Form 163.2

Technical Notes for Older Legacy Systems

Enclosed are the following Technical Notes from OPTO 22. This will assist you in using OPTO 22 hardware and software with non-OPTO 22 products. If you have any questions, please call Technical Support at 714-695-9299 or 800-321-OPTO.

This technical note describes how to use the OPTO 22 analog modules on a custom mother board instead of an OPTO 22 Analog I/O Mounting Rack.

Digital to Analog (Output) Module Timing Requirements

All of the OPTO D/A modules are 12 bit serial devices. The D/A modules have two 5 VDC digital input lines, the STROBE input on pin 10 and the DATA input on pin 8. The 12 bits of serial data are loaded with the most significant bit (MSB) first. The data is positive true logic and clocked in on the low to high (positive edge) transition of the strobe. The DAC module will output the last 12 bits of data that are clocked in.

The driving device must be capable of sinking eight milliamperes while maintaining 0.4 volts or less at its output. Typical driving devices are Motorola 6821 (PIA), Intel 8255A (PPI), standard TTL, and LS gates. The inputs require negligible current while they are high.

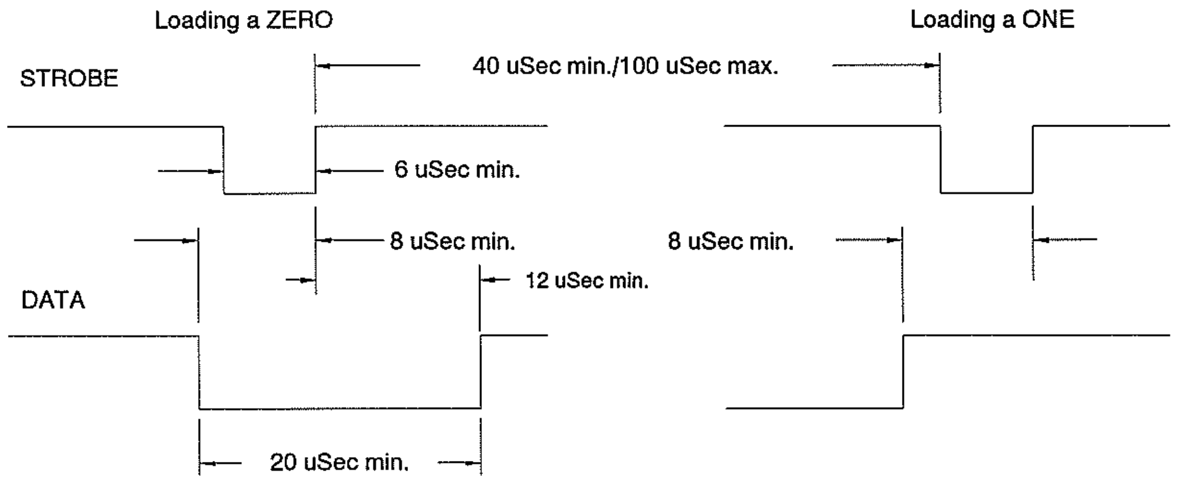
There are timing requirements that must be kept in mind when loading data into the D/A modules as follows:

STROBE:

The data is clocked in on the low to high transition of the strobe. The strobe must be low for six microseconds minimum before it can go high. After the strobe goes high it must remain high for 30 microseconds minimum and 100 microseconds maximum while clocking in the data. It will take 438 microseconds minimum to load all 12 bits of data.

DATA:

The data must be present eight microseconds minimum before the clock goes high and last 20 microseconds minimum.

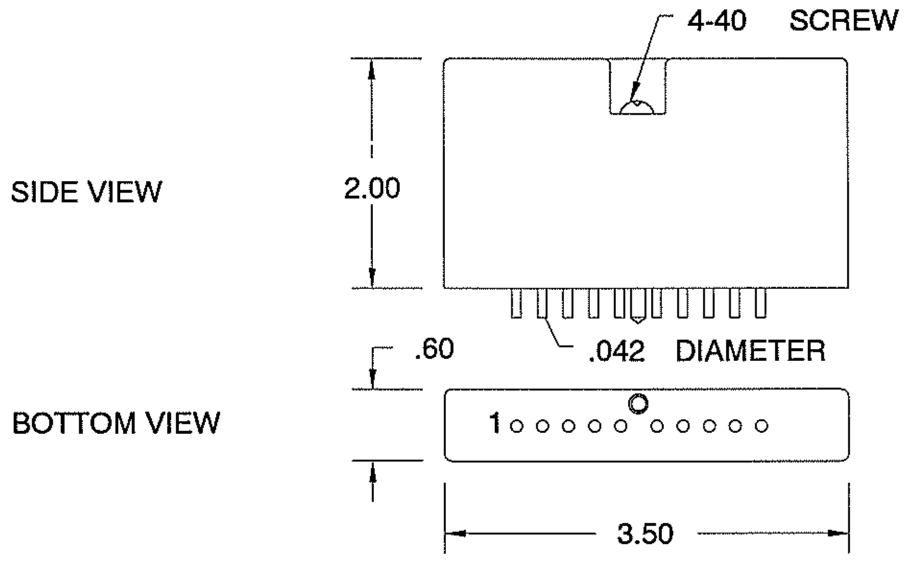


NOTE: Typical driving devices include Motorola 6821 (PIA), Intel 8255A (PPI), Standard TTL and LS Gates.

Analog Module Pin Out Description

The following table contains the descriptions of each of the pins on an analog module. Pin one is the pin closest to the terminal barrier strip of the analog I/O mounting rack. Each module may or may not use all of the pins.

