state transition diagram for each command is shown in figure 5.

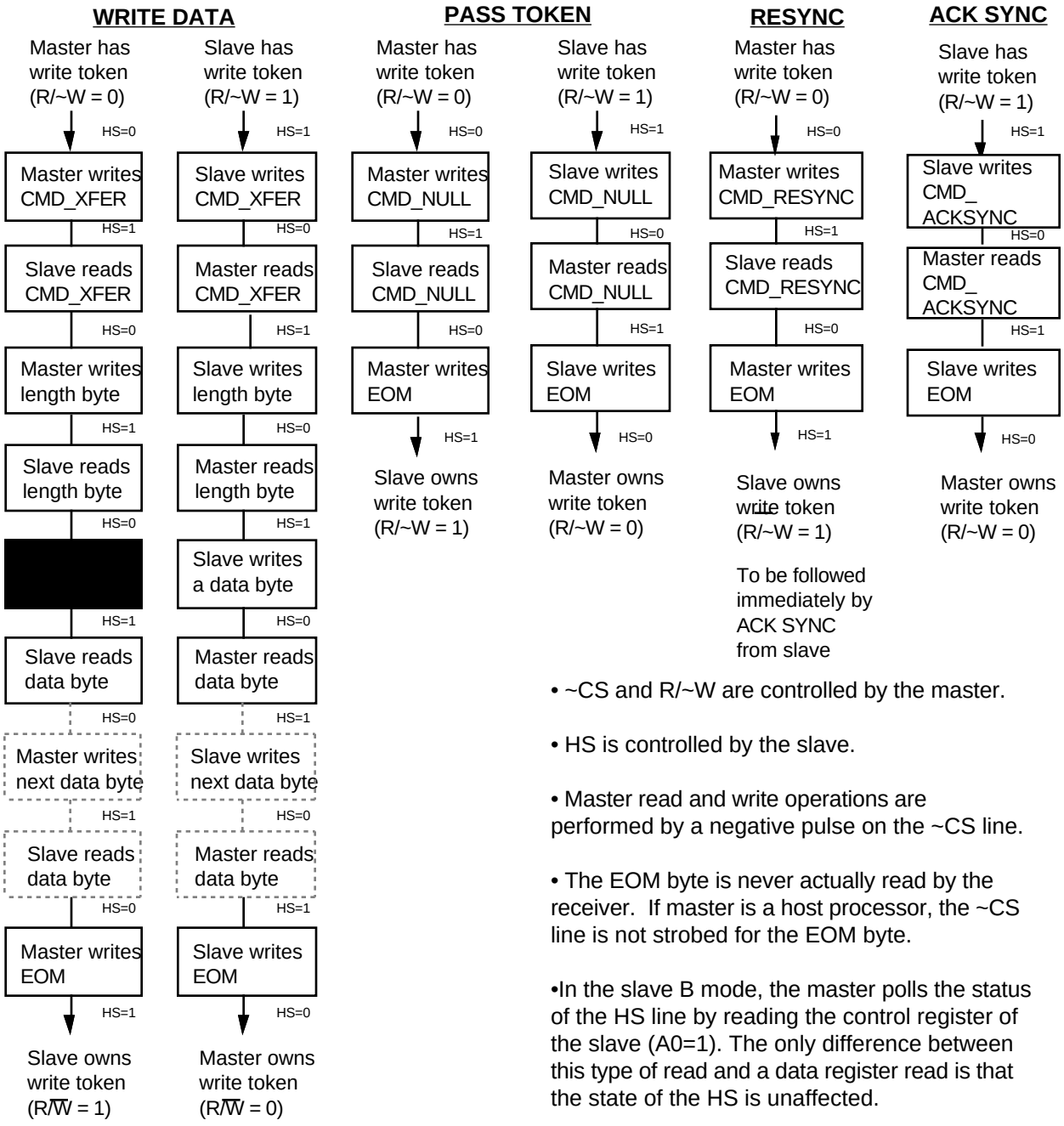

Figure 5 Micro-operations of the handshake protocol

Master read and write operations are performed by a negative pulse (high to low to high) on the ~CS line. For the read operation, this causes the slave to put the data on the bus so that it may be strobed in by the master. In the case of the write operation, the data on the bus is strobed into the slave's input buffer. For both read and write, the actual data is latched (strobed) on the rising edge of ~CS. The low-to-high transition of the ~CS causes the HS line to go high (the only exception to this is

when the master reads the control register in the slave B mode. HS is unaffected in this case).

