Building a LONTALK®-to-PLC Gateway

May 1994 **LONWORKS™ Engineering Bulletin**

Introduction

In the industrial controls arena, *centralized* Programmable Logic Controllers (PLCs) have long been the standard control system. LONWORKS technology offers a powerful means for implementing industrial control systems that perform *distributed* sensing, monitoring, and control. As users migrate to distributed control systems that offer lower cost and better diagnostic capability per I/O point, these systems will be found side by side with PLCs on the factory floor.

This bulletin describes a general approach for building a gateway between a Programmable Logic Controller and a LonTalk network, with specific examples coming from a gateway implemented between a LonTalk network and an Omron PLC.

Outside of industrial control applications, users of the Echelon Programmable Serial Gateway will find this sample gateway application useful for the general concepts that are necessary for all gateway products: management of the serial link and migration of control points between a LonTalk network and another type of network.

The example code in this bulletin was first demonstrated on an Omron CV500 PLC in September 1993. Echelon has no plans to maintain or support this example.

Glossary of Terms

The following terms are used throughout this bulletin:

Gateway Demo - a gateway to an Omron PLC that was implemented by Echelon for the Instrument Society of America trade show in September of 1993. This demo included an Echelon LonWorks Programmable Serial Gateway (PSG) and an Omron CV500-BSC21 EIA-232 interface unit. The Omron unit is programmed using the Basic programming language.

I/O points, blocks and modules - an I/O point is an interconnection between a logic controller and one logical device. An I/O block is a grouping of I/O points from the perspective of the control logic. An I/O module is a hardware interface, typically providing physical connection to several I/O points.

PLC - Programmable logic controller. A centralized, user-programmable industrial I/O controller.

PLC side - The PLC side of the gateway, consisting of the Omron programmable EIA-232 interface.

PSG, **PSG side** - the term *programmable serial gateway* refers to two Echelon products, each containing a Neuron Chip, PROM socket, and UART, which may be programmed by the user to implement a serial gateway to a LonTalk network. The model 65200 PSG-10 product is a single in-line module (SIM) that may be embedded in OEM products. The model 73000-3 Programmable Serial Gateway is the packaged Echelon Serial LonTalk Adapter (SLTA) product without the SLTA firmware or control module. The Gateway Demo was implemented with the 73000-3 PSG, custom firmware, and a TP/XF-78 twisted-pair control module.

Overview of PLC Architecture

PLCs provide the ability to monitor and control industrial processes. The interface to the physical industrial process is a variety of process control signals conveying temperature, relay status, motor speed, and other process control information. The PLC senses and controls the various process control signals according to the user-defined control logic.

A PLC typically consists of a main system controller unit and I/O modules. The main system controller is responsible for executing the user program. The I/O modules provide the interface to the I/O devices. The main system controller may be packaged in a separate box, as seen in figure 1, or in the same type of enclosure as the I/O modules. The interface between the main system controller and the I/O modules is a proprietary design of the manufacturer.

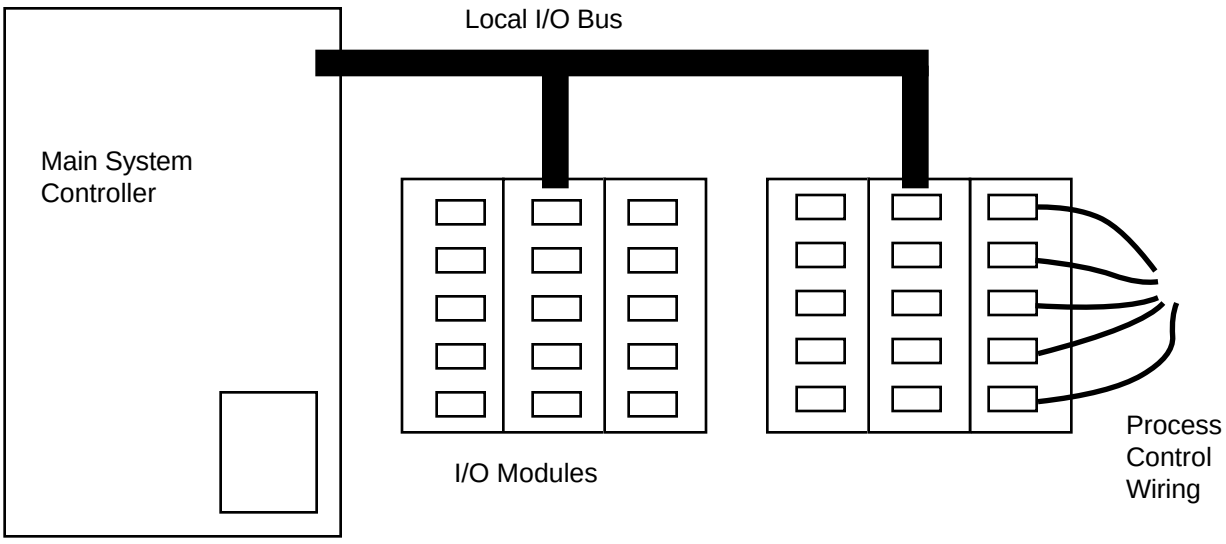

Figure 1 A Typical PLC Architecture

PLC manufacturers typically provide an EIA-232 interface to the main system controller—either directly integrated in the system controller or in a form factor similar to an I/O module, as shown in figure 2. This EIA-232 interface allows an

interface device to exchange data with the main system controller, enabling third-party devices to be integrated into the industrial automation system. Echelon's serial gateway products, the PSG-10 and the model 73000-3 Programmable Serial Gateway, can be programmed to communicate with PLCs through an EIA-232 interface, allowing Neuron-based devices to be integrated into the PLC automation system.

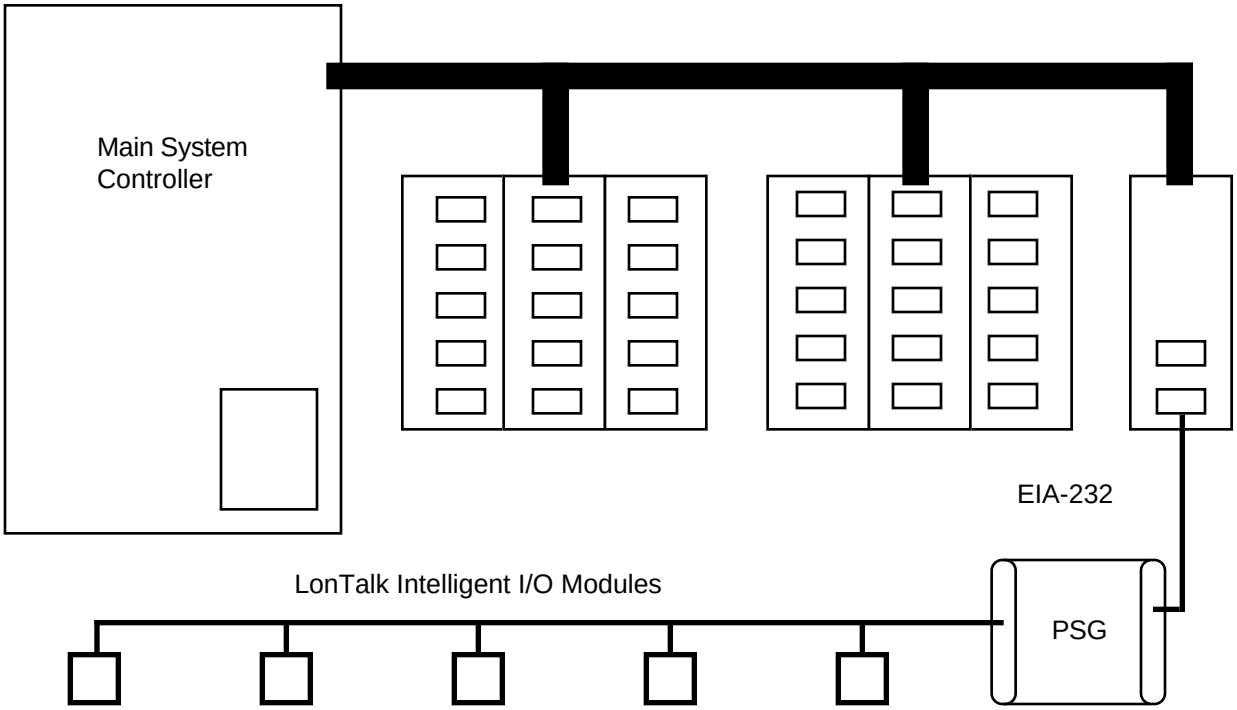

Figure 2 PLC to LonTalk Connectivity

Migration of Control Points

PLC I/O points are controlled through a register map within the system controller. Blocks of I/O points are mapped into registers to allow easy manipulation by the PLC control program. For example, the Gateway Demo includes 16 input switches and 16 output LEDs, for a total of 32 I/O points. Since each of these control points may only have status values of ON or OFF, each may be mapped to a control register using just one bit. Thus, one 16-bit input word and one 16-bit output word are brought out to the LonTalk network by the Gateway Demo.

LonTalk I/O points, known as *network variables*, are defined to allow devices from disparate manufacturers to communicate with each other. Echelon's Standard Network Variable Types, as described in [2], provide standard units of measurement for common control quantities such as pressure, temperature, and volume.

The task of the gateway is to translate input points from the PLC into output points on the LonTalk network, and input points on the LonTalk network to output points

on the PLC. The Gateway Demo does this through a set of translation tables, as shown in figure 3.