PIN	DESCRIPTION
1	Upper B Barrier Strip
2	Upper A Barrier Strip
3	Analog Supply Common
4	- 15 VDC Analog Supply
5	+ 15 VDC Analog Supply
6	- 5.000 VDC Reference +/- 1 mV
7	+ 5 VDC Logic Supply
8	DATA Line
9	Logic Supply Ground
10	STROBE/CLOCK Line (D/A Modules only)



Analog To Digital (Input) Module Specifications

The A/D modules convert the field input value to a serial pulse train. The frequency range is from 1920 Hz (zero scale) to 9600 Hz (full scale). Each positive pulse has an on-time of 30 microseconds typical. The off-time equals the period minus the on-time. This pulse train appears on the DATA line located at pin 8 of the module. The pulse train is continuous and referenced to the Logic Supply Ground line at pin 9. The Analog Supply Common is separate from the Logic Supply Ground. All modules are optically isolated from field side to logic side. The "T" modules (AD3T, AD6T, etc.) are also transformer isolated, providing isolation between the analog supply and the analog input. Some modules have terminal screws on the top of the module, these screws should be used for field connections instead of, or in addition to, the terminal screws on the mounting rack.

BCD-TO-DECIMAL CONVERSIONS

Since the PB16J and PB16K input racks are ideal for interfacing to thumbwheel switches or encoder outputs, BCD-to-Decimal conversion routines are needed for proper interpretation of the data. This type of conversion can be done using the BASIC programming language.

If you are using the OPTOWARE driver to access an OPTOMUX brain board connected to a PB16J or PB16K rack, then you should use OPTOWARE driver command 64 to read the data. Command 64 is a Binary Read command which returns a 16-bit integer value (representing all the inputs on the rack), in the INFO%(0) variable.

Once that data is in the INFO%(0) variable, the following routine will convert that BCD value to a decimal value and place the result in the variable named VALUE%.

This routine assumes that the least significant bit is at position 0 and the most significant bit is at position 15. If the input is negative true logic, you will need to XOR INFO%(0) with a minus one (INFO%(0) = INFO%(0) XOR -1).

```
1000  '*------------------------------------------------------------*
1010  '
1020  '                 BCD TO DECIMAL CONVERSION
1030  '
1040  '                 INPUT: INFO%(0)   OUTPUT: VALUE%
1050  '
1060  '*------------------------------------------------------------*
1070  V1% = INFO%(0) AND 15                           'Convert first digit
1080  V2% = ((INFO%(0) AND 240) \ 16) * 10            'Convert second digit
1090  V3% = ((INFO%(0) AND 3840) \256) * 100          'Convert third digit
1100  V4% = ((INFO%(0) AND &HF000) \ 4096) * 1000     'Convert fourth digit
1110  IF V4% < 0 THEN V4% = V4% + 16000               'Correct upper bit
1120  VALUE% = V1% + V2% + V3% + V4%                  'Make it one value
1130  RETURN
```

DECIMAL-TO-BCD CONVERSIONS

The PB16L rack is ideal for writing to BCD devices such as 7-segment displays. Use the OPTOWARE command 65 to write the value to an OPTOMUX brain board connected to a PB16L output rack. This command will take a 16-bit integer value in the INFO%(0) variable and write it to all 16 positions. Position 0 corresponds to the least significant bit of the value in INFO%(0). Position 15 corresponds to the most significant bit of the 16-bit value in INFO%(0).

Since the BCD device expects the data in BCD format, the following routine converts a decimal value in variable VALUE% to the BCD equivalent and puts the result in the variable INFO%(0).

```
2000  '*-----------------------------------------------------------*
2010  '
2020  '                  DECIMAL TO BCD CONVERSION
2030  '
2040  '                  INPUT: VALUE%    OUTPUT: INFO%(0)
2050  '
2060  '*-----------------------------------------------------------*
2070  R% = VALUE% MOD 1000
2080  V1% = VALUE% \ 1000
2090  V2% = R% \ 100
2100  R% = R% MOD 100
2110  V3% = R% \ 10
2120  V4% = R% MOD 10
2130  INFO%(0) = V4% OR (V3%*16) OR (V2%*256) OR (V1%*4096)
2140  RETURN
```

NOTE: Depending on how the BCD devices are connected, you may need to invert the bits on outputs (Output &HFFFF to display 0000). This can be accomplished by using the NOT statement

INFO%(0) = NOT INFO%(0)).

CONFIGURING THE IBM PC SERIAL PORT

The IBM PC and many compatibles use an 8250 UART device for serial communications. The DOS operating system and many of the programming languages use calls to the BIOS to configure the serial port. However, the BIOS will only accept baud rates of 9600 or less. This is generally because reliable communications cannot be maintained by the inefficient BIOS routines for transmitting and receiving characters.

The OPTOWARE driver accesses the 8250 UART directly for transferring characters, therefore allowing higher baud rates to be used. Since the OPTOWARE driver does not do any initialization for the baud rate, the 8250 UART must be initialized prior to calling the driver.

The 8250 UART Registers

The 8250 UART device contains several registers which specify which baud rate and type of protocol to use. The program listings which follow this section use variables for the values to be written to the registers. These variables can be set to values which would provide the desired baud rates or protocols. The following tables specify the range of values that can be used.

PORT	BASE Variable	Interrupt Jumper
COM1	3F8 Hex	COM1 (IRQ4)
COM2	2F8 Hex	COM2 (IRQ3)
COM3	348 Hex	IRQ2
COM4	340 Hex	IRQ5

BAUDRATE	DLH	DLL Variables
38400	00	03
19200	00	06
9600	00	0C Hex
4800	00	18 Hex
2400	00	30 Hex
1200	00	60 Hex
300	01	80 Hex

	PROTOCOL		
Data	Parity	Stop	DLAB Variable
8	NONE	1	03
8	EVEN	1	1B Hex
8	ODD	1	0B Hex
8	NONE	2	07
8	EVEN	2	1F Hex
8	ODD	2	0F Hex
7	NONE	1	02
7	EVEN	1	1A Hex
7	ODD	1	0A Hex
7	NONE	2	06 Hex
7	EVEN	2	1E Hex
7	ODD	2	0E Hex

To set the baud rate on the 8250, a value of 80 Hex must first be written to the register at location BASE+3. The DLL value can then be written to the register at location BASE, and the DLH value can be written to the register at location BASE+1. Once these values have been written, the DLAB value can be written to the register at location BASE+3. Finally, a value of 2 is written to the register at location BASE+4.