The EOM byte always terminates a command and is never actually read by the device to which it is sent. The EOM byte is used by the slave to toggle the state of the HS line at the end of a command in order to pass the write token.

The writing of the CMD_NULL and EOM is done automatically by the Neuron Chip when it posseses the write token and the Neuron C application program executes a `when(io_in_ready())` or a `when (io_out_ready())`. If the Neuron Chip has an `io_out_request()` pending, it will not automatically pass the write token back to the host.

When the Neuron Chip is interfaced to a host processor, the host may hold the token indefinitely without holding up the Neuron Chip. However, if the Neuron Chip holds the token, the host must obtain the token within the watchdog timer period of the Neuron Chip in order to prevent a Neuron Chip reset.

The HS line is the main handshaking control signal used to control actual data transfers. The action of a master reading from or writing to the bus sets the HS line high. This is a hardware controlled, not a firmware controlled, action. When the slave performs a read or a write, the HS line is set low again. When a host processor is the master, the HS line must be explicitly polled by that processor's software routine to properly initiate the read and write operations (controlled by the ~CS and R/~W lines).

Synchronization

Upon a Neuron Chip reset, the write token is, by definition, in the possession of the master. Synchronization across the parallel bus is required by the Neuron Chip following any reset condition. The purpose of this step is to prevent a state from occurring where the Neuron Chip assumes that the device attached to it is in a given state when it may not be. The results of this misunderstanding could be false starts of data transfers, or incorrect data transfers. This is automatically accomplished by the Neuron Chip through the use of a synchronization sequence.

The Neuron Chip's automatic synchronization process occurs just before the reset clause of the application program is executed, and just after configuration of the Neuron Chip's I/O pins. Prior to this step the Neuron Chip's I/O pins are configured as inputs, which is always the case immediately following Neuron Chip reset.

The automatic synchronization sequence carried out by the Neuron Chip is dependent on the mode of its parallel I/O object. If the Neuron Chip is a master, then following a reset, it will initiate a resynchronization command. If the Neuron Chip is a slave (A or B) then it will await the arrival of a resynchronization command from the master (any other command will be ignored).

The parallel I/O object provides a way to explicitly synchronize the devices when a host processor is the master. This enables the foreign processor to ascertain the integrity of the communication medium, and re-establish a predetermined state, at any time. Aside from the initial synchronization necessary after a reset, the host processor is not required to perform this operation at any other time. The capability, however, is provided for the system designer in case a need does arise.

The resynchronization operation can be initiated by the token-holding master at any time by the use of the RESYNC command. The RESYNC command sends a special message (CMD_RESYNC) to the slave which in turn triggers it to send its own special message (CMD_ACKSYNC) back to the master. Thus, a two-way communication has taken place and the token has been passed from the master to the slave and back to the master again.

It is recommended that any device (master or slave) in a system be aware of the other device's reset so that synchronization may be reestablished. For example, a host processor should monitor the reset output of an attached Neuron Chip with a status latch or reset. In addition, a reset on the host processor could cause the Neuron Chip to reset.

Most of the operations described by the above state diagrams, in addition to the synchronization operations, are transparent to the Neuron Chip application programmer. They are automatically executed by the Neuron Chip's firmware. When interfacing a host processor to the Neuron Chip, however, the above-mentioned operations must be explicitly carried out by the attached processor.

The Neuron C programming language allows access to the parallel I/O object. The following section describes the available resources within the Neuron C programming language.

Neuron C Resources

The parallel I/O object is declared in a Neuron C program using the following syntax (see the *Neuron C Programmer's Guide* for details):

```
IO_0 parallel slave|slave_b|master io_object_name;
```

In order to use the parallel I/O object of the Neuron Chip, `io_in()` and `io_out()` require a pointer to the `parallel_io_interface` structure defined below.