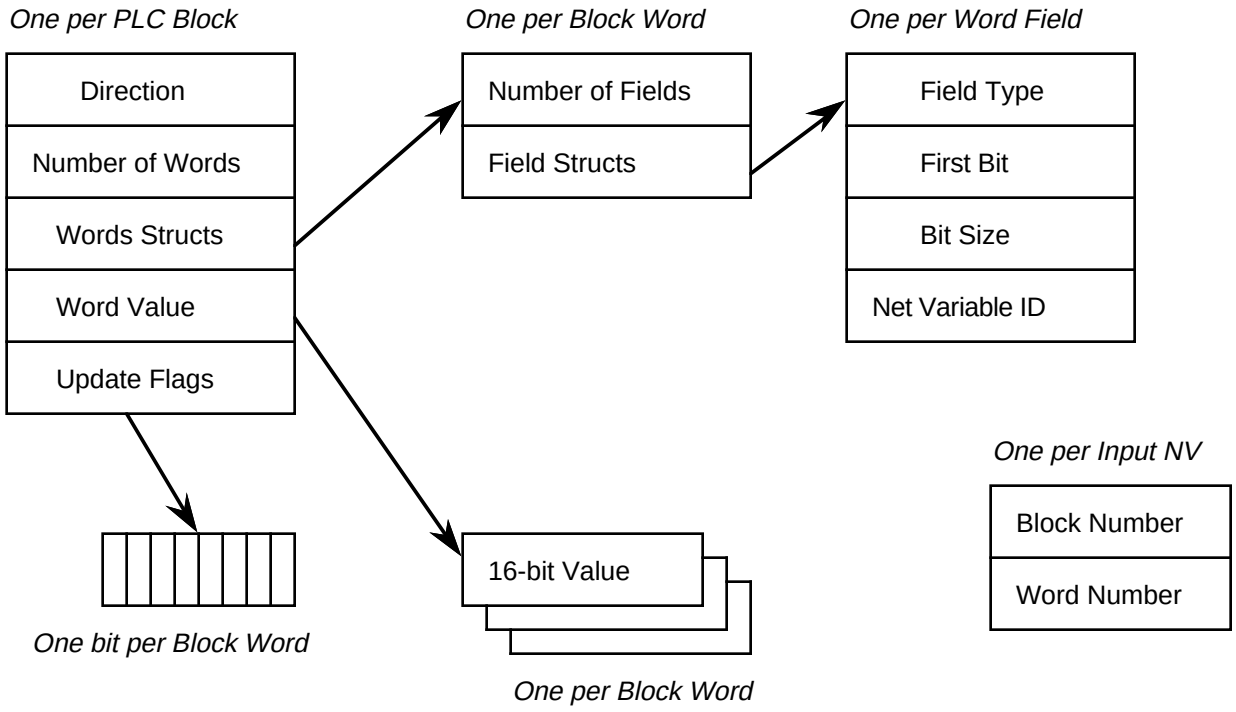

Figure 3 Translation Tables for PLC Gateway

The Gateway Demo groups common control points into I/O *block*s. PLC I/O blocks are assumed to consist of one or more 16-bit words, with a *word* being the smallest unit of update data sent on the PLC-PSG link. Each word is composed of one or more *field*s, where each field represents a PLC control point that will be converted to a LonTalk control point. The field types are: 1) bit fields, consisting of from 1 to 15 bits at any location in the word, 2) char fields, consisting of 8 bits, starting at either bit 0 or bit 8 in the word, or 3) long fields, consisting of the entire 16-bit word.

Each word update that arrives over the serial link is processed by looking at the mapping tables. Once the update is reduced to its component fields, a site-specific routine, `set_nv()` is called to perform any necessary normalization and set the NV value for propagation over the network.

In the other direction, when an input network variable is updated, it determines via table lookup which PLC word should change and sets the update flag for that word. The next time that the serial link state machine checks for pending updates to the PLC, it determines which words should be updated and assembles the new values for those words. This is done by breaking the word up into fields, just as in the PLC

input case, and then calling a site-specific routine, `get_nv()`, to transform an NV value into a field value.

The site-specific mapping tables as well as `set_nv()` and `get_nv()` are in the file `xlate.nc` of the demo software. The input NV processing resides in a **`when`** statement in the file `plcgate.nc`.

Serial Link Protocol

The serial gateway protocol between the PLC side and the PSG has to meet a few requirements. It must reliably handle control point updates in both directions on the serial link, arbitrating ownership of the link. It must verify data received over the serial link. It must be flexible enough to handle a wide range of functionality on the PLC side—from dedicated serial port processors to multi-tasking control units which handle serial input in the background.

The serial link control state machine is shown in figure 4. The states shown are:

IDLE - No link activity.

GUARD - Behaves just like IDLE on incoming events from the PLC gateway side. No outgoing messages are initiated while in this state. It is used, for example, right after reset to listen for activity on the serial link when resynching. As shown in figure 6, when a transmitted message gets a negative acknowledgment, it is also used to give the other side a chance to transmit before retrying.

RX_MSG - Receiving an incoming message. An incoming ALERT has been acknowledged, and we are processing incoming data bytes.

RX_NACK - An incoming message has been invalidated, and we are throwing away incoming data bytes and waiting a brief amount of time before sending a NACK.

TX_REQ - An outgoing message request (ALERT) has been sent, and we are waiting for an ALERT ACK.

TX_MSG - Transmitting an outgoing message.

TX_DONE - Done transmitting an outgoing message, we are waiting for an ACK or NACK from the other side.

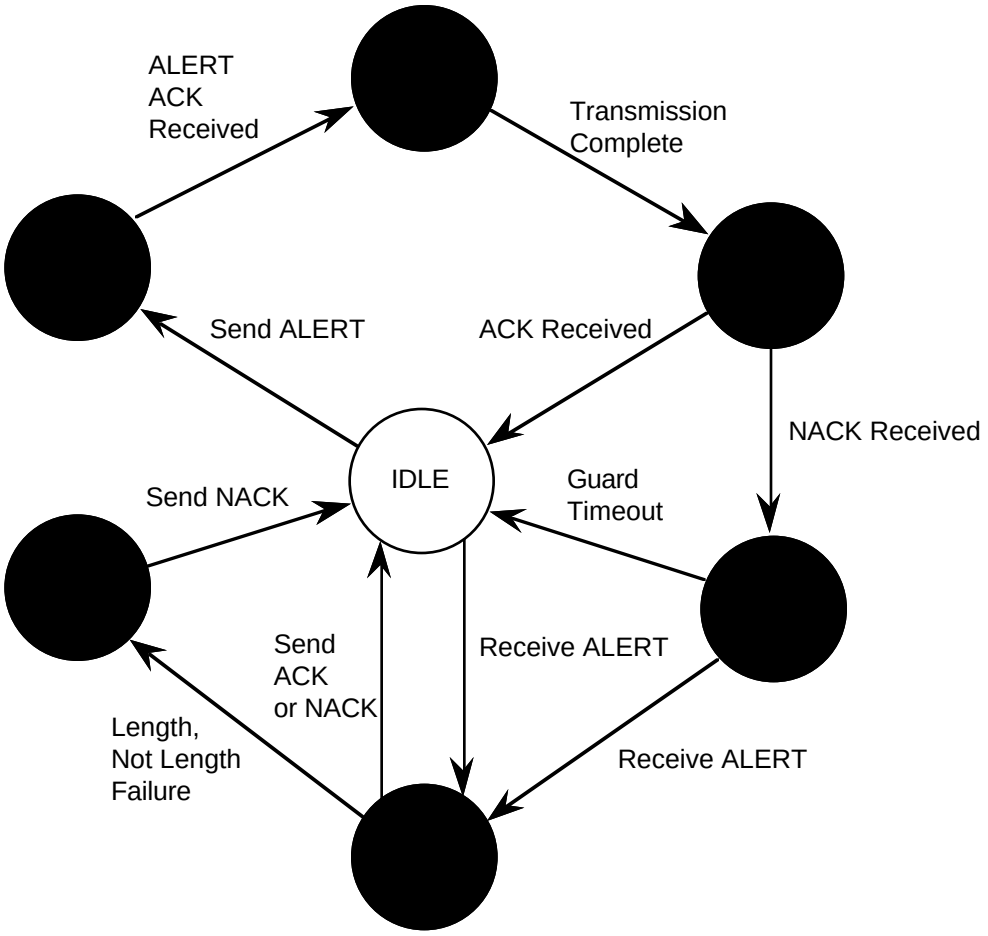

Figure 4 The Serial Link State Machine

Due to space limitations, state transitions due to timeouts are not included in the diagram. The one exception is the transition from GUARD to IDLE state, which is the only non-exception timeout in the state machine. Other timers and their timeout handling may be examined in `plcgate.nc`. Timeout values will be site-specific, dependent on the serial bit rate, processor speeds, UART buffering, etc. To tune the timeouts for a particular site, choose likely values and observe the numbers of exception timeouts logged in the Gateway Demo's `dbg_...` variables. Be sure that initial timeout values are low enough that you see some timeouts, then raise them until they are rare. See the chapter *Using a Programmable Serial Gateway* in [1] for information about using the LonBuilder debugging environment for tuning the Gateway Demo for your site.

The protocol defined here is meant to contain a superset of features that may be necessary at a particular site. The link between the gateway sides is a point-to-point link which needs to communicate control point status changes. In most cases, and certainly for the Gateway Demo, all status updates for the gateway control points

will fit into one update message. Successful completion of an update message means that, for the time being, both sides of the gateway agree on the state of its control points. Features of the link protocol include:

- An ALERT/ACK sequence that ensures that the receiver has time to prepare for message reception.
- Length, Not Length, and Checksum bytes to verify that the message bytes are received in order and without bit errors.
- An ACK/NACK to tell the sender when an update message has been successfully received.

The link layer packet format is shown in figure 5. If the PLC side is capable of reacting quickly to serial activity and buffering characters, the ALERT/ACK sequence can be replaced with a simple ALERT byte at the beginning of the message. In this case, the TX_REQ state should be removed from the state machine as well as the sending of ALERT ACK byte in the IDLE and GUARD states. This will speed up response time on a slow serial link.