For more detailed information on the registers of the 8250 UART, please refer to the IBM PC Technical Reference Manual #6025005.

NOTE: The IBM AT, PS/2, and many non-PC/XT compatibles use a different UART (16450 or 16550), however, they are functionally similar for this application.

Program Examples:

Turbo PASCAL Example

```
Procedure SetPort;
const
        BASE = $3F8;                {COM1 Address}
        DLL = 6;                    {19.2k Baud }
        DLH = 0;
        DLAB = 3;                   {8 Data, No Parity, 1 Stop Bit}

begin
        Port[BASE + 3] := $80;      {Write Parameters To Registers}
        Port[BASE] := DLL;
        Port[BASE + 1] := DLH;
        Port[BASE + 3] := DLAB;
        Port[BASE + 4] := 2;
end;
```

BASIC Example

```
BASEAD% = &H3F8              'Base Address of COM1
DLL% = 6                     '19.2k Baud
DLH% = 0
DLAB% = 3                    '8 Data, No Parity, 1 Stop Bit For OPTOMUX
OUT BASEAD%+3,&H80           'Sets Up Control Register 8250
OUT BASEAD%,DLL%             'Write Parameters To Registers
OUT BASEAD%+1,DLH%
OUT BASEAD%+3,DLAB%
OUT BASEAD%+4,2              'Sets RTS To Always Be Low
```

Turbo C Example

```
#include <stdio.h>
#include <dos.h>

#define BASE 0x2F8           /* COM2 Address */
#define DLL 0x03             /* 38.4k Baud */
#define DLH 0x00
#define DLAB 0x03            /* 8 Data, No Parity, 1 Stop Bit */

void setport ( )
{
        outportb(BASE+3,0x80);
        outportb(BASE,DLL);
        outportb(BASE+1,DLH);
        outportb(BASE+3,DLAB);
        outportb(BASE+4,2);
}
```

Microsoft C Example

```
#include <stdio.h>
#include <conio.h>

#define BASE 0x2F8           /* COM2 Address */
#define DLL 0x03             /* 38.4k Baud */
#define DLH 0x00
#define DLAB 0x03            /* 8 Data, No Parity, 1 Stop Bit */

setport ( )
{
        outp(BASE+3,0x80);
        outp(BASE,DLL);
        outp(BASE+1,DLH);
        outp(BASE+3,DLAB);
        outp(BASE+4,2);
}
```

USING THE OPTOWARE DRIVER WITH Borland's Turbo BASIC

The OPTOWARE Driver can be used with Turbo BASIC with the following modification.

Turbo BASIC arrays are stored in their own segments, which are different than the data segment where integer variables are stored. The driver expects all variables to reside in the segment pointed at by the 8088/80286 DS register. Therefore, to use the driver, you must define all array elements as integer variables stored consecutively.

The following excerpt from a Turbo BASIC program demonstrates how to define all driver parameters and how to call the driver.

```
'*
'*      DIMENSION AND INITIALIZE DRIVER PARAMETERS
'*
POSITION15% = 0         '* Allocate Arrays As Integer Scalars
POSITION14% = 0         '* Instead Of An Integer Array
POSITION13% = 0         '* Order Is Important Here
POSITION12% = 0
POSITION11% = 0
POSITION10% = 0
POSITION9% = 0
POSITION8% = 0
POSITION7% = 0
POSITION6% = 0
POSITION5% = 0
POSITION4% = 0
POSITION3% = 0
POSITION2% = 0
POSITION1% = 0
POSITION0% = 0
MODIFIER1% = 0
MODIFIER0% = 0
INFO15% = 0
INFO14% = 0
INFO13% = 0
INFO12% = 0
INFO11% = 0
INFO10% = 0
INFO9% = 0
INFO8% = 0
INFO7% = 0
INFO6% = 0
INFO5% = 0
INFO4% = 0
INFO3% = 0
INFO2% = 0
INFO1% = 0
INFO0% = 0


ERRCOD% = 0
ADDRESS% = 0
CMD% = 0
'*
'*      LOAD THE DRIVER SUBROUTINE
'*
DEF SEG = &H6000                '* Define Segment For Driver
BLOAD "DRIVER.COM",0  '* Load The Driver
OPTOWARE% = 0                   '* Use "OPTOWARE%" To Call The Driver
```

```
'***************************************************************************
'*
'*              MAIN ROUTINE
'*
'***************************************************************************

GOSUB SetSerial                 '* Initialize The Serial Port
GOSUB puc                       '* Send puc To Board 255
GOSUB readstat          '* Read Status Of All Inputs
END
'***************************************************************************
'*
'*              DRIVER CALLING SUBROUTINE
'*
'***************************************************************************
CallDriver:
CALL  ABSOLUTE  OPTOWARE%(ERRCOD%,ADDRESS%,CMD%,POSITION0%,
        MODIFIER0%,INFO0%)
IF ERRCOD% = 0 THEN RETURN
'*
'*              ERROR HANDLING CODE
'*
PRINT
PRINT "Error: ";ERRCOD%;"  Address: ";ADDRESS%;"  Command: ";CMD%
PRINT
RETURN
'***************************************************************************
'*
'*              SEND POWER UP CLEAR COMMAND TO ADDRESS 255
'*
'***************************************************************************
puc:
CMD% = 0                '* Power Up Clear Command
ADDRESS% = 255
GOSUB CallDriver        '* Call The Driver
RETURN
'***************************************************************************
'*
'*              SEND READ STATUS COMMAND TO ADDRESS 255
'*
'***************************************************************************
readstat:
CMD% = 12               '* Read Status Of All Inputs
ADDRESS% = 255
GOSUB CallDriver        '* Call The Driver
RETURN
'***************************************************************************
'*
'*              SETUP SERIAL PORT SUBROUTINTE
'*
'***************************************************************************
SetSerial:
OUT &H3FB,&H80          '* Setup UART At COM1 Address
OUT &H3F8,6             '* For 19.2k Baud
OUT &H3F9,0
OUT &H3FB,3
OUT &H3FC,2
CMD%=101                '* Set Driver Turn Around Delay
INFO0%=10               '* To 0.10 Second
GOSUB CallDriver
RETURN
'*
.
.
.
```