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

The above structure must be declared, with an appropriate definition of `maxlength` signifying the largest expected buffer size for any data transfer.

In the case of `io_out()`, `length` is the number of bytes to be transferred out and is set by the user program. In the case of `io_in()`, `length` is the number of bytes to be transferred in. If the number of bytes received is less than or equal to `length` , then

`length` is replaced with the new number. Otherwise, `length` is set to zero. The length field must be set before calling `io_in()` or `io_out()`. The maximum value for the `length` and `maxlength` fields is 255.

The parallel I/O object of the Neuron Chip is easily accessed with the use of built-in Neuron C functions and events. The following functions and events are provided specifically for use with the parallel I/O object:

`io_out_request()` This function is used to request an `io_out_ready` indication for an I/O object. Calling this function sets a flag in the Neuron Chip which prevents it from giving up the write token at the end of an `io_in_ready` or `io_out_ready` evaluation (see below). It is up to the application to buffer the data until the `io_out_ready` event is TRUE.

`io_in_ready` This event becomes TRUE whenever a message arrives on the parallel bus that must be read. The application must then call `io_in()` to retrieve the data. The exact algorithm used for evaluation of this event is shown in figure 6.

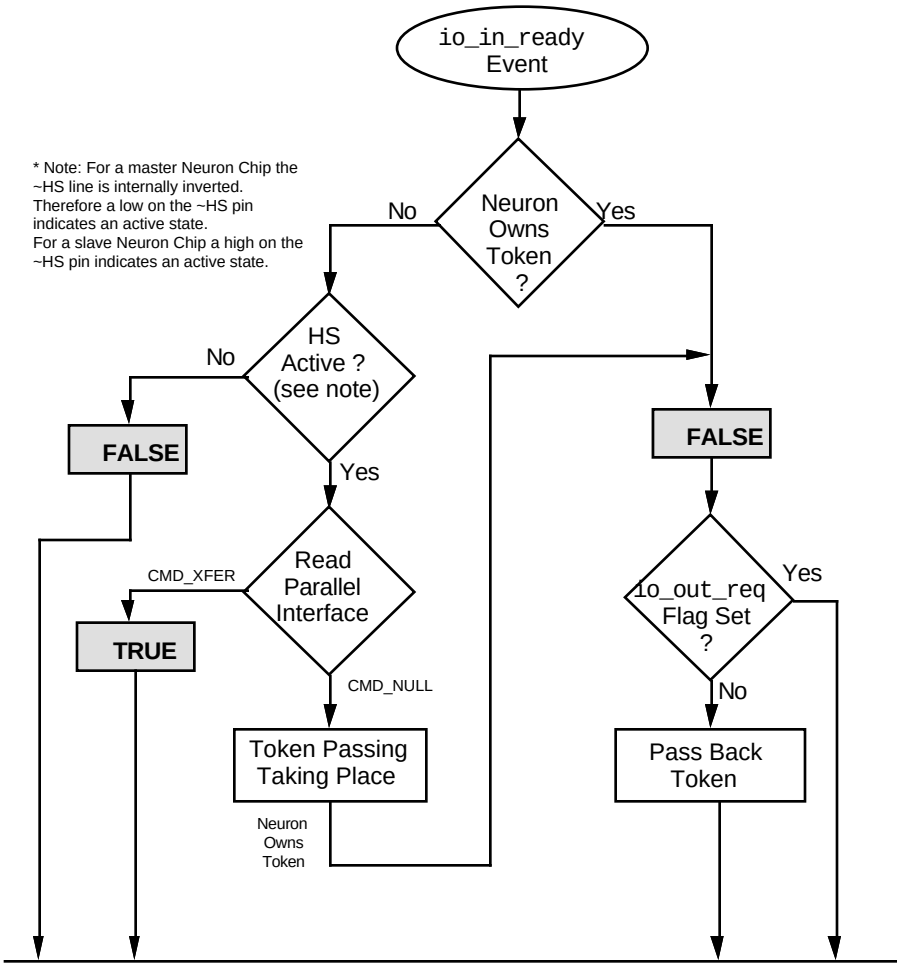

Figure 6 Neuron Chip algorithm for evaluating the `io_in_ready()` event

`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 invoked. The application must then call the `io_out()` function to write the data to the parallel port.
`io_in()`	The CMD_XFER byte has already been read. This function reads the length byte and the rest of the transfer data. Once the length byte is read, the Neuron Chip executes a block read instruction.
`io_out()`	The CMD_XFER byte is written to the interface. Following this the Neuron Chip executes a block write instruction which writes the length byte followed by the rest of the transfer data. The Neuron Chip then executes the EOM procedure, which varies depending on whether it is configured as a master or a slave. This function also clears the `io_out_request` flag.