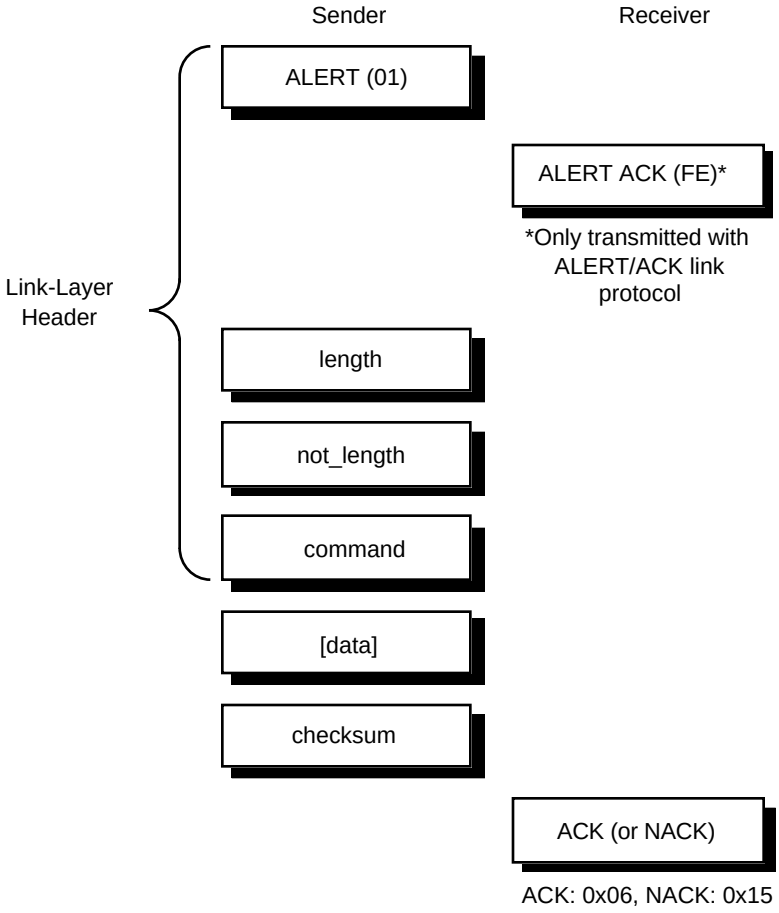

Figure 5 Link Layer Protocol

There are two update message formats, as shown in figure 6. The Word Update message will probably be more common during normal system operation, since only one or two words per block are likely to change within an update window. The Block Update message is more useful for resynchronization (see next section) or where many words are likely to change state simultaneously.

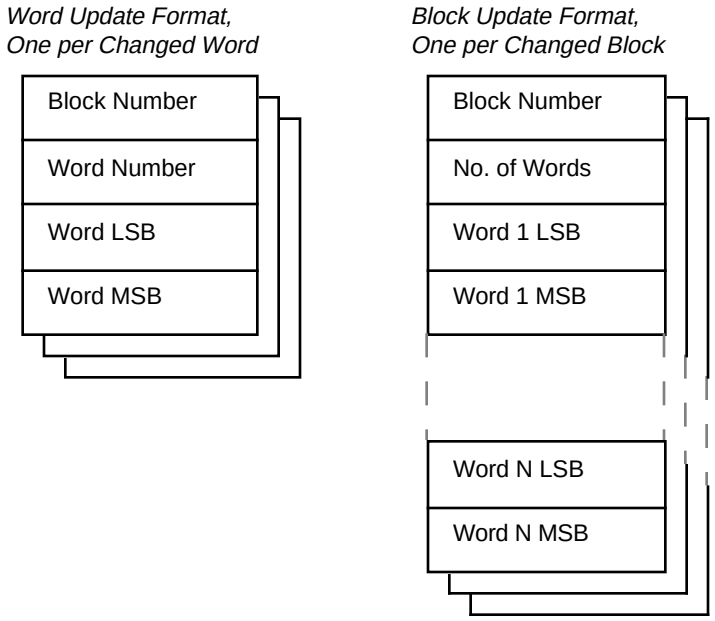

Figure 6 Update Message Data Formats

Resynchronization upon Reset

In a distributed network, one of the most challenging aspects of the design is handling resynchronization of the network when some group of nodes is reset. Data that is distributed throughout a system must find its way back to the places where it is needed.

Since the PLC gateway will be part of a larger system that will have a specific data flow, this problem may be site-specific. The Gateway Demo, however, assumes that the PLC is the master in a master-slave system.

When the Gateway Demo is reset, the PSG side of the gateway sends a Request Resynch message to the PLC side and waits for a response. The PLC side responds by sending a current status for all I/O blocks, both input and output, that are known to the gateway. The PSG side updates all network variable values during this time, output *and input*. The end of this phase is signaled by a Resynch Complete command from the PLC side. If the PSG side gets input NV updates after the reset

but before the PLC side resynchs, they will not be overridden by the resynch information, and instead will be sent to the PLC side after the resynch is complete.

Left As an Exercise for the User

Perhaps as important as the description of what is included in the Gateway Demo is a description of what is not.

The Gateway Demo has only been tested as a demonstration application. When integrating it into one of your products, a thorough unit and system test cycle is required.

No performance analysis has been done on the Gateway Demo. Any such analysis would specifically apply to the Gateway Demo, and not be generally applicable. Performance factors to keep in mind when building a gateway are: How often can PLC registers be polled? What is the maximum speed of the PLC's EIA-232 link? Can the PLC's EIA-232 unit handle asynchronous serial input without the ALERT/ACK sequence? Is it acceptable to throw away updates when you get a checksum error on a received message? In a real installation, expected traffic loads should be tested and error rates and handling verified.

No diagnosis of the serial link is included, as it is assumed that the LonTalk network is slave to the PLC control logic. If the LonTalk network has more built-in alarms, the user may want to add a periodic "pinging" of the serial link and a network variable giving the link status.

References

1. Serial LonTalk® Adapter and Serial Gateway User's Guide, revision 5, Echelon Corporation.

2. The SNVT Master List and Programmer's Guide, Echelon Engineering Bulletin.

3. Neuron C Programmer's Guide, revision 2, Echelon Corporation.

Appendix A - The Programmable Serial Gateway Code

The files appear in the following order: `plcgate.h`, `plcgate.nc`, `xlate.nc`, `rx_msg.nc` and `tx_msg.nc`. These were compiled and tested on LonBuilder release 2.2. Debugging was done on a LonBuilder emulator with a model 73000-3 Programmable Serial Gateway, Module Application Interface, and Application Interface Board as described in [1], in the chapter entitled *Using a Programmable Serial Gateway*. Soft copies of these files are available on LONlink.

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      PLCGATE.H -- Example LONWORKS gateway to PLC.
//      Copyright (c) 1994 by Echelon Corporation.
//      All Rights Reserved.
//
//      Date last modified: 1/31/94
//
//      Definitions file for the Programmable Serial
//  Gateway interface to a PLC.  Definitions include
//  protocol constants, timeout values, and PLC register
//  to Network Variable encoding and decoding.
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#pragma enable_sd_nv_names  // turn on self-documentation

typedef unsigned long  ulong;
typedef unsigned long  plc_word;
typedef unsigned char  bits;



//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      SITE-SPECIFIC SECTION
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// System Timeouts (in milliseconds)
#define TIMER_OFF                   0   // must keep this 0!
```

```
#define INTER_BYTE_TIMEOUT          100
#define OUT_OF_SYNCH_TIMEOUT        200
#define RX_NACK_TIMEOUT             100
#define RETRY_TIMEOUT               100
#define WAIT_FOR_ACK_TIMEOUT        450
#define ALERT_TIMEOUT               450
#define RESYNCH_TIMEOUT             10000


// Current number of PLC register blocks that we are
// passing through.  Input/output is with respect to
// the PLC, so associated Network Variables will have
// the opposite direction.
#define  NUM_INPUT_BLOCKS   1
#define  NUM_OUTPUT_BLOCKS  1
#define  NUM_PLC_BLOCKS     2
#define  NUM_INPUT_WORDS    1
#define  NUM_OUTPUT_WORDS   1
#define  NUM_INPUT_FIELDS   16
#define  NUM_OUTPUT_FIELDS  16
// input NVs go to output PLC register fields
#define  NUM_INPUT_NVS      NUM_OUTPUT_FIELDS
#define  NUM_OUTPUT_NVS     NUM_INPUT_FIELDS


//
// The following enum has a couple of uses:
//  1) index a case statement in the PLC
//     register to output NV conversion
//     routine set_nv().  During normal
//     processing, set_nv() is only used
//     to set output NV values, but during
//     a resynch following reset, input
//     NV values are updated from the PLC.
//  2) Used for input NVs to index into
```

```
//     the nv_to_plc array in order to
//     determine what PLC word must be
//     updated following an NV update.
//     Use the offset from FIRST_INPUT_NV
//     to determine the place in nv_to_plc.
//
//  In order to work, all Input NVs should
//  be consecutive values in the enum.
//
typedef enum {
        NVO_LEV_DISC_0 = 0,
        NVO_LEV_DISC_1,
        NVO_LEV_DISC_2,
        NVO_LEV_DISC_3,
        NVO_LEV_DISC_4,
        NVO_LEV_DISC_5,
        NVO_LEV_DISC_6,
        NVO_LEV_DISC_7,
        NVO_LEV_DISC_8,
        NVO_LEV_DISC_9,
        NVO_LEV_DISC_10,
        NVO_LEV_DISC_11,
        NVO_LEV_DISC_12,
        NVO_LEV_DISC_13,
        NVO_LEV_DISC_14,
        NVO_LEV_DISC_15,

        NVI_LEV_DISC_0,     // Input NVs
        NVI_LEV_DISC_1,
        NVI_LEV_DISC_2,
        NVI_LEV_DISC_3,
        NVI_LEV_DISC_4,
        NVI_LEV_DISC_5,
        NVI_LEV_DISC_6,
        NVI_LEV_DISC_7,
```

```
        NVI_LEV_DISC_8,
        NVI_LEV_DISC_9,
        NVI_LEV_DISC_10,
        NVI_LEV_DISC_11,
        NVI_LEV_DISC_12,
        NVI_LEV_DISC_13,
        NVI_LEV_DISC_14,
        NVI_LEV_DISC_15
} nv_ref_id;
#define FIRST_INPUT_NV  NVI_LEV_DISC_0


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      end SITE-SPECIFIC SECTION
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++


// macros for handling bitmap structures
#define MAX_BITMAP (256/8)

// use the following to size a bitmap array declaration
#define SIZEOF_BITMAP(num_bits) ((num_bits)/8 + 1)

// use the following to manipulate bits within a map
#define SET_BIT(bitmap, id) \
   bitmap[(id)/8] |= (1 << ((id) % 8))
#define CLEAR_BIT(bitmap, id) \
   bitmap[(id)/8] &= ~(1 << ((id) % 8))
#define BIT_IS_SET(bitmap, id) \
   (bitmap[(id)/8] & (1 << ((id) % 8)))