USING THE OPTOWARE DRIVER WITH Borland's Turbo C

The OPTOWARE driver can be called from Borland's Turbo C, by using the following statements in your program.

```
#include <stdio.h>
#include <ctype.h>

/*******************************************************************
*
*
*       The following statements are the variable and array
*       definitions needed to call the driver.  All parameters are
*       declared as global.
*
*
*               Define Driver Parameters
*/

int near        errors,             /* OPTOWARE Driver Error Status */
                address, /* OPTOMUX Board Address Range 0 - 255 */
                command,            /* OPTOMUX command - See OPTOWARE Manual */
                positions[ 16 ],    /* Module Positions Table - See Manual: POSITIONS Array */
                modifiers[ 2 ],     /* Modifier Table - See Manual: MODIFIERS Array */
                info[ 16 ];         /* Info Table - See Manual: INFO Array */


/*-------------------------------------------------------------------
*
*       Declare driver module as a far pascal call, no return values,
*       and parameter list is passed to driver on the stack.
*
*       void - This keyword specifies no return values.
*
*       far - Aligns all memory models.
*
*       pascal - This keyword causes arguments to be pushed on the
*               stack from left to right (last argument is last pushed).
*
*       int - Integer variable.
*
*       near - Passes only the offset address, not the segment.
*
*       *p1, *p2, *p3, ..., *p6 - Declares pointers as 16 bits.
*/
void far pascal optoware (int near *p1, int near *p2, int near *p3, int near *p4,
        int near *p5, int near *p6);
```

Below is a sample main program which uses the declarations made earlier. The example sets the driver to use COM port 2, then sends a Power Up Clear command to the OPTOMUX unit at address 255. This example assumes that COM port 2 has been previously initialized to the proper baud rate.

```
main ( )
{
        errors = 0;                 /* Initialize Errors Variable To 0 */
        address= 0;                 /* Initialize Address Variable To A Value */
        command = 102;   /* Select Com Port */
        info[ 0 ] = 2;              /* Selects Port 2 */
        optoware (&errors, &address, &command, positions, modifiers, info);
        .
        .
        .
        printf ("\nthe return error is: %d\n", errors);
        .
        .
        .
        command = 0;                /* Power Up Clear Command */
        address = 255;              /* Address Of OPTOMUX Board */
        optoware (&errors, &address, &command, positions, modifiers, info);
        .
        .
        .
        printf ("\nthe return error is: %d\n", errors);

}
```

Make sure that you link either the DRIVER.OBJ or IDRIVER.OBJ file with your program. This can be easily done by declaring the path and file name in the project file, so that Turbo C can find the driver when compiling.

USING The OPTOWARE DRIVER With Borland's Turbo Pascal 3.0

To interface to the OPTOMUX network using Turbo Pascal version 3, you must use the OPTOWARE driver which is in the Pascal format (Include file). The Turbo Pascal Include file is called TURBO30.INC (or IDRIVER.INC on previous OPTOWARE Source disk). The SETPORT.INC file is also included on the OPTOWARE Source disk; it contains the procedure for initializing the serial port.

These two INCLUDE files must be specified in your program using the Turbo Pascal Include Directive statement:

```
{$I turbo30.inc}
{$I setport.inc}
```

These two statements must be placed before the procedures they contain are referenced (typically, at the beginning).

To call the OPTOWARE driver, the PrepOptowareDriver procedure must be called once at the beginning of the program to initialize the driver and variables. After calling that procedure, you can call the OPTOWARE driver as follows:

```
Opto(Errors, Address, Command, Positions, Modifiers, Info);
```

A typical Turbo Pascal 3.0 program would contain the following statements:

```
Program ProgName;

{$I turbo30.inc}                        {or idriver.inc}
{$I setport.inc}
.
.
Begin
      PrepOptowareDriver;
      SetSerial (2,19200);              {Sets Port 2 to 19.2 K baud}
      .
      .
      Errors := 0;
      Address := 0;
      Command := 102;           {Sets The Driver To Port 2}
      Info [0] := 2;
      Opto(Errors, Address, Command, Positions, Modifiers, Info);
      .
      .
      .
      Command := 0;
      Opto(Errors, Address, Command, Positions, Modifiers, Info);
      .
      .
      .
End.
```

Version 4 of the OPTOWARE driver contains a new command, Configure Serial Port (command 104), which configures the serial port to a given baud rate. It also sets it to eight data bits, no parity, and one stop bit. Command 102, Set Serial Port Number, and command 104 can replace the Include Directive {$I Setport.inc} and the procedure SetSerial (2,19200) statements. Therefore, the example would be as follows:

```
Program ProgName;

{$I turbo30.inc}                          {or idriver.inc}
.
.
Begin
      PrepOptowareDriver;
      .
      .
      Errors := 0;
      Address := 0;
      Command := 102;          {Set The Driver To Port 2}
      Info [0] := 2;
      Opto(Errors, Address, Command, Positions, Modifiers, Info);
      Command := 104;          {Configure The Serial Port To 19200 Baud}
      Info [0] := 19200;
      Opto(Errors, Address, Command, Positions, Modifiers, Info);
      .
      .
End.
```

NOTE: Remember to check for errors after calling the OPTO procedure.

