EIA-232C Serial Interfacing with the Neuron® Chip

January 1995 LONWORKS® Engineering Bulletin

Introduction

The Neuron Chip is a programmable device that includes a rich variety of input/output capabilities. The Neuron Chip's firmware can configure the 11 I/O pins of the processor in more than 30 different modes supported by software drivers. Application programs running on the processor can then access this I/O functionality through simple calls to the I/O driver functions. The Neuron Chip is also a device that can communicate with other Neuron Chips over a variety of networking media, such as twisted-pair wiring and power line, using the LonTalk® protocol. It is thus an ideal device to implement applications for control networks.

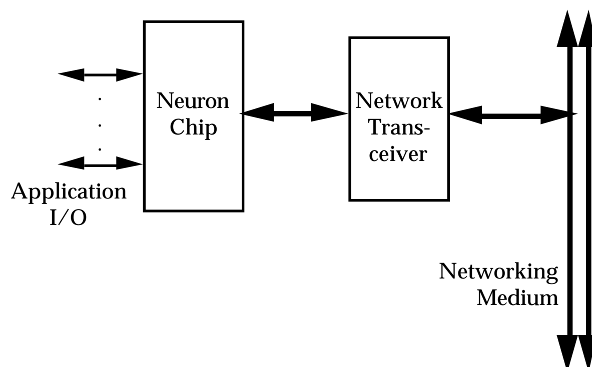

Figure 1. Architecture of a Neuron Chip-based node.

A LONWORKS® network uses a multi-drop concurrent-access architecture, so that multiple nodes can communicate in a peer-to-peer fashion. The RS-485 standard supports such a multi-drop architecture, allowing any node to communicate with any other node on the network, with up to 32 nodes per physical channel. On the other hand, the EIA-232C standard (formerly known as RS-232C) is a point-to-point architecture, which allows only two devices to communicate with each other. The standard was originally designed for communications between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE) such as modems. However, it has been widely applied in recent years to other point-to-point communications needs.

This engineering bulletin describes implementation of an EIA-232C asynchronous serial interface that enables a Neuron Chip to communicate with another device that employs the EIA-232C standard. This could be, for example, a PLC, machine

controller, CRT terminal, printer, card reader, or modem. The same Neuron Chip can also be a part of a network communicating with the LonTalk protocol, forming a gateway between an EIA-232C link and a LONWORKS network.

The Neuron Chip does not contain any UART hardware; instead, the serial I/O interface is implemented in firmware, moving data a bit at a time using the Neuron Chip's application CPU. If bit rates higher than 4,800 bps or serial data buffering greater than one bit are required, then an external hardware UART should be used. Input buffering is especially important. With a software UART, the Neuron Chip's application CPU must be waiting at an input statement when the start bit of the asynchronous character arrives. At 4,800 bps, a bit time is approximately 200 µs, which is below the typical event latency for a Neuron C program. However, if hardware buffering is used, the required latency is much greater. For example, if the UART has a 16-character input FIFO, messages of up to 16 characters in length may be received at any supported bit rate without loss of data, and the Neuron C program may read this data at its leisure.

The PSG/2 and PSG-10 Programmable Serial Gateways are suitable for handling high speed asynchronous data streams. They include a Neuron 3150 Chip with a 5MHz or 10MHz input clock, a socket for a PROM to contain system and application firmware, and a 16550-compatible UART. This UART provides sixteen characters of serial input and output data buffering, and bit rates of up to 115,200 bps. A low-level software library for creating serial gateway applications is included with LonBuilder® and NodeBuilder™ development systems. For more details on the Programmable Serial Gateway, see the *Serial LonTalk Adapter and Serial Gateway User's Guide.*

For applications where a serial interface to a host computer is required, the SLTA/2 or LTS-10 Serial LonTalk Adapters may be used. These are pre-programmed serial interface devices based on the same hardware as the Programmable Serial Gateway devices. The on-board firmware communicates with a driver on the host computer, and supports all network management functions, as well as remote access via modems. The Serial LonTalk Adapter firmware moves the LonTalk application layer to the host computer, giving it the capability to send and receive network variables and to implement LONMARK objects. Source code for DOS and UNIX drivers is included. For more details, see the *Serial LonTalk Adapter and Serial Gateway User's Guide.*

This engineering bulletin only describes the use of the Neuron C serial I/O object, which is implemented as a software UART.