// Macros for serial protocol encoding and decoding
#define ALERT               0x01    // Alert for msg receipt
#define ALERT_ACK           0xFE    // ack upcoming msg
```

```
#define ACK                 0x06    // verify msg receipt
#define NACK                0x15    // Negative acknowledge
// message types
#define WORD_UPDATE_MSG     0x81    // block/word format
#define BLOCK_UPDATE_MSG    0x91    // Update format of block
#define REQ_RESYNCH         0xA1    // Req resynch from PLC
#define RESYNCH_COMPLETE    0xB1    // PLC resynch done


// Macros for locations of bytes within a message record
#define MSG_LEN         0
#define MSG_NOT_LEN     1
#define MSG_CMD         2


// Message header includes a length byte
// and a bitwise NOT of the message length.
#define HEADER_SIZE     2

// A one-byte checksum at the end of the
// message is not included in the msg length
#define CHECKSUM_SIZE   1

// Message size does not include the message
// header.  It includes the message type, data
// and one checksum byte. The maximum update
// size was chosen to limit the length of
// timeouts in the link protocol.
// Most updates should be much smaller.
#define MAX_UPDATE_SIZE 64


// A structure of the following type will be used
// to give the mapping from input NV to PLC block
// and word number.
```

```
typedef struct {
    int     block_num;
    int     word_num;
} nv_to_plc;


// The following structures map PLC register blocks
// to LonTalk Network Variables.  Each block is made
// up of words, which are made up of one or more
// fields, which map to Network Variables.
typedef enum {
    FTYPE_CHAR,     // 8-bit field
    FTYPE_LONG,     // 16-bit field
    FTYPE_BITS      // 1 to 15 bit field
} ftype;
typedef struct {
    // Bit order of word is 15 to 0
    ftype           field_type;

    // FTYPE_BITS, first bit of field
    unsigned        first_bit : 4;

    // FTYPE_BITS, number of bits in field
    unsigned        bit_size  : 4;

    // #define uniquely identifying NV
    nv_ref_id       nv_id;
} field_nv;
typedef struct {
    int                num_fields;

    // assoc array of field structures
    const field_nv    *field;
} word_fields;
typedef enum {
```

```
    PLC_OUTPUT = 0,
    PLC_INPUT
} block_dir;
typedef struct {
    // input or output I/O block?
    block_dir           dir;

    int                 num_words;

    // assoc array of word structures
    const word_fields  *word;

    // latest word value (not necessarily PLC value)
    plc_word           *word_value;

    // word update needed? bit flags
    bits               *update_flags;
} block_words;

// The following enum keeps the state of the RS-232 link
// in order to prevent clashes between incoming and out-
// going messages.
typedef enum {
    IDLE,           // Idle, no uplink or downlink traffic

    RX_MSG,         // Receiving a message uplink from PLC

    RX_NACK,        // Received message had invalid header,
                    // wait for right time to NACK

    TX_REQ,         // We have ALERTed, waiting for ALERT-ACK

    TX_MSG,         // Sending a message downlink to PLC

    TX_DONE,        // Done sending, waiting for ACK or NACK
```

```
    GUARD           // A recovery state where we do not
                    // attempt to send any downlink traffic
} link_state;


// After a reset, the gateway tries to resynchronize with
// the PLC by sending a REQ_RESYNCH command down to it.  The
// PLC should respond by sending the current status of all
// blocks--input and output--followed by a RESYNCH_COMPLETE
// message.  In a hierarchical system like this, you have to
// trust that the master--the PLC--is alive and well.
typedef enum {
    RESYNCH_NOT_NEEDED,     // no resynch needed

    RESYNCH_REQUEST,        // REQ_RESYNCH in progress

    RESYNCH_IN_PROGRESS,    // in this state from successful
                            // send of REQ_RESYNCH until
                            // either resynch timeout or
                // receive RESYNCH_COMPLETE
} synch_state;


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      PLCGATE.NC -- Example LONWORKS gateway to PLC.
//      Copyright (c) 1994 by Echelon Corporation.
//      All Rights Reserved.
//
//      Date last modified: 1/31/94
//
//      This is an interoperable LonWorks Serial Gateway
//  to Programmable Logic Controller (PLC) implementation.
//      The site-specific code in this example monitors
//  one PLC input word and updates one PLC output word.
```

```
//  Despite the small number of control points shared
//  between the PLC and the LonTalk network, the gateway
//  code is structured to be easily scalable to larger
//  numbers of monitored PLC registers or larger numbers
//  of Network Variables.
//      The serial link protocol is designed to be PLC-
//  independent.  It assumes a PLC I/O register structure
//  that is common, and the link timeouts are configurable
//  to allow for easy tuning.
//
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#include <stddef.h>
#include <slta.h>
#include <SNVT_lev.h>
#include "plcgate.h"


//
//  Debugging error counts for fine-tuning the
//  serial link timeouts and debugging translation
//  tables.  May be removed to free some RAM in
//  the final product.
//
int     dbg_alert_timeout,
        dbg_ack_nack_timeout,
        dbg_header_error,
        dbg_collision,
        dbg_out_of_synch,
        dbg_mapping_error,
        dbg_invalid_ftype,
        dbg_invalid_state,
        dbg_receive_timeout,
        dbg_nacked_msg,
        dbg_early_nack,
        dbg_resynch_timeout;
```

```
char    oos_char;

// Serial Link message buffer and state
unsigned int    msg_buffer [HEADER_SIZE + MAX_UPDATE_SIZE +
CHECKSUM_SIZE];
unsigned int    curr_msg_byte = 0;
link_state      state = IDLE;
boolean         plc_update_pending = FALSE;
synch_state     resynch_state = RESYNCH_REQUEST;

// Serial link protocol timers
mtimer  data_receive_timer;
mtimer  tx_ack_nack_timer;
mtimer  resynch_timer;
mtimer  rx_nack_timer;
mtimer  guard_timer;
mtimer  alert_timer;


//+++++++++++++++++++++++++++++++++++++++++++++++++
//  Include some more code files after the global
//  variable declarations.  The site-specific code
//  is confined to xlate.nc, plcgate.h, and a
//  well-marked section of the current file.
//+++++++++++++++++++++++++++++++++++++++++++++++++
#include "xlate.nc"     // translation tables:  PLC <--> NV
#include "rx_msg.nc"    // receive updates from PLC
#include "tx_msg.nc"    // transmit updates to PLC



// Programmable Serial Gateway initialization
when (reset)
{
    int  i;
```

```
    // Initialize the attributes of the PLC link
    // SITE-SPECIFIC CODE:
    slta_init (format_8N1, baud_19200, intfc_8wire);
    // end SITE-SPECIFIC CODE

    // Queue a request to the PLC for resynch.  After
    // a reset, we reset our current snapshot of all
    // PLC blocks--input and output--based on values
    // we read from the PLC.  We will then let NV
    // updates proceed as usual.

    // First, give any backed up uplink traffic a chance
    guard_timer = INTER_BYTE_TIMEOUT;
    state = GUARD;

}


//
//  Receive bytes on the serial link.  Receiving on
//  the serial link is the highest priority, followed
//  by transmitting on the link.
//
priority when (slta_rxrdy())
{
    char ch;
    ch = (char)slta_getchar();    // read UART

    switch (state)
    {
        // Receive message state:  An ALERT
        // has been received and acknowledged, and
        // we expect a full message to follow.
        case RX_MSG:
```

```
        data_receive_timer = INTER_BYTE_TIMEOUT;
        msg_buffer[curr_msg_byte] = ch;
        ++curr_msg_byte;
        if (curr_msg_byte < HEADER_SIZE)
        {
            ; // No error checking to be done yet
        }
        else if (curr_msg_byte == HEADER_SIZE)
        {
            if (~msg_buffer[MSG_LEN] !=
              msg_buffer[MSG_NOT_LEN])
            {
                ++dbg_header_error;
                rx_nack_timer = RX_NACK_TIMEOUT;
                state = RX_NACK;
            }
        }
        else if (curr_msg_byte ==
    (msg_buffer[MSG_LEN]+HEADER_SIZE+CHECKSUM_SIZE))
        {
            data_receive_timer = TIMER_OFF;
            process_uplink_msg();
            curr_msg_byte = 0;

            // check for message to transmit
            state = check_for_transmit();
        }
        break;

    // The receive NACK state indicates that an error
    // occurred while receiving a message, and we
    // are ignoring received chars until the RX
    // NACK timer expires, when we will send a NACK.
    case RX_NACK:
        break;
```

```
// The transmit request state indicates that we
// have sent an ALERT in anticipation of sending
// a downlink message, and are waiting for an
// acknowledgement.
case TX_REQ:
    alert_timer = TIMER_OFF;
    if (ch == ALERT_ACK)
    {   // the other side is ready for our message
        state = TX_MSG;
        break;
    }
    else if (ch == ALERT)
    {   // other side is also sending,
        // receive its message first
        ++dbg_collision;
        slta_putchar (ALERT_ACK);
        curr_msg_byte = 0;
        state = RX_MSG;
        break;
    }

    //  NO BREAK HERE!! Fall through to TX_MSG!!