Neuron C applications may be written that use the parallel bus in a uni-directional manner (i.e., applications may be written without either an `io_in_ready` or `io_out_ready` when clause). In the case where no `io_in()` function is called, it is up to the programmer to assure that no read transfers of real data messages will ever be required by the application. This is to protect the device on the other side of the bus from waiting forever on a data transfer.

Refer to the *Neuron Chip-to-Neuron Chip Interface* section of this document for an example which uses the above Neuron C functions and events.

Timing

Timing data for the parallel I/O interface can be categorized into three distinct classes: Firmware timing, clocked timing, and hardware AC characteristics. Below is a listing of timing numbers for the firmware portion of the interface for a Neuron Chip running at 10MHz. Refer to the *Neuron C Resources* section for a detailed explanation of these parameters.

io_out_request()	Function call to return	13.2µs
io_in_ready()	Multiple code paths are possible (see figure 6):	
	1) Node owns token and `io_out_request` is posted	33.0µs
	2) Node does not own token and HS is not active	45.6µs
	3) Node does not own token, node reads CMD_NULL, no `io_out_request` is posted (node passes token)	1.57ms

	4) Node does not own, node reads CMD_XFER	68.4µs
io_out_ready()	In addition to the execution time of the io_in_ready() function	10.2 to 24.6µs
io_in()	-Function call to read length byte	57.0µs
	-Read length byte to block read instruction	36.0µs
	-End of block read to function return	75.6µs
io_out()	-Function call to write CMD_XFER	51.6µs
	-Write CMD_XFER to start of block write instruction	21.6µs
	-End of block write to write EOM	40.8µs
	-Write EOM to function return	24.6µs

For a detailed timing specification of the hardware AC timing portion of the parallel I/O interface refer to the *Neuron Chip Data Book*.

The maximum data transfer rate for the parallel I/O object running in the slave A mode (Neuron Chip-to-Neuron Chip) is one byte per 2.4µs, or 3.3Mbps, for Neuron Chips operating at 10MHz. Note that this rate applies to the data portion of the transfer only. The overhead associated with the processing of the command and the length bytes must also be taken into account when calculating the average data rate.

The overhead associated with reading the handshake status by the attached host microprocessor will affect the average data rate. This will be a function of both the speed of the host processor and also the method used to detect handshake transitions (interrupt or poll).

Neuron Chip-to-Neuron Chip Interface

The parallel connection of one Neuron Chip to another is accomplished by assigning one as the master device and the other as a slave A device. The hardware requirements in this case reduce to a direct, one-to-one, connection of all eleven I/O pins on both sides.

The following program illustrates a typical parallel I/O processing interface routine which would reside on both the master and the slave A Neuron Chips .

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

```
} piofc;

when (io_in_ready(s_bus))           //ready to input data
{
   piofc.length = DATA_SIZE;        //number of bytes to read
   io_in(s_bus, &piofc);            //get 10 bytes of incoming data
}
when (io_out_ready(s_bus))          //ready to output data
{
   piofc.length = 10;               //number of bytes to write
   io_out(s_bus, &piofc);           //output 10 bytes from buffer
}
when(...)                           //user defined event
{
   io_out_request(s_bus);        //post the write transfer request
}
```

Neuron-to-Host Processor Interface (Slave B mode)

This example illustrates the use of the slave B mode of the parallel I/O by interfacing the Neuron Chip to the Motorola 68HC11 microcontroller. The 68HC11 is the master and the Neuron Chip is the slave residing in the 68HC11's address space.

No interface circuitry is needed aside from some address decoding logic that would allow the 68HC11 to access the Neuron Chip by using specific addresses (one address for the data register and one for the control register). A typical design for this address decoding logic is shown in figure 10.