USING The OPTOWARE DRIVER With Borland's Turbo Pascal 4.0

Turbo Pascal version 4 does not use INCLUDE files to interface to the OPTOMUX network . It links directly to the OBJECT file contained on the OPTOWARE Source disk. The Turbo Pascal Link Directives are:

```
{$L turbo40}
{$L driver}
```

The TURBO40 file is the OBJECT file that interfaces between Turbo Pascal version 4 and the OPTOWARE driver. After the Link Directive statements, you must define the type of variables, the variables, and the procedures. The following example shows the necessary statements required:

```
Program ProgName;

{$L turbo40}
{$L driver}
.
.
Type
        Range = Array [0..15] of Integer;

Var
        Errors, Address, Command: Integer;
        Positions, Modifiers, Info: Range;

Procedure Opto(Var A, B, C: Integer; Var D, E, F:Range); External;
Procedure Optoware; External;
.
.
Begin
        Errors := 0;
        Address := 0;
        Command := 102;          {Set The Driver To Port 2}
        Info [0] := 2;
        Opto(Errors, Address, Command, Positions, Modifiers, Info);
        Command := 104;          {Configure The Serial Port To 19200 Baud}
        Info [0] := 19200;
        Opto(Errors, Address, Command, Positions, Modifiers, Info);
        .
        .
End.
```

NOTE: Remember to check for errors after calling the OPTO procedure.

To compile the Turbo Pascal version 4 program on the command line, using the Turbo Pascal Compiler, type:

```
TPC filename
```

Using The Optoware Drivers With Microsoft's QuickBASIC

If you are switching from interpretive BASIC to compiled QuickBASIC, be sure to delete the three statements that loads the OPTOWARE driver into memory. The three statements are:

```
DEF SEG = &H3300
BLOAD "DRIVER.COM",0
OPTOWARE = 0
```

Using The OPTOWARE Driver With Microsoft's QuickBASIC 2.0

Microsoft's QuickBASIC 2.0 currently works with OPTOWARE Version 3.01 and above with no modifications.

Using The OPTOWARE Driver With Microsoft's QuickBASIC 3.0

Treat the call to the OPTOWARE driver the same as when using the IBM BASIC compiler. However, some users have experienced problems when placing the CALL OPTOWARE statement in subprograms. Therefore, the CALL OPTOWARE statement should remain in the main program. The CALL OPTOWARE statement can reside as a subroutine in the main program and be accessed by calls from subprograms.

Compile the program from the command line rather than from the QuickBASIC environment. The command line should look like this:

```
QB filename.ext /D/O/V/E;
```

This will generate an OBJ file that does not require the runtime library, allows all interrupt driven events, and also perform error checking. The / options may change depending on the program requirements. The ; allows the compiler to use the default settings when compiling.

To link the driver use the following command:

```
LINK filename.ext + DRIVER + GWCOM;
        or
LINK filename.ext + IDRIVER + GWCOM;
```

The DRIVER.OBJ file is the standard polled version of the driver, the IDRIVER.OBJ file is the interrupt version of the driver. The polled version of the driver will work at baud rates from 300 to 19200 with no problems except that if you are doing a lot of communications, the PC's timer may lose clock ticks. This happens because the polled version of the driver must disable all system interrupts while it is waiting for a response from OPTOMUX. The interrupt version of the driver also waits in a loop for the response but allows all system interrupts to continue, so no clock ticks are missed.

When using QuickBASIC ON KEY and ON TIMER statements with the interrupt version of the driver (IDRIVER) communicating with OPTOMUX at 19200 baud, there is a possibility that characters may be dropped during communications, resulting in checksum errors (error code = -31). The solution is to use a slower baud rate.

Using The OPTOWARE Driver With Microsoft's QuickBASIC 4.0

Application programs written to be compiled using Microsoft's QuickBASIC 4.0 must be modified. Three additional statements must be added, two at the beginning of the program and one after DIMensioning the OPTOWARE array variables and the CALL OPTOWARE statement must be modified. The three additional statements are:

```
REM $STATIC
DECLARE SUB OPTOWARE (A%, B%, C%, BYVAL D%, BYVAL E%, BYVAL F%)
DIM Statements For The OPTOWARE Driver Array Variables
        .
        .
COMMON SHARED ERRCOD%, ADDR%, CMD%, POSI%( ), MODI%( ), INFO%( )
        .
        .
```

The CALL OPTOWARE statement must be modified to:

```
CALL OPTOWARE(ERRCOD%, ADDR%, CMD%, VARPTR(POSI%(0)),
              VARPTR(MODI%(0)), VARPTR(INFO%(0)))
```

Please note that integer variables in the DECLARE SUB OPTOWARE statement are dummy variables. They are not used in the remainder of the program. However, the COMMON SHARED statement must contain the integer variables which are referenced in the program. The COMMON SHARED statement must also precede any executable statements. Please refer to Microsoft's "Reference Manual" regarding these commands and statements.

Compile and link on the command line as follows after the application program has been modified:

BC *filename.ext* /D/O/V/E;

LINK *filename.ext* + IDRIVER;

Please refer to Chapter 8 in Microsoft's "Learning And Using Microsoft QuickBASIC" manual on how to create Quick Library (QLB) and Library (LIB) files. This is only required to run the program in the QuickBASIC environment and create an executable file from the QuickBASIC environment.

USING The OPTOWARE DRIVER With Microsoft's C 4.0/5.0

The OPTOWARE driver can be called from Microsoft C Version 4.0 and 5.0, by using the following statements in your program. The program example uses the small memory model.

```
#include <stdio.h>
#include <ctype.h>

/*****************************************************************
*
*
*       The following statements are the variable and array
*       definitions needed to call the driver. All parameters are
*       declared as global.
*
*
*               Define Driver Parameters
*/

int near        errors,         /* OPTOWARE Driver Error Status */
                address,        /* OPTOMUX Board Address Range 0 - 255 */
                command,        /* OPTOMUX Command - See OPTOWARE Manual */
                positions[ 16 ], /* Module Positions Table - See Manual: POSITIONS Array */
                modifiers[ 2 ], /* Modifiers Table - See Manual: MODIFIERS Array */
                info[ 16 ];     /* Info Table - See Manual: INFO Array */


/*-----------------------------------------------------------------
*
*       Declare driver module as a far pascal call, no return values,
*       and parameter list is passed to driver on the stack.
*
*       void - This keyword specifies no return values.
*
*       far - Aligns all memory models.
*
*       pascal - This keyword causes arguments to be pushed on the
*               stack from left to right (last argument is last pushed).
*
*       int - Integer variable.
*
*       near* - Declares pointers as 16 bits.
*
*/

void far pascal optoware (int near*, int near*, int near*, int near*, int near*, int near*);
```