// The transmit message states indicates that
// we are sending a message downlink.  We do
// not expect to receive any bytes in this
// state, so any received bytes are in error.
case TX_MSG:
    if (ch == NACK)
    {   // The other side probably did not
        // receive the LENGTH and NOT LENGTH
        // bytes accurately.  Set the guard
        // timer just in case the other side
```

```
            // has a message coming right behind
            // the NACK.  When GUARD state times
            // out, it will retry downlink msg.
            ++dbg_early_nack;
            curr_msg_byte = 0;
            guard_timer = INTER_BYTE_TIMEOUT;
            state = GUARD;
        }
        else
        {   // out of synch, try to make the other
            // side back off, enter GUARD state
            ++dbg_out_of_synch;
            oos_char = ch;
            slta_putchar (NACK);
            curr_msg_byte = 0;
            guard_timer = OUT_OF_SYNCH_TIMEOUT;
            state = GUARD;
        }
        break;

    // The transmit done state indicates that we
    // have finished sending a downlink message
    // and are expecting an ACK or NACK from the
    // other side.
    case TX_DONE:
        tx_ack_nack_timer = TIMER_OFF;
        if (ch == ACK)
        {   // update our PLC status to reflect
            // the info we just sent downlink
            state = downlink_msg_success();

            // check for message to transmit
            if (state == IDLE)
            {
                state = check_for_transmit();
```

```
            }
        }
        else if (ch == NACK)
        {   // attempt to resend the message
            // after giving uplink a chance
            ++dbg_nacked_msg;
            guard_timer = RETRY_TIMEOUT;
            state = GUARD;
        }
        else
        {   // unexpected uplink data
            ++dbg_out_of_synch;
            oos_char = ch;
            slta_putchar (NACK);
            guard_timer = OUT_OF_SYNCH_TIMEOUT;
            state = GUARD;
        }
        break;

    // The protocol only allows sending a message
    // after a successful ALERT-ALERT/ACK sequence,
    // so we only expect to see an ALERT char in
    // this state.  The GUARD state is equivalent
    // to IDLE on the receive side.  Since transmit
    // messages are inhibited during GUARD state,
    // it provides a means of quieting the link on
    // this side when the other side is in an
    // uncertain state.
    case IDLE:
    case GUARD:
        if (ch == ALERT)
        {   // Incoming message, ACK it and
            // get into the receive state
            guard_timer = TIMER_OFF;
            slta_putchar (ALERT_ACK);
```

```
                curr_msg_byte = 0;
                state = RX_MSG;
            }
            else
            {   // Fan mail from some flounder?
                ++dbg_out_of_synch;
                oos_char = ch;
                guard_timer = OUT_OF_SYNCH_TIMEOUT;
                state = GUARD;
            }
            break;

        default:
            ++dbg_invalid_state;
            break;

    }
}


//
//  Transmit bytes on the serial link.  The placement
//  of this WHEN clause and its character-at-a-time
//  output mode allow us to check for collisions on
//  the serial link.
//
priority when ((state == TX_MSG) && slta_txrdy())
{
    if (curr_msg_byte == 0)
    {   // Assemble message before sending
        state = assemble_downlink_msg();
        if (state == TX_MSG)
        {   // if assembly successful, still in TX_MSG state
            slta_putchar (msg_buffer[curr_msg_byte]);
            ++curr_msg_byte;
```

```
        }
    }
    else
    {   // send a character downlink
        slta_putchar (msg_buffer[curr_msg_byte]);
        ++curr_msg_byte;
        if (curr_msg_byte ==
       (msg_buffer[MSG_LEN] + HEADER_SIZE + CHECKSUM_SIZE))
        {   // done sending message, wait for ack or nack
            curr_msg_byte = 0;
            tx_ack_nack_timer = WAIT_FOR_ACK_TIMEOUT;
            state = TX_DONE;
        }
    }
}


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//                SITE-SPECIFIC CODE:
//      Handle incoming network variable update
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
when (nv_update_occurs(NVI_lev_disc))
{
    SNVT_lev_disc value;
    unsigned int  offset;

    value = NVI_lev_disc[nv_array_index];// get new value

    // identify the word that has a pending update
    offset = NVI_LEV_DISC_0+nv_array_index-FIRST_INPUT_NV;

SET_BIT(block_map[nv_plc_map[offset].block_num].update_flags,
                      nv_plc_map[offset].word_num);
    plc_update_pending = TRUE;
```

```
    if (state == IDLE)
    {   // alert other side to imminent message
        state = check_for_transmit();
    }

}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//              end SITE-SPECIFIC CODE:
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++


//
//  Now exiting the GUARD state, check
//  for a downlink message to send
//
when (timer_expires(guard_timer))
{
    if (state == GUARD)
    {
        state = check_for_transmit();
    }
}

//
//  Update message failed to get an
//  ACK or NACK, try to resend it
//
when (timer_expires(tx_ack_nack_timer))
{
    if (state == TX_DONE)
    {
        ++dbg_ack_nack_timeout;

        // This will force the update to be
        // reassembled, in case of new status
```

```
        curr_msg_byte = 0;
        state = check_for_transmit();
    }
}

//
//  Timed out while receiving a message or...
//  timing out of RX_NACK state.
//
when (timer_expires(data_receive_timer))
when (timer_expires(rx_nack_timer))
{
    if (state == RX_MSG)
    {   // inter-byte timeout
        ++dbg_receive_timeout;
    }

    // NACK to get other side to back off.
    slta_putchar(NACK);

    // Might get some more hiccups coming
    // uplink, allow some time for them.
    guard_timer = OUT_OF_SYNCH_TIMEOUT;
    state = GUARD;

}


//
//  Timed out while trying to send a message.  There
//  was no answer from the PLC to our message request.
//
when (timer_expires(alert_timer))
{
    if (state == TX_REQ)
```

```
    {
        ++dbg_alert_timeout;

        // try again to send it
        state = check_for_transmit();
    }
}


//
//  Timed out while waiting for the PLC to send
//  a RESYNCH_COMPLETE message.
//
when (timer_expires(resynch_timer))
{
    ++dbg_resynch_timeout;

    // try again to send it
    resynch_state = RESYNCH_REQUEST;

    if (state == IDLE)
    {
        state = check_for_transmit();
    }
}


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      XLATE.NC -- Example LONWORKS gateway to PLC.
//      Copyright (c) 1994 by Echelon Corporation.
//      All Rights Reserved.
//
//      Date last modified: 1/31/94
//
//  This file is intended to contain all of the gateway
```

```
//  code and data that is site-specific.  The one
//  exception to that is that there will be a WHEN
//  statement in PLCGATE.NC for every input Network
//  Variable.  Also, PLCGATE.H has a clearly delineated
//  section for site-specific definitions.
//
//  This file contains the translation tables needed to
//  convert PLC register data to Network Variable data and
//  vice versa.  The contents of the translation tables are
//  implementation-dependent, but the format of the tables
//  is intended to support a wide variety of configurations.
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
//
//  The Network Variable interface to this node consists
//  of 16 discrete level inputs and 16 discrete level
//  outputs.  Since each of these maps to one bit on the
//  PLC, only the ST_ON and ST_OFF levels are supported.
//  Any other level is translated to a "0" in the PLC
//  register bitfields.  The value translation is done
//  in the user-defined routines set_nv() and get_nv().
//
network  input SNVT_lev_disc NVI_lev_disc[NUM_INPUT_NVS];
network output SNVT_lev_disc NVO_lev_disc[NUM_OUTPUT_NVS];

//
// The plc_inputs and plc_outputs variables are
// copies of the latest PLC register values as
// we know them.  The pending flags for PLC output
// variables tell us that the new values have not
// yet been sent down the serial link to the PLC.
```

```
//
plc_word  plc_inputs[NUM_INPUT_WORDS];
plc_word  plc_outputs[NUM_OUTPUT_WORDS];
bits  plc_pending_updates[SIZEOF_BITMAP(NUM_OUTPUT_WORDS)];



// Translation tables

// First, the tables to take PLC inputs and convert
// them to Network Variable update values:
const field_nv  in_field_map[NUM_INPUT_FIELDS] = {
    FTYPE_BITS, 0, 1, NVO_LEV_DISC_0,
    FTYPE_BITS, 1, 1, NVO_LEV_DISC_1,
    FTYPE_BITS, 2, 1, NVO_LEV_DISC_2,
    FTYPE_BITS, 3, 1, NVO_LEV_DISC_3,
    FTYPE_BITS, 4, 1, NVO_LEV_DISC_4,
    FTYPE_BITS, 5, 1, NVO_LEV_DISC_5,
    FTYPE_BITS, 6, 1, NVO_LEV_DISC_6,
    FTYPE_BITS, 7, 1, NVO_LEV_DISC_7,
    FTYPE_BITS, 8, 1, NVO_LEV_DISC_8,
    FTYPE_BITS, 9, 1, NVO_LEV_DISC_9,
    FTYPE_BITS, 10, 1, NVO_LEV_DISC_10,
    FTYPE_BITS, 11, 1, NVO_LEV_DISC_11,
    FTYPE_BITS, 12, 1, NVO_LEV_DISC_12,
    FTYPE_BITS, 13, 1, NVO_LEV_DISC_13,
    FTYPE_BITS, 14, 1, NVO_LEV_DISC_14,
    FTYPE_BITS, 15, 1, NVO_LEV_DISC_15
};
const word_fields  in_word_map[NUM_INPUT_WORDS] =
{NUM_INPUT_FIELDS,
(const field_nv *)&in_field_map};

// Now, the tables to take Network Variable inputs
// and convert them to PLC output values:
```

```
const field_nv  out_field_map[NUM_OUTPUT_FIELDS] = {
    FTYPE_BITS, 0, 1, NVI_LEV_DISC_0,
    FTYPE_BITS, 1, 1, NVI_LEV_DISC_1,
    FTYPE_BITS, 2, 1, NVI_LEV_DISC_2,
    FTYPE_BITS, 3, 1, NVI_LEV_DISC_3,
    FTYPE_BITS, 4, 1, NVI_LEV_DISC_4,
    FTYPE_BITS, 5, 1, NVI_LEV_DISC_5,
    FTYPE_BITS, 6, 1, NVI_LEV_DISC_6,
    FTYPE_BITS, 7, 1, NVI_LEV_DISC_7,
    FTYPE_BITS, 8, 1, NVI_LEV_DISC_8,
    FTYPE_BITS, 9, 1, NVI_LEV_DISC_9,
    FTYPE_BITS, 10, 1, NVI_LEV_DISC_10,
    FTYPE_BITS, 11, 1, NVI_LEV_DISC_11,
    FTYPE_BITS, 12, 1, NVI_LEV_DISC_12,
    FTYPE_BITS, 13, 1, NVI_LEV_DISC_13,
    FTYPE_BITS, 14, 1, NVI_LEV_DISC_14,
    FTYPE_BITS, 15, 1, NVI_LEV_DISC_15
};
const word_fields  out_word_map[NUM_OUTPUT_WORDS] =
{NUM_OUTPUT_FIELDS,
(const field_nv *)&out_field_map};
const block_words  block_map[NUM_PLC_BLOCKS] = {
PLC_INPUT,  NUM_INPUT_WORDS, in_word_map, plc_inputs,  NULL,
PLC_OUTPUT, NUM_OUTPUT_WORDS, out_word_map, plc_outputs,
plc_pending_updates
};