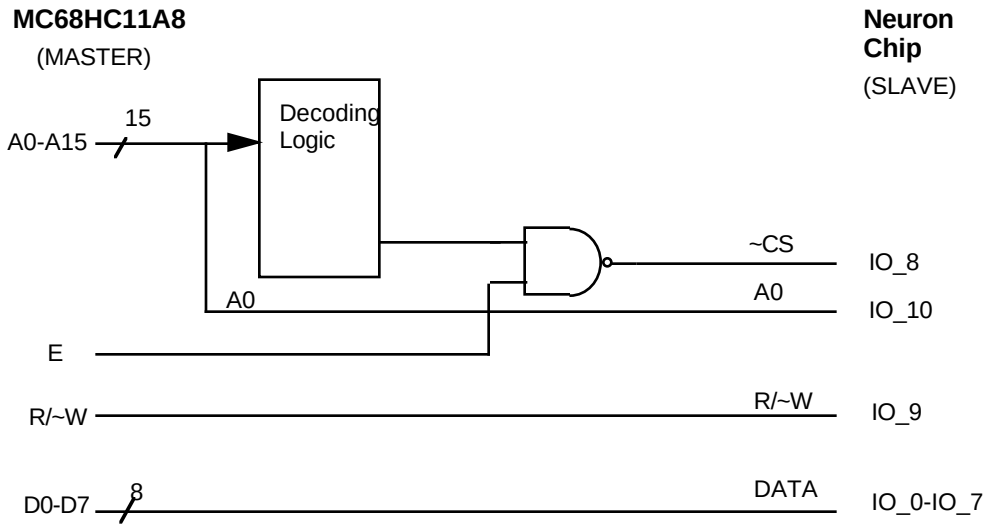

Figure 10 Address decoding logic for interfacing the Neuron Chip to the 68HC11

The following is the assembly listing of a program that runs on the 68HC11. The corresponding code for the Neuron Chip is identical to that shown in the Neuron Chip-to-Neuron Chip example. Due to the transparent nature of the communication protocol at the Neuron C programming level, the Neuron Chip programmer need not be aware that the interface is to a 68HC11 (or any other host processor for that matter) instead of to another Neuron Chip .

```
************************************************************
** Test program for master/SlaveB mode where
** master is resident on 68HC11 and slave
** is resident on the Neuron Chip.
**
** The code below implements the 68HC11 portion,
** receiving any data and sending pre-defined data
** messages. This code is implemented more as a test
** of the interface rather than a test of the protocol.
************************************************************
NEURON_ADDR equ   $df00
DEBUG_ADDR  equ   $0030
HS_MASK     equ   $01
MAXMSGLEN   equ   $20
EOM         equ   $0
TRUE        equ   $1
FALSE       equ   $0
CMD_RESYNC  equ   $5A
CMD_ACKSYNC equ   $07

** The NEURON CHIP is sitting on the HC11's data bus with a chip
** select address decoder set to the following addresses.

data        equ   $df00
control     equ   $df01

   ORG      $0000

   XDEF     token       * boolean representing which side has the
token       RMB   1        * token

   XDEF     counter     * general purpose counter
counter     RMB   1

   XDEF     msgi        * message in structure
msgi        RMB   34
```

```
mi_command  equ   0  * location of command in the msg structure
mi_length   equ   1  * location of data length in   "  "
mi_data     equ   2  * location of start of data in "  "
   *
   * Program Section
   *
   ORG   $E000
************************************************************
** start of parallel master code
************************************************************
   XDEF  start_pio
start_pio
   JSR   master_init    * initialize

   XDEF  main_loop
main_loop
   LDAB  token          * load token
   BEQ   no_token       * if token==0, can't write
   *
   JSR   pio_write      * send code message
no_token
   *
   * This test program receives any messages
   JSR   pio_read       *try to read
   BRA   main_loop      *repeat

************************************************************
** wait_hs
**    When the NEURON CPU reads or writes the data port,
**    it drives the HS line low. The master must wait for
**    HS low before reading from or writing to the port.
************************************************************
   XDEF  wait_hs
wait_hs:
   LDAB  control
   ANDB  #HS_MASK
   BNE   wait_hs
```

```
   RTS

************************************************************
** master_init
** Proceed with the standard synchronization with the
** NEURON. Write the CMD_RESYNC value plus EOM. Wait
** for the CMD_ACKSYNC value. Return owning token.
************************************************************
   XDEF  master_init
master_init
   JSR   wait_hs        * wait for H.S.
   LDAB  #CMD_RESYNC    *
   STAB  data           * send the resync value
   JSR   write_eom      * and the EOM.

   JSR   wait_hs        * wait for the CMD_ACKSYNC.
   LDAB  data           * read data from the port