Asynchronous Data Format

The Neuron Chip receives and transmits serial data using eight-bit character frames, with one start bit and one stop bit. The EIA-232C standard defines two voltage levels.

The negative voltage corresponds to logic level 1 and the positive voltage to logic level 0. When the line is idle, it is at logic level 1. A character frame begins with a start bit, which holds the line at 0 for one bit time. Then the eight data bits are transmitted with the least significant bit first. Finally, the line returns to 1 for at least one bit time, forming the stop bit. A new character frame may start at any time after the end of the stop bit. This asynchronous data format is commonly used with the EIA-232C standard interface, although strictly speaking, it is not a part of the standard. It is termed asynchronous since it is not necessary to share a clock between the transmitting and receiving devices. Both devices can use independent local clocks running at the same nominal frequency. Actual synchronization is on a character-by-character basis using the start and stop bits.

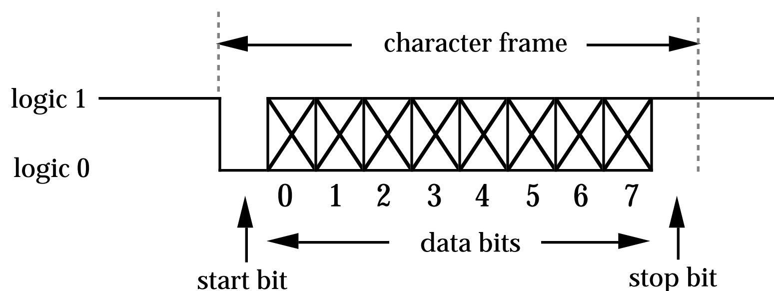

Figure 2. Asynchronous data format.

Hardware Considerations

The Neuron Chip supports this asynchronous serial data format using the serial I/O object. The serial output object is implemented on IO10 pin and the serial input device on the IO8 pin . The nine remaining I/O pins can be used for other I/O objects. The I/O pins have TTL input levels and standard CMOS output levels. Devices such as the Motorola MC145407 may be used to convert these levels to and from EIA-232C voltage levels. Figure 3 shows a typical schematic for a bi-directional EIA-232C interface for a Neuron Chip configured as Data Terminal Equipment (DTE). The interface chip chosen is a 5-volt-only driver/receiver that uses an on-chip charge pump to generate the EIA-232C voltage levels with the help of four external capacitors.

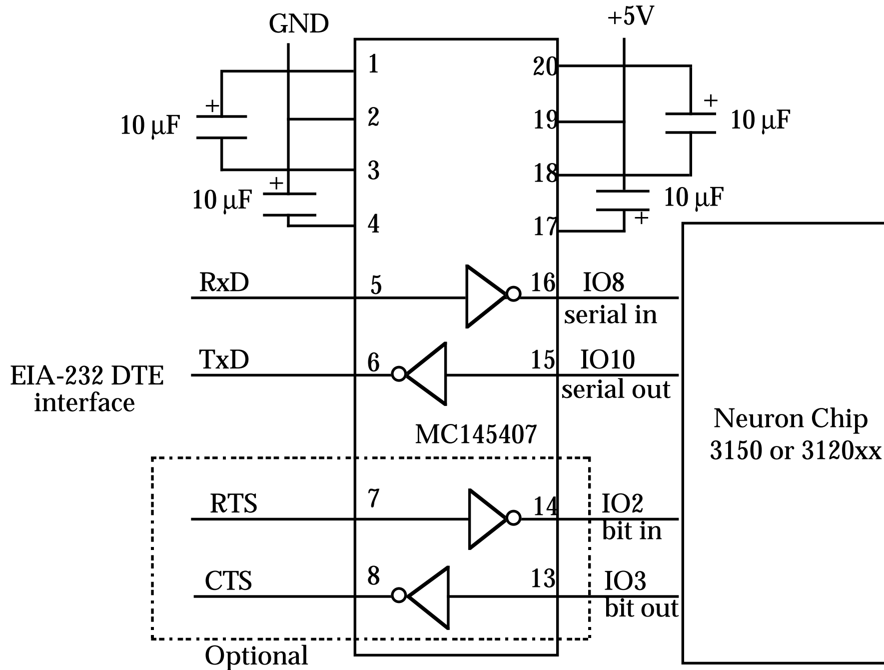

Figure 3. Typical EIA-232C interface circuit.