//
//  The nv_plc_map is used when an NV update arrives
//  to set the associated PLC update pending flag.
//
const nv_to_plc  nv_plc_map[NUM_INPUT_NVS] =
{   1,  0,      // In this example, all input NVs are
    1,  0,      // in block 1, word 0
    1,  0,
```

```
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0,
    1,  0
};


//
//  PROCEDURE:  set_nv()
//  The following site-specific procedure should
//  have a case statement for every output NV in
//  the gateway that originates from a PLC word
//  value.  It takes a PLC field value input
//  and makes any transformations necessary to
//  update an NV based on the input.
//
void  set_nv (nv_ref_id  nv_id, plc_word  value)
{
    SNVT_lev_disc  discrete_value;

    switch (nv_id)
    {
        case NVO_LEV_DISC_0:
        case NVO_LEV_DISC_1:
        case NVO_LEV_DISC_2:
        case NVO_LEV_DISC_3:
```

```
case NVO_LEV_DISC_4:
case NVO_LEV_DISC_5:
case NVO_LEV_DISC_6:
case NVO_LEV_DISC_7:
case NVO_LEV_DISC_8:
case NVO_LEV_DISC_9:
case NVO_LEV_DISC_10:
case NVO_LEV_DISC_11:
case NVO_LEV_DISC_12:
case NVO_LEV_DISC_13:
case NVO_LEV_DISC_14:
case NVO_LEV_DISC_15:
    if (value == 1)
    {
        discrete_value = ST_ON;
    }
    else
    {   // fields are 1 bit wide, so
        // value may only be 0 or 1
        discrete_value = ST_OFF;
    }
    NVO_lev_disc[nv_id - NVO_LEV_DISC_0] =
            discrete_value;
    break;

// These will only be updated during a resynch.
// Since we know that a reset will change the
// value of these RAM variables to 0 (as we do
// not set them to any other value explicitly),
// we should only update the value if the
// current value is 0.  IF YOU STATICALLY
// INITIALIZE YOUR NVs, check for your own
// initialization value, not necessarily 0!
case NVI_LEV_DISC_0:
case NVI_LEV_DISC_1:
```

```
case NVI_LEV_DISC_2:
case NVI_LEV_DISC_3:
case NVI_LEV_DISC_4:
case NVI_LEV_DISC_5:
case NVI_LEV_DISC_6:
case NVI_LEV_DISC_7:
case NVI_LEV_DISC_8:
case NVI_LEV_DISC_9:
case NVI_LEV_DISC_10:
case NVI_LEV_DISC_11:
case NVI_LEV_DISC_12:
case NVI_LEV_DISC_13:
case NVI_LEV_DISC_14:
case NVI_LEV_DISC_15:
    if (value == 1)
    {
        discrete_value = ST_ON;
    }
    else
    {   // fields are 1 bit wide, so
        // value may only be 0 or 1
        discrete_value = ST_OFF;
    }

    // only set value if it has not already
    // been updated over the network
    if (NVI_lev_disc[nv_id - NVI_LEV_DISC_0] == 0)
    {
        NVI_lev_disc[nv_id - NVI_LEV_DISC_0] =
                discrete_value;
    }
    break;

default:
    break;
```

```
    }

}


//
//  PROCEDURE:  get_nv()
//  The following site-specific procedure should
//  have a case statement for every input NV in
//  the gateway that translates to a PLC output
//  word.  It takes an NV value and makes any
//  necessary transformations to produce a PLC
//  field value.
//
plc_word  get_nv (nv_ref_id  nv_id)
{
    plc_word  value;

    switch (nv_id)
    {
        case NVI_LEV_DISC_0:
        case NVI_LEV_DISC_1:
        case NVI_LEV_DISC_2:
        case NVI_LEV_DISC_3:
        case NVI_LEV_DISC_4:
        case NVI_LEV_DISC_5:
        case NVI_LEV_DISC_6:
        case NVI_LEV_DISC_7:
        case NVI_LEV_DISC_8:
        case NVI_LEV_DISC_9:
        case NVI_LEV_DISC_10:
        case NVI_LEV_DISC_11:
        case NVI_LEV_DISC_12:
        case NVI_LEV_DISC_13:
        case NVI_LEV_DISC_14:
```

```
        case NVI_LEV_DISC_15:
            value = NVI_lev_disc[nv_id - NVI_LEV_DISC_0];
            if (value == ST_ON)
            {
                value = 1;
            }
            else
            {   // reduce it to binary value
                // for the bitfield destination
                value = 0;
            }
            break;

        default:
            value = 0;
            break;

    }

    return (value);

}


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      RX_MSG.NC -- Example LONWORKS gateway to PLC.
//      Copyright (c) 1994 by Echelon Corporation.
//      All Rights Reserved.
//
//      Date last modified: 1/31/94
//
//  This file contains most of the gateway code dealing
//  with receiving messages from the PLC.  Routines
//  included here:
//
```

```
//      translate_plc_input()
//      comp_checksum()
//      checksum_checks()
//      process_word_format()
//      process_block_format()
//      process_uplink_msg()
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
//
//  PROCEDURE:  translate_plc_input()
//  This procedure translates PLC word data into "fields",
//  which are translated into LonTalk Standard Network
//  Variable Types.  Three types of fields are available:
//  16-bit words, 8-bit bytes, and bitfields from 1 to 15
//  bits wide.  This routine extracts fields from PLC words,
//  one at a time, and for each field calls the site-specific
//  procedure set_nv() to make the final transformation from
//  field data to Network Variable data.
//
void  translate_plc_input (int  block_num, int  word_num, plc_word
new_value)
{
    ulong         new_field_value,
                  curr_field_value, curr_value;
    int           i, num_flds;
    const word_fields  *word;
    const field_nv     *fld;

    if ((block_num >= NUM_PLC_BLOCKS) ||
        (word_num >= block_map[block_num].num_words))
    {   // no such block or no such word in block
```

```
        ++dbg_mapping_error;
    }
    else
    {
        fld = block_map[block_num].word[word_num].field;
        num_flds =
     block_map[block_num].word[word_num].num_fields;
        curr_value =
     block_map[block_num].word_value[word_num];
        for (i = 0; i < num_flds; i++)
        {
            switch (fld[i].field_type)
            {
                case FTYPE_CHAR:
                    if (i == 0)
                    {   // char is the lower byte
                        new_field_value = 0x00ff & new_value;
                        curr_field_value =
                                    0x00ff & curr_value;
                    }
                    else
                    {   // char is the upper byte
                        new_field_value =
                            (0xff00 & new_value) >> 8;
                        curr_field_value =
                            (0xff00 & curr_value) >> 8;
                    }
                    break;

                case FTYPE_LONG:
                    if (i != 0)
                    {   // only one long field will fit
                        ++dbg_invalid_ftype;
                    }
                    else
```

```
                {   // field takes up entire word
                    new_field_value = new_value;
                    curr_field_value = curr_value;
                }
                break;

            case FTYPE_BITS: // extract from bit field
                new_field_value = new_value <<
                (16 - fld[i].bit_size-fld[i].first_bit);
                new_field_value = new_field_value >>
                   (16 - fld[i].bit_size);
                curr_field_value = curr_value <<
                 (16 - fld[i].bit_size-fld[i].first_bit);
                curr_field_value = curr_field_value >>
                   (16 - fld[i].bit_size);
                break;

            default:
                ++dbg_invalid_ftype;
                break;

        }

        // Now, tmp_value contains the extracted
        // field value from the PLC register
        if ((new_field_value != curr_field_value) ||
            (resynch_state == RESYNCH_IN_PROGRESS))
        {
            set_nv (fld[i].nv_id, new_field_value);
        }

    }

    // Set the current word value to the updated value
    block_map[block_num].word_value[word_num] =
```

```
                        new_value;

    }

}


//
//  PROCEDURE:  comp_checksum()
//  Compute the checksum of a message just received.
//  This routine assumes that the message is waiting
//  in the message buffer.
//
unsigned char  comp_checksum (void)
{
    unsigned char  sum;
    int  i;

    // First, sum up the command and data portion of
    // the message in the buffer.
    sum = 0;
    for (i = MSG_CMD;
       i < (MSG_CMD + msg_buffer[MSG_LEN]); i++)
    {
        sum += msg_buffer[i];
    }

    // Checksum is the bitwise NOT of the sum
    sum = ~sum;

    return (sum);

}
```

```
//
//  PROCEDURE:  checksum_checks()
//  Does the checksum of the current message have
//  the value that we expect?
//
boolean  checksum_checks (void)
{
    unsigned char   sum;
    boolean         ret;

    // compute checksum of the input message
    sum = comp_checksum();

    // If the checksum is the same as the checksum received,
    // the message is validated.  If not, it fails.
    if (sum == msg_buffer[MSG_CMD + msg_buffer[MSG_LEN]])
    {
        ret = TRUE;
    }
    else
    {
        ret = FALSE;
    }

    return (ret);

}


//
//  PROCEDURE:  process_word_format()
//  Process a message in word update format.  This
//  format includes the length and ~length bytes,
//  followed by the WORD_UPDATE_MSG command byte
//  and the block number byte.  One or more word
```

```
//  updates follow this--word number byte, followed
//  by the two byte PLC word, least significant
//  byte first.  The message is ended with the
//  checksum byte.
//
unsigned char  process_word_format (void)
{
    unsigned char   ack_code;
    int    block_num,
           word_num,
           field_num,
           len, i;
    plc_word        data_word;

    // assume a good message format for now
    ack_code = ACK;

    // In word-format updates, the block number
    // occurs once, at the start of the message
    block_num = msg_buffer[MSG_CMD+1];
    len = msg_buffer[MSG_LEN];
    if ((block_num >= NUM_PLC_BLOCKS) ||
        (len > MAX_UPDATE_SIZE) ||
        ((len % 3) != 2))
    {   // invalid block number or message length
        ack_code = NACK;
    }
    else
    {   // process each update word
        for (i = 2; i < len;)
        {
            word_num = msg_buffer[MSG_CMD + i];
            ++i;
            data_word = (plc_word)msg_buffer[MSG_CMD + i];
            ++i;
```