   CMPB  #CMD_ACKSYNC
   BEQ   read_complete  * repeat if not sync'ed
   BRA   master_init

************************************************************
** pio_read
************************************************************
   XDEF  pio_read
pio_read
   LDAB  control        *load control
   ANDB  #HS_MASK
   BEQ   da
   RTS                  *no data available
   *
   * We have data available, handshake line is low
da
   LDY   #msgi          * set up Y index
   LDAB  data           * read data from the port
   STAB  0,Y            * store in message.command
   INY
```

```
   TST B                * check the data
```

```
   BNE   have_data      * go get data, if command!=NULL
   *
   * This was token passing message (NULL)
   CLR   0,Y            * msgi.length=0
   BRA   read_complete
   *
   * Since the command was non-zero, get the length byte next.
have_data
   JSR   wait_hs        * wait for indication of data
   LDAB  data           * read data from port
   STAB  0,y            * msgi.length=ACCB
   INY
   STAB  counter        * set up the counter

loop_data
   LDAB  counter        *load the counter, Z=1, if counter==0
   BEQ   read_complete  *if counter==0, we are done
   *
   * There is more data to be read from port.
   JSR   wait_hs        *wait for data available
   LDAB  data           *read byte from data port
   STAB  0,Y            *store byte at Y[0]
   INY                  *increment Y
   DEC   counter        *decrement counter
   BRA   loop_data

read_complete
   LDAB  #TRUE
   STAB  token
   JSR   wait_hs        *wait for EOM to be sent
   RTS

*************************************************************
** pio_write
*************************************************************
   XDEF  pio_write
```

```
pio_write
   LDY   #msgo       *load pointer to message
   LDAB  0,Y         *store Y[0] in ACCB
   STAB  data        *X[0]=Y[0]
   BEQ   write_eom   *if command !=0 , then there is a message

   * There is data (non-zero command) so send it
is_data:
   INY               *increment to length
   JSR   wait_hs     *wait for handshake
   LDAB  0,Y         *load length byte
   STAB  counter     *store in counter
   STAB  data        *send the length
   *
   * Send the data
send_next
   LDAB  counter     *load the counter
   BEQ   write_eom   *if counter==0, then done
   DEC   counter     *counter--
   INY               *increment message pointer
   JSR   wait_hs     *wait for receiver
   LDAB  0,Y         *load the next byte
   STAB  data        *send the byte
   BRA   send_next

   XDEF  write_eom
write_eom
   JSR   wait_hs     * wait before sending EOM
   CLR   data        * send EOM
   CLR   token       * token=FALSE
   RTS

   * coded outgoing message:
   XDEF  msgo
msgo
   FCB   $01,$05,$51,$52,$53,$54,$55
   END
```

Disclaimer

Echelon Corporation assumes no responsibility for any errors contained herein.
No part of this document may be reproduced, translated, or transmitted in any form without permission from Echelon.

Part Number 005-0021-01 Rev. C

Echelon Corporation
4015 Miranda Avenue
Palo Alto, CA 94304
Telephone (415) 855-7400
Fax (415) 856-6153

Echelon Europe Ltd
Elsinore House
77 Fulham Palace Road
London W6 8JA
England
Telephone +44-81-563-7077
Fax +44-81-563-7055

Echelon Japan K.K.
Kamino Shoji Bldg. 8F
25-13 Higashi-Gotanda 1-chome
Shinagawa-ku, Tokyo 141
Telephone (03) 3440-7781
Fax (03) 3440-7782