Below is a sample main program which uses the declarations made earlier. The example sets the driver to use COM port 2, then sends a Power Up Clear command to the OPTOMUX unit at address 255. This example assumes that COM port 2 has been previously initialized to the proper baud rate.

```
main ( )
{
        errors = 0;                    /* Initialize Errors Variable To 0 */
        address= 0;                    /* Initialize Address Variable To A Value */
        command = 102;                 /* Select Com Port */
        info[ 0 ] = 2;                 /* Selects Port 2 */
        optoware (&errors, &address, &command, positions, modifiers, info);
        .
        .
        .
        printf ("\nthe return error is: %d\n", errors);
        .
        .
        .
        command = 0;                   /* Power Up Clear Command */
        address = 255;                 /* Address Of OPTOMUX Board */
        optoware (&errors, &address, &command, positions, modifiers, info);
        .
        .
        .
        printf ("\nthe return error is: %d\n", errors);
}
```

Make sure that you link either the DRIVER.OBJ or IDRIVER.OBJ file with your program.

ASSEMBLY LANGUAGE PROGRAMMING For The LC2

BASIC is an easy and fairly friendly language to use. One of its main problems is that it is slow. Machine language subroutines can be used to increase the performance of your BASIC code.

HARDWARE

Before you can competently write machine language routines for the LC2, you have to be familiar with the hardware. The main hardware components of the LC2 are:

PROCESSOR:

The processor used on the LC2 is the Zilog Z80. The Z80 is an 8 bit processor running at 4.9 MHz.

SERIAL PORTS:

The two serial ports on the LC2 are derived from the Zilog DART. The DART is a two channel serial port chip. It is located in the I/O space at addresses 0 through 3. COM0 uses channel A of the DART and COM1 uses channel B of the DART.

Each serial port is able to tri-state it's RS422/485 driver chips (allowing multidropping of the LC2s). The RTS (Request To Send) line is used to enable and disable the driver chips. Each port uses it's own RTS line to control it's driver. Setting the RTS bit high in the DART will enable the driver chip, setting the RTS bit low in the DART will disable the driver chip.

The RI input (Ring Indicator) on channel B is connected to the interrupt output of the RTC (real time clock). Every 0.1 seconds an interrupt is generated. This interrupt is used for the ON TIMER statement in the BASIC.

WATCHDOG TIMER:

The Watchdog Timer output is the DTR (Data Terminal Ready) output on channel A of the DART. This line must be toggled at least once every 100 ms, otherwise, your LC2 will reset. The banging of the Watchdog Timer is normally handled by BASIC.

REAL TIME CLOCK:

The Real Time Clock chip is the National MM58274. It is addressed from 10 Hex to 1F Hex.

ROM:

The LC2 has 32K of ROM addressed from 0 to 7FFF Hex.

RAM:

There is 32K of battery backed RAM addressed from 8000 Hex to FFFF Hex.

SOFTWARE

There are two BASIC statements that are used when accessing your own machine language routines, the CLEAR statement and the CALL statement.

CLEAR STATEMENT

The clear statement is used for allocating space for your machine language routines. The CLEAR will move the start of your BASIC program the specified number of bytes away from the start of RAM (8000 Hex). Your BASIC program will not be trash by this command. The following examples reserves 150 bytes of space for you machine language routine:

```
100    CLEAR 150        'Clear Space For Subroutine
```

NOTE: After clearing some RAM space, you should not do a NEW statement. A NEW will deallocate the RAM you have cleared. To remove the existing BASIC program without deallocating the reserved RAM, do a DELETE command. The following example will remove the existing BASIC program without deallocating the reserved RAM:

```
DELETE                  'Delete With No Numbers Means Delete All
```

CALL STATEMENT

Machine language subroutines are accessed via the CALL statement. Using the CALL statement, you can pass variables to the machine language subroutine. The CALL passes pointers to the variable data, not the data itself.

The possible data types that can be passed are integer variables, floating point variables and string variables. Each variable type uses different amounts of storage, so your subroutine must know what data type(s) it is manipulating.

DATA TYPES

INTEGER VARIABLES

Integer variables take up two contiguous bytes of storage. The low byte of the data is in low memory and the high byte of data is in high memory.

FLOATING POINT VARIABLES

Floating point variables take up two contiguous bytes of storage. The low byte of the mantissa is store in low memory, the middle byte is next, the high byte of the mantissa is next and the exponent is last.

STRINGS

Strings are anywhere from 0 to 255 bytes long. Each string is terminated with a 03 Hex. Do NOT change the length of a string in your machine language subroutine, you will only confuse the BASIC interpreter.

ARRAYS

The data in an array is stored contiguously, array element 0 is followed by array element 1, 1 is followed by 2, etc.

ON THE STACK

The data that is given to a machine language routine is passed on the processor stack. A pointer to each item being passed is pushed on the stack before the subroutine is called. The last item being passed is actually the first item pushed. Examine the following example:

```
CALL TEST (MU, OMEGA1, OMEGA2)
```

After all the parameters have been pushed onto the stack, a processor call will be performed. This will push the return address onto the stack. The stack should look as follows to your machine language routine:

```
high memory   -     pointer to OMEGA2
                    pointer to OMEGA1
                    pointer to MU
low memory    -     return address
```

The stack pointer will be pointing to the low byte of the return address.