If additional modem control lines such as CDET, DSR, DTR, CTS, and RTS are required, then any of the other Neuron Chip I/O lines may be configured as bit input or output lines under control of the application software. Not all of these signals are active high. The application developer should check which sense is valid and handle each one of them appropriately. For full details on the modem control lines, see the Electronics Industries Association EIA-232C Standard document.

Software Considerations

The designer uses statements in the Neuron C programming language to declare and activate serial I/O objects. This provides a high-level interface that frees the application designer from considerations of bit timing, data framing, and character assembly and disassembly.

Serial Output

To declare a serial output object, a statement of the form should be used:

```
IO_10 output serial baud (constant) io_object_name;
```

For example, the following statement declares the device named `CRT_screen` as a serial output object operating at 1200bps:

```
IO_10 output serial baud(1200) CRT_screen;
```

To output data to the serial device, use a statement of the form:

```
io_out(io_object_name, buffer_pointer, character_count);
```

For example, the following statement sends the twelve ASCII characters `"Hello, world"` in serial format on the IO10pin:

```
io_out(CRT_screen, "Hello, world", 12);
```

The parameters to the `io_out()` function call for a serial output object are:

`io_object_name`	A name declared as a serial output object
`buffer_pointer`	A parameter of type `(const char *)` pointing to an array of characters
`character_count`	A parameter of type `(unsigned int)`

The `io_out()` function call suspends execution of the application program until all the characters have been transmitted on the output pin. For example, transmitting 120 characters at 600bps will suspend the application for 120 * (1 + 8 + 1) / 600 = 2 seconds. During serial I/O, the network and media access CPUs on the Neuron Chip continue to execute the network protocol, but the application processor does not execute other tasks to handle any generated events. Application designers should take this into account.

The allowable values for the constant expression used to specify the bit rate are 600, 1200, 2400, and 4800, which refer to the serial bit rate at a Neuron Chip input clock rate of 10 MHz. If other input clock rates are used, refer to Table 1.1.

Input Clock (MHz)	baud(600)	baud(1200)	baud(2400)	baud(4800)
10.0	600	1,200	2,400	4,800
5.0	300	600	1,200	2,400
2.5	150	300	600	1,200
1.25	75	150	300	600
0.625	37.5	75	150	300

Table 1. Serial bit rates for different Neuron Chip input clock rates.

Serial Input

To declare a serial input object, a statement of the form should be used:

```
IO_8 input serial baud(constant) IOobjectName;
```

For example, the following statement declares the device named `keyboard` as a serial input object, operating at 600 bps:

```
IO_8 input serial baud(600) IOkeyboard;
```

To input data from the serial device, a statement of the form should be used:

```
num_chars_received =
        io_in(IOobject_name, pBuffer, maxCharacterCount);
```

For example, the following statement causes the Neuron Chip to wait for up to twelve serial characters to be received in serial format on the IO_8 pin:

```
numCharsReceived = io_in(IOkeyboard, inputBuffer, 12);
```

The parameters to the `io_in()` function call for a serial input device are:

IOobjectName	A name declared as a serial input object
pBuffer	A parameter of type `(char *)` pointing to an array of characters to accept the received data
maxCharacterCount	A parameter of type `(unsigned int)`

The `io_in()` function returns a value of type `(unsigned int)`, indicating the number of characters actually received. Suitable declarations for the above example are:

```
char inputBuffer[12];
unsigned numCharsReceived;
```

Note that in the C language, an object of type `(char[])` is automatically promoted to type `(char *)` in the context of a function call. It is the designer's responsibility to ensure that the input buffer for serial input is large enough to contain the number of characters requested. If the `io_in()` call specifies a maximum character count that exceeds the size of the input buffer, unpredictable behavior will result.

The `io_in()` function suspends application processing until it is complete. This occurs at the first of the following:

1. The number of characters specified in the `maxCharacterCount` parameter of the function call have all been received,
2. The line has been continuously at the idle level for 20 character times, for example 166.7 msec at 1200 bps, or
3. A framing error has occurred – an expected start bit or stop bit has the wrong polarity.

In all cases, the returned value of the function is the number of characters actually received and stored in the buffer.