```
            data_word +=
        ((plc_word)msg_buffer[MSG_CMD + i] << 8);
            ++i;
            translate_plc_input (block_num, word_num,
                        data_word);

            // save status in order to limit NV changes
            block_map[block_num].word_value[word_num] =
                        data_word;

        }

    }

    return (ack_code);
}


//
//  PROCEDURE:  process_block_format()
//  Process a message in word update format.  This
//  format includes the length and ~length bytes,
//  followed by the BLOCK_UPDATE_MSG command byte.
//  One or more block updates follow this--block
//  number byte, followed by the block length byte,
//  then all of the two byte PLC words in the
//  block, least significant byte first.  The
//  message is ended with the checksum byte.
//
unsigned char  process_block_format (void)
{
    unsigned char  ack_code;
    int    block_num, num_words,
           block_size, pos,
           last_pos, len, i;
```

```
    plc_word       data_word;

    ack_code = ACK;     // assume success for now

    // get first and last positions for message parsing
    last_pos = msg_buffer[MSG_LEN] + HEADER_SIZE - 1;
    pos = MSG_CMD + 1;
    while (pos < last_pos)
    {
        block_num = msg_buffer[pos++];
        num_words = msg_buffer[pos++];
        block_size = (int)(sizeof(plc_word) * num_words);
        if ((block_num >= NUM_PLC_BLOCKS) ||
            (num_words != block_map[block_num].num_words) ||
            ((pos + block_size - 1) > last_pos))
        {   // invalid message
            ack_code = NACK;
            break;
        }
        else
        {
            for (i = 0; i < block_size; i += 2)
            {
                data_word = (plc_word)msg_buffer[pos+i];
                data_word +=
          ((plc_word)msg_buffer[pos+i+1] << 8);
                translate_plc_input (block_num, i/2,
                       data_word);
            }

            // adjust position in message
            pos += block_size;

        }
```

```
    }

    return (ack_code);

}


//
//  PROCEDURE:  process_uplink_msg()
//  Process an incoming message from the PLC.
//  Call routines to check the checksum and
//  extract PLC update information.  Finally,
//  send an ACK if the message is in the
//  proper format and the checksum is good,
//  and send a NACK otherwise.
//
void  process_uplink_msg (void)
{
    unsigned char  ack_code;  // ack code back to PLC

    if (checksum_checks())
    {   // Message succeeds, process it and ACK it
        if (msg_buffer[MSG_CMD] == WORD_UPDATE_MSG)
        {
            ack_code = process_word_format();
        }
        else if (msg_buffer[MSG_CMD] == BLOCK_UPDATE_MSG)
        {
            ack_code = process_block_format();
        }
        else if (msg_buffer[MSG_CMD] == RESYNCH_COMPLETE)
        {   // no expected PLC resynch outstanding
            resynch_state = RESYNCH_NOT_NEEDED;
            resynch_timer = TIMER_OFF;
            guard_timer = TIMER_OFF;
```

```
            ack_code = ACK;
        }
        else
        {   // unknown command
            ack_code = NACK;
        }
    }
    else
    {   // checksum failed, NACK the message
        ack_code = NACK;
    }

    // Either ACK it or NACK it
    slta_putchar (ack_code);

}


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//      TX_MSG.NC -- Example LONWORKS gateway to PLC.
//      Copyright (c) 1994 by Echelon Corporation.
//      All Rights Reserved.
//
//      Date last modified: 1/31/94
//
//  This file contains most of the gateway code dealing
//  with transmitting messages to the PLC.  Routines
//  included here:
//
//      check_for_transmit()
//      translate_plc_output()
//      downlink_msg_success()
//      assemble_downlink_msg()
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
//
//  PROCEDURE:  check_for_transmit()
//  This routine is called when the serial link
//  becomes idle, to check for an outgoing message
//  from the Echelon gateway to the PLC.
//
link_state  check_for_transmit (void)
{
    link_state  ret_state;

    ret_state = IDLE;   // default: return to IDLE

    // If a downlink message is waiting and
    // the PLC is not currently sending to us,
    // start the process of sending an update

    if (resynch_state == RESYNCH_IN_PROGRESS)
    {   // getting initial status from PLC
        ret_state = GUARD;
    }
    else if (plc_update_pending ||
        (resynch_state == RESYNCH_REQUEST))
    {
        if (!slta_rxrdy())
        {   // alert other side to imminent message
            slta_putchar (ALERT);
            alert_timer = ALERT_TIMEOUT;
            ret_state = TX_REQ;
        }
    }

    return (ret_state);
```

```
}


//
//  PROCEDURE:  translate_plc_output()
//  Determine the value of a PLC output word
//  based on the current values of Network
//  Variables in the gateway.  This procedure
//  is analogous to translate_plc_input().
//
plc_word  translate_plc_output (int  block_num,
                  int  word_num)
{
    const field_nv  *fld;
    unsigned int     i, num_fields;
    plc_word         temp, value;

    // Prepare to cycle through all of the fields in
    // a word, updating each of them from the value
    // of the NV that drives it.
    num_fields =
     block_map[block_num].word[word_num].num_fields;
    fld = block_map[block_num].word[word_num].field;
    value = 0;
    for (i = 0; i < num_fields; i++)
    {
        temp = get_nv (fld[i].nv_id);
        switch (fld[i].field_type)
        {
            case FTYPE_LONG:   // no conversion needed
                break;

            case FTYPE_CHAR:
                if (i != 0)
```

```
                {   // only needs shifting if in upper byte
                    temp = temp << 8;
                }
                break;

            case FTYPE_BITS:
                temp = temp &
          ((-1) >> (16 - fld[i].bit_size));
                temp = temp << fld[i].first_bit;
                break;

            default:
                ++dbg_invalid_ftype;
                break;
        }

        // OR in each field value to assemble PLC word
        value |= temp;

    }

    // return PLC word value
    return (value);

}


//
//  PROCEDURE:  downlink_msg_success()
//  This routine is called when a downlink
//  message is successfully completed--an
//  ACK is received from the PLC.  It resets
//  the "update pending" flags for all of the
//  PLC register words that were sent downlink--
//  provided that their values have not changed
```

```
//  in the meantime.
//
link_state  downlink_msg_success (void)
{
    link_state    ret_state;
    unsigned int  block_num,
                  word_num,
                  len, i;

    if (msg_buffer[MSG_CMD] == REQ_RESYNCH)
    {   // no downlink traffic until resynch complete
        resynch_state = RESYNCH_IN_PROGRESS;
        guard_timer = RESYNCH_TIMEOUT + 50;
        resynch_timer = RESYNCH_TIMEOUT;
        ret_state = GUARD;
    }
    else
    {   // In word-format updates, the block number
        // occurs once, at the start of the message.
        // This example program does not support
        // block-format updates.
        block_num = msg_buffer[MSG_CMD+1];
        len = msg_buffer[MSG_LEN];
        for (i = 2; i < len; i += 3)
        {
            word_num = msg_buffer[MSG_CMD + i];
            if (block_map[block_num].word_value[word_num] ==
                  translate_plc_output(block_num, word_num))
            {   // if update sent is current, clear status
                CLEAR_BIT(block_map[block_num].update_flags,
                          word_num);
            }
        }

        // Update pending status by checking all status flags
```

```
        plc_update_pending = FALSE;
        for (i = 0;
        i<SIZEOF_BITMAP(block_map[block_num].num_words);
        i++)
        {
            if (block_map[block_num].update_flags[i] != 0)
            {
                plc_update_pending = TRUE;
                break;
            }
        }

        // go to idle state after this
        ret_state = IDLE;

    }

    return (ret_state);

}


//
//  PROCEDURE:  assemble_downlink_msg()
//  Assemble a message to send downlink.  The global
//  flag plc_update_pending tells whether there are
//  any PLC words to be updated, and we check it here
//  when looking at individual word status flags.
//
link_state  assemble_downlink_msg (void)
{
    link_state  ret;
    int         buffer_pos, i, j;

    ret = IDLE;     // go to IDLE state if no downlink msg
```

```
buffer_pos = MSG_CMD;
if (resynch_state == RESYNCH_REQUEST)
{   // send a resynch request
    msg_buffer[buffer_pos++] = REQ_RESYNCH;
    ret = TX_MSG;
}
else if (resynch_state == RESYNCH_IN_PROGRESS)
{   // not yet ready for downlink traffic
    ret = GUARD;
}
else
{   // Send a word-format update message downlink
    msg_buffer[buffer_pos++] = WORD_UPDATE_MSG;
    for (i = 0; i < NUM_PLC_BLOCKS; i++)
    {
        if (block_map[i].dir == PLC_OUTPUT)
        {   // Only word-format updates from the gateway
            // to the PLC are implemented here.
            for (j = 0; j < block_map[i].num_words; j++)
            {
                if (BIT_IS_SET(block_map[i].update_flags,
                        j))
                {
                   // block num
                   msg_buffer[buffer_pos++] = i;
                   // word num
                   msg_buffer[buffer_pos++] = j;
                   block_map[i].word_value[j] =
                   translate_plc_output(i, j);
                    msg_buffer[buffer_pos++] =
                      (unsigned char)
                        block_map[i].word_value[j];
                    msg_buffer[buffer_pos++] =
                        (unsigned char)
```

```
                        (block_map[i].word_value[j] >> 8);

                    ret = TX_MSG;   // update to send
                }

            }

        }

    }

    if (ret == IDLE)
    {   // there were no messages to be sent
        plc_update_pending = FALSE;
    }
    else if (ret == GUARD)
    {   // do nothing
        ;
    }
    else
    {   // finish formatting the message
        msg_buffer[MSG_LEN] = buffer_pos - MSG_CMD;
        msg_buffer[MSG_NOT_LEN] = ~msg_buffer[MSG_LEN];
        msg_buffer[buffer_pos] = comp_checksum();
    }

    return (ret);

}
```

Appendix B - A gateway to the Omron Sysmac CV500 PLC

The following code, written in a variant of the Basic programming language, runs on an Omron BSC21 Basic Unit. This unit is available with various serial port configurations. For the Gateway Demo, the Basic Unit was programmed through the first EIA-232 port via a Windows 3.1 terminal emulator. The second serial port was used for the gateway serial link. The PLC I/O points were polled every 100 ms, which is the most frequent clock tick available.