WARNING - WARNING - WARNING

If your machine language routine is going to take a while, you must bang the watchdog timer circuitry. If you don't, the LC2 will be reset. The alternate register set on the Z80 has been setup by the BASIC for the bang. The following code segement is the code required for banging the watchdog timer:

```
DI              ;stop the interrupts for a while
EXX             ;get the other register set
OUT (C), B      ;select correct DART register
OUT (C), D      ;turn on watchdog bang bit
OUT (C), B      ;select correct DART register
OUT (C), E      ;turn off watchdog bang bit
EXX             ;get original register set
EI              ;restart the interrupts
```

EXAMPLE

The following example clears some memory, loads the machine language routine into the cleared RAM and then executes the machine language routine.

```
1000  '
1010  '
1020  '          Clear Some Memory For A Machine Language Subroutine
1030  '
1040  CLEAR 100
1050  '
1060  '          Now Poke Exchange Routine Into Cleared RAM
1070  '
1080  FOR X% = &H8000 TO &H800F
1090  READ P%                       'Get Program Byte
1100  POKE X%, P%                   'Stick Program Byte Into RAM
1110  NEXT
1120  '
1130  '
1140  EXCHANGE% = &H8000            'Define Address Of Subroutine
1150  '
1160  '          Set Up Some Variable To Pass To Routine
1170  '
1180  ALPHA% = 12345
1190  BETA% = 23456
1200  '
1210  '          Print The Variables
1220  '
1230  PRINT ALPHA%, BETA%
1240  '
1250  '          Now Do Call To Exchange The Variables
1260  '
1270  CALL EXCHANGE% (ALPHA%, BETA%)          'Do Call
1280  '
1290  '          Now Print The Variables Again
1300  '
1310  PRINT ALPHA%, BETA%
1320  '
1330  '
1340  '          This Is The Machine Language Subroutine
1350  '
1360  '                    POP HL         ;get return address from stack
1370  DATA &HE1
1380  '                    POP DE         ;get pointer to 1st parameter
1390  DATA &HD1
1400  '                    EX (SP), HL    ;exchange return with 2nd parameter
```

```
1410  DATA &HE3
1420  '              LD A, (DE)     ;get low byte of 1st parameter
1430  DATA &H1A
1440  '              LD C, (HL)     ;get low byte of 2nd parameter
1450  DATA &H4E
1460  '              LD (HL), A     ;and put 1st into 2nd variable
1470  DATA &H77
1480  '              LD A, C        ;get low byte of 2nd parameter
1490  DATA &H79
1500  '              LD (DE), A     ;and put 2nd into 1st variable
1510  DATA &H12
1520  '              INC HL
1530  DATA &H23
1540  '              INC DE
1550  DATA &H131560  '              LD A, (DE)     ;get high byte of 1st parameter
1570  DATA &H1A
1580  '              LD C, (HL)     ;get high byte of 2nd parameter
1590  DATA &H4E
1600  '              LD (HL), A     ;and put 1st into 2nd variable
1610  DATA &H77
1620  '               LD A, C       ;get high byte of 2nd parameter
1630  DATA &H79
1640  '              LD (DE), A     ;and put 2nd into 1st variable
1650  DATA &H12
1660  '              RET
1670  DATA &HC9
1680  '
```

ASSEMBLY LANGUAGE PROGRAMMING For The LC4

BASIC is an easy and fairly friendly language to use. One of its main problems is that it is slow. Machine language subroutines can be used to increase the performance of your BASIC code.

HARDWARE

Before you can competently write machine language routines for the LC4, you have to be familiar with the hardware. The main hardware components of the LC4 are:

PROCESSOR:

The processor used on the LC4 is the Hitachi 64180. The 64180 a highly integrated Z80 type of processor. The processor contains two serial channels, two DMA channels, two counter timers and a memory management unit. Besides the integration of the above peripherals, the instruction set has had any of the wasted cycles removed. This lets the 64180 run faster than the Z80 using the same clock speed.

The 64180 allows you to move the internal peripherals to different I/O addresses. The BASIC interpreter does not modify the I/O addresses, so the I/O space for the internal peripherals starts at address 0 (default power-up address).

SERIAL PORTS:

The serial ports on the LC4 base board are contained in the 64180. Channel 0 is used for COM0 and channel 1 is used for COM1.

The enabling and disabling of the RS422/485 driver chips is handled by writing a value to a memory location. The memory address you write to is 380E0 Hex (remember that the 64180 can talk to 512K of RAM). Bit 0 controls the tri-stating of COM1 and bit 7 controls the tri-stating of COM0. A 1 bit will enable the driver chip and a 0 bit will disable the driver.

WATCHDOG TIMER:

The Watchdog Timer circuitry needs to be banged at least once every 100 ms, else the LC4 will be reset. To bang the Watchdog, output any value to address C0 Hex.

REAL TIME CLOCK:

The Real Time Clock chip is the Epson RTC-62421. It is addressed from 50 Hex to 5F Hex.

ROM:

The LC4 BASIC has 32K of ROM addressed from 0 to 7FFF Hex.

RAM:

There is 64K of battery backed RAM addressed from 10000 Hex to 1FFFF Hex. The startup code for the BASIC maps the RAM to 08000 to 17FFF Hex. If possible, try not to use the RAM above 0FFFF Hex. If you need to use the extra RAM, do not use the RAM from 10000 to 103FF Hex. This section of RAM is used for the serial port input buffers.

SOFTWARE

There are two BASIC statements that are used when accessing your own machine language routines, the CLEAR statement and the CALL statement.

CLEAR STATEMENT

The clear statement is used for allocating space for your machine language routines. The CLEAR will move the start of your BASIC program the specified number of bytes away from the start of RAM (8000 Hex). Your BASIC program will not be trash by this command. The following examples reserves 150 bytes of space for you machine language routine:

```
100   CLEAR 150        'Clear Space For Subroutine
```

NOTE: After clearing some RAM space, you should not do a NEW statement. A NEW will deallocate the RAM you have cleared. To remove the existing BASIC program without deallocating the reserved RAM, do a DELETE command. The following example will remove the existing BASIC program without deallocating the reserved RAM:

```
DELETE                 'Delete With No Numbers Will Delete All
```

CALL STATEMENT

Machine language subroutines are accessed via the CALL statement. Using the CALL statement, you can pass variables to the machine language subroutine. The CALL passes pointers to the variable data, not the data itself.