The Neuron Chip's application CPU is suspended during execution of the serial `io_in()` function. This means that it cannot process other I/O events, timer expirations, network variable updates, or incoming messages while it is waiting for

input characters on the serial port. If no input occurs for 20 character times, then execution proceeds after the `io_in()` call. The application can simply tickle the watchdog timer to avoid resetting the node, and re-enter the serial input call. This has the advantage that the application is unlikely to miss any input characters, because it is almost always in the tight loop waiting for input. For example:

```
#include <control.h>                               // define 'watchdog_update'
IO_8 input serial baud( 2400 ) IOcharIn;           // serial input device
char inputChar;                                    // place to store received
                                                   // character
unsigned numChars;
when( serial input is desired ) {
   do {                                            // tight loop until character
                                                   //is received
      numChars = io_in(IOcharIn,&inputChar, 1);    // read one character
      watchdog_update( );                          // avoid watchdog timeouts
   } while( numChars == 0 );                       // if error or time_out,
                                                   //try again
   process_char( inputChar );                      // process this character
}
```

However, the node is frequently required to process other input events besides the serial character input. In this case, the application processor must periodically return to the event scheduler to handle the other events. This has the disadvantage that the application processor cannot handle serial input while it is processing those events. There is a chance that some characters will be missed, depending on the relative time taken to process these other events, and how fast the application processor can return to the `io_in()` call. For example, to process other events every time the `io_in()` call returns, the following code can be used:

```
IO_8 input serial baud( 2400 ) IOcharIn;           // serial input device
char inputChar;                                    // place to store received
                                                   // character
unsigned numChars;

when (1) {                                         // do this every pass through
                                                   // the scheduler
   numChars = io_in(IOcharIn, &inputChar, 1 );     // read one character
   if (numChars != 0) ProcessChar(inputChar);
                                                   // process this input
                                                   // character
}                                                  // end of task, return to
                                                   // scheduler to handle other
                                                   // events
```

```
when (other events)........                    // process NV updates,
                                               //messages etc.
```

If the io_in() call is successful, and a character has been received, the io_in() call will return to the calling task approximately at the center of the stop bit. The Neuron C application must re-enter the io_in() call before the beginning of the next start bit. If the characters are being transmitted back-to-back with one stop bit, that means that the application has approximately half of a bit time to process the character. At 1200bps, that is about 400µs. If more time is desired, it will be advantageous to have the sending device use two or more stop bits, since this will extend the idle period before the start bit of the next character. The best solution may very well be to spend most of the time waiting for serial input to occur, and occasionally returning to the scheduler to check for network and timer events.

Synchronizing with an External Serial Device

Because the serial input object is implemented using software-driven serial I/O, there are a few restrictions in the use of serial input:

- The application program must be waiting at an io_in() function call when the start bit of the character is received. If the call to io_in() is made after the beginning of the start bit of a character, an immediate framing error is likely. If the call to io_in() is made after the end of a character, then that character will be totally missed.
- If the start bit of the first character is delayed more than 20 character times after calling the io_in() function, then the call will time out and no characters will be returned. A time-out will also occur if there is a pause of more than 20 character times between the end of one character and the beginning of the next. In this case, the call will return the characters received up until the time-out.

Solutions to these limitations depend on the device that is generating the input characters. If the input device is not an operator typing at a keyboard, but rather another processor, then some form of handshaking can be implemented. For example, the sending device can indicate its desire to transmit by raising a request to send (RTS) signal connected to a bit input of the Neuron Chip. The Neuron Chip detects the request to send, and activates a bit output that indicates to the sending device that the Neuron Chip is ready to receive data (a clear-to-send, or CTS signal). The Neuron Chip then immediately enters the io_in() call for the serial input object. The external device, when it receives the CTS indication, waits until the Neuron Chip is in the io_in() call, and then transmits its characters. If the number of characters transmitted is fixed in advance, then the io_in() call can specify this number of characters. Alternatively, it can wait 20 character times for the time-out, or the input device can generate a *break* condition on the line at the end of the data,

causing a framing error and termination of input. Or, the Neuron Chip can read a single character at a time, and look for a terminating character such as a carriage return. The following example assumes that the input line is sent without pauses greater than 20 character times between the characters, and that a carriage return is always sent at the end of the line, which is never longer than 120 characters.

```
IO_3 output bit CTS;             // clear to send output
IO_2 input bit RTS;              // request to send input
IO_8 input serial RXD;           // serial data input
char inputBuf[120];              // RAM buffer for input line
char * pBbuf;                    // pointer into buffer

when (io_changes(RTS) to 1) {    // wait for request to send
   pBuf = inputBuf;              // initialize buffer pointer
   io_out(CTS, 1);               // raise clear to send
   do {
      (void)io_in(RXD, pBuf, 1); // get one character into buffer
   } while (*pBbuf++ != '\r');   // keep going if not CR
   io_out(CTS, 0);               // drop clear to send
   ProcessBuffer();              // handle input line
}
```