```
PNAME ""
NEW
PNAME "GATEWAY"
PGEN 1
AUTO 10, 10

PARACT 0 WORK 4096
DIM IN.LAST%    1
DIM STATE%
DIM IDLE.CT%
DIM IBYTE.CT%
DIM BYTE.NO%
DIM ACK.CODE%
DIM SUM%
DIM SYN.STATE%
DIM MSG.LEN%
DIM OUT.MSG$    12
DIM IN.CHAR$    1
DIM IN.BLK$     2
DIM OUT.BLK$    2
DIM SENT.BLK$   2

OPEN "COM2:19200,N,8,1,XN" AS #2
ON TIMER 1 GOSUB *CLOCK.TICK
ON COM(2) GOSUB *IO.IN
```

```
TIMER ON
COM(2) ON
WHILE 1
PAUSE
WEND
END

*CLOCK.TICK
IF (IBYTE.CT = 0) THEN *IDLE.TICK
IBYTE.CT = IBYTE.CT - 1
IF (IBYTE.CT <> 0) THEN *IDLE.TICK
STATE = 0
ACK.CODE = 21
COM(2) OFF
GOSUB *SEND.N.ACK
COM(2) ON
*IDLE.TICK
IF (IDLE.CT = 0) THEN *CHK.PLC.BLOCK
IDLE.CT = IDLE.CT - 1
IF (IDLE.CT <> 0) THEN *CHK.PLC.BLOCK
STATE = 0
IF (SYN.STATE = 2) THEN SYN.STATE = 1
IF (SYN.STATE <> 3) THEN *CHK.PLC.BLOCK
GOSUB *OUT.REQ
GOTO *EXIT.TICK
*CHK.PLC.BLOCK
IF (SYN.STATE = 1) THEN GOSUB *OUT.REQ
IF (SYN.STATE <> 0) THEN *EXIT.TICK
COM (2) OFF
PC READ "@R,1,1,1A3";IN.BLK$
COM (2) ON
IF ((STATE = 0) AND (CVI(IN.BLK$) <> IN.LAST)) THEN GOSUB *OUT.REQ
*EXIT.TICK
RETURN
```

```
*OUT.REQ
COM(2) OFF
IF (STATE <> 0) THEN *QUICK.OUT
OUTSTR$ = SPACE$(1)
MID$(OUTSTR$,1,1) = CHR$(1)
PRINT #2,OUTSTR$;
IDLE.CT = 4
STATE = 1
*QUICK.OUT
COM(2) ON
RETURN

*SEND.MSG
PC READ "@R,1,1,1A3";IN.BLK$
IF (SYN.STATE = 1) THEN MSG.LEN = 9 ELSE MSG.LEN = 5
OUT.MSG$ = SPACE$(MSG.LEN + 3)
MID$(OUT.MSG$,1,1) = CHR$(MSG.LEN)
MID$(OUT.MSG$,2,1) = CHR$((NOT MSG.LEN) AND 255)
MID$(OUT.MSG$,3,1) = CHR$(145)
MID$(OUT.MSG$,4,1) = CHR$(0)
MID$(OUT.MSG$,5,1) = CHR$(1)
MID$(OUT.MSG$,6,1) = MID$(IN.BLK$,2,1)
MID$(OUT.MSG$,7,1) = MID$(IN.BLK$,1,1)
SUM = 146
SUM = SUM + ASC(MID$(IN.BLK$,2,1))
SUM = SUM + ASC(MID$(IN.BLK$,1,1))
IF (SYN.STATE <> 1) THEN *WRAP.UP
PC READ "@R,0,1,1A3";OUT.BLK$
MID$(OUT.MSG$,8,1) = CHR$(1)
MID$(OUT.MSG$,9,1) = CHR$(1)
MID$(OUT.MSG$,10,1) = MID$(OUT.BLK$,2,1)
MID$(OUT.MSG$,11,1) = MID$(OUT.BLK$,1,1)
SUM = SUM + 2
SUM = SUM + ASC(MID$(OUT.BLK$,2,1))
SUM = SUM + ASC(MID$(OUT.BLK$,1,1))
```

```
*WRAP.UP
SUM = NOT SUM
MID$(OUT.MSG$,MSG.LEN+3,1) = CHR$(SUM AND 255)
PRINT #2,OUT.MSG$;
IDLE.CT = 9
IF (SYN.STATE = 1) THEN SYN.STATE = 2
SENT.BLK$ = IN.BLK$
STATE = 3
RETURN

*SEND.RC.MSG
OUT.MSG$ = SPACE$(4)
MID$(OUT.MSG$,1,1) = CHR$(1)
MID$(OUT.MSG$,2,1) = CHR$(254)
MID$(OUT.MSG$,3,1) = CHR$(177)
MID$(OUT.MSG$,4,1) = CHR$(78)
PRINT #2,OUT.MSG$;
IDLE.CT = 4
SYN.STATE = 3
STATE = 3
RETURN

*IO.IN
TIMER OFF
IF (LOC(2) = 0) THEN *IN.DONE
IN.CHAR$ = INPUT$(1,#2)
IF ((STATE = 0) AND (ASC(IN.CHAR$) = 1)) THEN *MSG.BEGIN
IF (STATE = 4) THEN *MSG.INPUT
IF (STATE = 3) THEN *RCV.ACK.NACK
IF (STATE = 1) THEN *RCV.ALERT.ACK ELSE *IN.DONE
*MSG.BEGIN
GOSUB *RX.BEGIN
GOTO *IN.DONE
*MSG.INPUT
GOSUB *RX
```

```
GOTO *IN.DONE
*RCV.ACK.NACK
IF (ASC(IN.CHAR$) <> 6) THEN *CHK.FOR.NACK
IF ((SYN.STATE = 0) OR (SYN.STATE = 2)) THEN IN.LAST =
CVI(SENT.BLK$)
IDLE.CT = 0
STATE = 0
IF (SYN.STATE = 3) THEN SYN.STATE = 0
IF (SYN.STATE = 2) THEN GOSUB *OUT.REQ
GOTO *IN.DONE
*CHK.FOR.NACK
IF (ASC(IN.CHAR$) <> 21) THEN *IN.DONE
IDLE.CT = 0
STATE = 0
IF (SYN.STATE = 2) THEN SYN.STATE = 1
IF (SYN.STATE = 3) THEN GOSUB *OUT.REQ
GOTO *IN.DONE
*RCV.ALERT.ACK
IF (ASC(IN.CHAR$) <> 254) THEN *IN.DONE
STATE = 2
IF ((SYN.STATE = 2) OR (SYN.STATE = 3)) THEN GOSUB *SEND.RC.MSG
IF ((SYN.STATE = 0) OR (SYN.STATE = 1)) THEN GOSUB *SEND.MSG
*IN.DONE
TIMER ON
RETURN

*SEND.N.ACK
OUT.MSG$ = CHR$(ACK.CODE)
PRINT #2,OUT.MSG$;
RETURN
*RX.BEGIN
OUT.MSG$ = CHR$(254)
PRINT #2,OUT.MSG$;
BYTE.NO = 0
OUT.BLK$ = SPACE$(2)
```

```
STATE = 4
RETURN

*RX
BYTE.NO = BYTE.NO + 1
IBYTE.CT = 2
IF (BYTE.NO = 1) THEN *GET.MSG.LEN
IF (BYTE.NO = 2) THEN *CHK.MSG.LEN
IF (BYTE.NO = 3) THEN *CHK.MSG.CMD
IF ((BYTE.NO = 4) AND (SYN.STATE = 1)) THEN *MSG.FINISH
IF (BYTE.NO = 4) THEN *CHK.MSG.BLK
IF (BYTE.NO = 5) THEN *CHK.MSG.WORD
IF (BYTE.NO = 6) THEN *GET.BLK.LOW
IF (BYTE.NO = 7) THEN *GET.BLK.HIGH
IF (BYTE.NO = 8) THEN *MSG.FINISH ELSE *RX.EXIT
*GET.MSG.LEN
MSG.LEN = ASC(IN.CHAR$)
GOTO *RX.EXIT
*CHK.MSG.LEN
IF (((NOT MSG.LEN) AND 255) = ASC(IN.CHAR$)) THEN *RX.EXIT ELSE
*FAILED.MSG
*CHK.MSG.CMD
SUM = ASC(IN.CHAR$)
IF (ASC(IN.CHAR$) = 129) THEN *RX.EXIT
IF (ASC(IN.CHAR$) = 161) THEN SYN.STATE = 1 ELSE *FAILED.MSG
GOTO *RX.EXIT
*CHK.MSG.BLK
SUM = SUM + 1
IF (ASC(IN.CHAR$) = 1) THEN *RX.EXIT ELSE *FAILED.MSG
*CHK.MSG.WORD
IF (ASC(IN.CHAR$) = 0) THEN *RX.EXIT ELSE *FAILED.MSG
*GET.BLK.LOW
SUM = SUM + ASC(IN.CHAR$)
MID$(OUT.BLK$,2,1) = IN.CHAR$
GOTO *RX.EXIT
```

```
*GET.BLK.HIGH
SUM = SUM + ASC(IN.CHAR$)
MID$(OUT.BLK$,1,1) = IN.CHAR$
GOTO *RX.EXIT
*MSG.FINISH
SUM = (NOT SUM) AND 255
IF (SUM = ASC(IN.CHAR$)) THEN ACK.CODE = 6 ELSE ACK.CODE = 21
IF ((ACK.CODE = 6) AND (SYN.STATE = 0)) THEN PC WRITE
"@R,0,1,1A3";OUT.BLK$
GOSUB *SEND.N.ACK
IBYTE.CT = 0
STATE = 0
GOTO *RX.EXIT
*FAILED.MSG
ACK.CODE = 21
GOSUB *SEND.N.ACK
IDLE.CT = 2
IBYTE.CT = 0
STATE = 5
*RX.EXIT
RETURN

END PARACT
```

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 005-0044-01 Rev. A

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