Using Serial I/O to Interface with a CRT Terminal

If the device generating characters is an operator typing at a keyboard, then there are user interface issues to be considered. Normally, an operator will type with unpredictable pauses between characters. These pauses will probably exceed the 20 character times time-out limit. The serial `io_in()` function does not check for any input terminators such as the ASCII carriage return character, which is conventionally used to indicate the end of user input. It also does not echo input characters, nor does it recognize any input line editing characters such as back-spaces, as would be expected by a user typing at an interactive terminal device.

If this kind of functionality is required, it must be implemented by the application code calling `io_in()` and `io_out()` to perform the basic I/O. The following code shows how some of these interactive terminal capabilities might be implemented. The function `GetLine()` reads characters one at a time from the serial input device, processing some control characters and echoing printable characters to the serial output device. A null-terminated string is returned in `inputBuf`.

Following the function `GetLine()` is a demonstration task that calls `GetLine()` to read a line from the keyboard and then echoes it back to the screen.

```
// Sample EIA-232 driver for an interactive serial ASCII keyboard device

#include <control.h>
IO_8   input serial baud(2400) IOcharIn;  // serial input device
IO_10 output serial baud(2400) IOcharOut; // serial output device

char inputBuf[120];                       // input buffer for GetLine

unsigned GetLine(void) {
      // function waits for input line, returns number of characters received

   char * pInputBuf;    // next place to store received character
   char * pEndOfBuf;    // last byte in buffer
   char   thisChar;     // character being processed
   unsigned numChars;

   pInputBuf = inputBuf;            // initialize buffer pointers
   pEndOfBuf = inputBuf + sizeof( inputBuf ) - 1;

   while (pInputBuf < pEndOfBuf) {  // don't read past end of buffer

      do {
         numChars = io_in(IOcharIn, pInputBuf, 1); // read one character
         watchdog_update();            // avoid watchdog timeouts
       } while (numChars == 0);        // if error or time_out, try again

      thisChar = *pInputBuf & 0x7F;    // discard any parity bit
      if (thisChar == '\r') break;     // exit loop if a carriage return

      if ((thisChar == '\b') || (thisChar == 0x7F)) {
                                          // backspace or rubout
         if (pInputBuf > inputBuf) {      // buffer is not empty
            pInputBuf--;                  // forget last character
            io_out(IOcharOut, "\b \b", 3);   // erase character on screen
         }
         continue;
      }
      if (thisChar < ' ') continue; // ignore other control characters
      *pInputBuf = thisChar;        // store this character
      io_out(IOcharOut, pInputBuf++, 1);
                                    // echo this character, increment pointer
   }  // end while

   io_out(IOcharOut, "\r\n", 2);    // send CR LF at end of line
   *pInputBuf = '\0';               // null terminate string
   return (unsigned)(pInputBuf - inputBuf);
}                                   // return number of characters

// Demonstration task for the GetLine() function
unsigned numChars;
```

```
when (TRUE) {                  // do forever
   numChars = GetLine();       // read a line
   if (numChars != 0) {
      io_out(IOcharOut, output_buf, numChars);  // echo line
      io_out(IOcharOut, "\r\n", 2);             // send CR LF
   }
}
}
```

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 005-0008-01 Rev. D

©1991 - 1995 Echelon Corporation. ECHELON, LON, Neuron LonTalk, LONWORKS ,3150, and 3120 are U.S. registered trademarks of Echelon Corporation. Other names may be trademarks of their respective companies. Some of the LONWORKS tools are subject to certain Terms and Conditions. For a complete explanation of these Terms and Conditions, please call 1-800-258-4LON or +1-415-855-7400

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,
Hammersmith,
London W6 8JA, England.
Telephone +44-81-563-7077
Fax +44-81-563-7055

Echelon Japan K.K.
Kamino Shoji Building 8F
25-13 Higashi Gotanda 1-chome
Shinagawa-Ku, Tokyo 141, Japan
Telephone 011-81-3-3440-7781
Fax 011-81-3-3440-7782