The possible data types that can be passed are integer variables, floating point variables and string variables. Each variable type uses different amounts of storage, so your subroutine must know what data type(s) it is manipulating.

DATA TYPES

INTEGER VARIABLES

Integer variables take up two contiguous bytes of storage. The low byte of the data is in low memory and the high byte of data is in high memory.

FLOATING POINT VARIABLES

Floating point variables take up four contiguous bytes of storage. The low byte of the mantissa is store in low memory, the middle byte is next, the high byte of the mantissa is next and the exponent is last.

STRINGS

Strings are anywhere from 0 to 255 bytes long. Each string is terminated with a 03 Hex. Do NOT change the length of a string in your machine language subroutine, you will only confuse the BASIC interpreter.

ARRAYS

The data in an array is stored contiguously, array element 0 is followed by array element 1, 1 is followed by 2, etc.

ON THE STACK

The data that is given to a machine language routine is passed on the processor stack. A pointer to each item being passed is pushed on the stack before the subroutine is called. The last item being passed is actually the first item pushed. Examine the following example:

```
CALL TEST (MU, OMEGA1, OMEGA2)
```

After all the parameters have been pushed onto the stack, a processor call will be performed. This will push the return address onto the stack. The stack should look as follows to your machine language routine:

```
high memory   -     pointer to OMEGA2
                    pointer to OMEGA1
                    pointer to MU
low memory    -     return address
```

The stack pointer will be pointing to the low byte of the return address.

WARNING - WARNING - WARNING

If your machine language routine is going to take a while, you must bang the watchdog timer circuitry. If you don't, the LC4 will be reset. The following code segement is the code required for banging the watchdog timer:

```
OUT (0C0H), A          ;bang the watchdog timer
```

EXAMPLE

The following example clears some memory, loads the machine language routine into the cleared RAM and then executes the machine language routine.

```
1000  '
1010  '
1020  '          Clear Some Memory For A Machine Language Subroutine
1030  '
1040  CLEAR 100
1050  '
1060  '          Now Poke Exchange Routine Into Cleared RAM
1070  '
1080  FOR X% = &H8000 TO &H800F
1090  READ P%                             'Get Program Byte
1100  POKE X%, P%                         'Stick Program Byte Into RAM
1110  NEXT
1120  '
1130  '
1140  EXCHANGE% = &H8000        'Define Address Of Subroutine
1150  '
1160  '          Set Up Some Variable To Pass To Routine
1170  '
1180  ALPHA% = 12345
1190  BETA% = 23456
1200  '
1210  '          Print The Variables
1220  '
1230  PRINT ALPHA%, BETA%
1240  '
1250  '          Now Do Call To Exchange The Variables
1260  '
1270  CALL EXCHANGE% (ALPHA%, BETA%)                    'Do Call
1280  '
1290  '          Now Print The Variables Again
1300  '
1310  PRINT ALPHA%, BETA%
1320  '
1330  '
1340  '          This Is The Machine Language Subroutine
1350  '
1360  '                    POP HL          ;get return address from stack
1370  DATA &HE1
1380  '                    POP DE  ;get pointer to 1st parameter
1390  DATA &HD1
1400  '                    EX (SP), HL     ;exchange return with 2nd parameter
```

```
1410  DATA &HE3
1420  '             LD A, (DE)     ;get low byte of 1st parameter
1430  DATA &H1A
1440  '             LD C, (HL)     ;get low byte of 2nd parameter
1450  DATA &H4E
1460  '             LD (HL), A     ;and put 1st into 2nd variable
1470  DATA &H77
1480  '             LDA, C         ;get low byte of 2nd parameter
1490  DATA &H79
1500  '             LD(DE), A      ;and put 2nd into 1st variable
1510  DATA &H12
1520  '             INC HL
1530  DATA &H23
1540  '             INC DEC
1550  DATA &H13
1560  '             LD A, (DE)     ;get high byte of 1st parameter
1570  DATA &H1A
1580  '             LD C, (HL)     ;get high byte of 2nd parameter
1590  DATA &H4E
1600  '             LD (HL), A     ;and put 1st into 2nd variable
1610  DATA &H77
1620  '              LD A, C ;get high byte of 2nd parameter
1630  DATA &H79
1640  '             LD (DE), A     ;and put 2nd into 1st variable
1650  DATA &H12
1660  '             RET
1670  DATA &HC9
1680  '
```

LC4/OPTOMUX NETWORK INTERFACE

The LC4 Local Controller is an ideal device for protocol conversion. A protocol converter is a unit which communicates with one device using one format, translates the information to a different format, then passes the information to a second device which uses the new format. An example would be a local controller which collects information from a unit such as a gas analyzer, then passes this information to a host computer using the OPTOMUX protocol.

By programming the LC4 to pass data using the OPTOMUX protocol, devices such as gas analyzers, bar code readers, keyboards, displays, motion controllers, single loop controllers and programmable controllers can be connected to an OPTOMUX network. The OPTOMUX network provides a standard distributed interface to IBM PC/AT host computers running standard off-the-shelf software packages (such as PARAGON LC, The FIX, and ONSPEC). Therefore, by using the LC4 as a protocol converter, a device can be adapted to a network without having to modify the top level software.

Before we examine how the software for LC4 is written, one must be aware of possible limitations with this kind of implementation. The OPTOMUX protocol passes data as ASCII-Hex characters representing numeric data with an integer range of values. Software packages like PARAGON LC, The FIX, or ONSPEC poll OPTOMUX units and transfer data as either analog, or digital values in the integer range of 0 to 4095 (12-bit values). Discrete or digital data is an integer value in the range of 0 to FFFF Hex, with the status of each bit position representing one position on a discrete module rack.

This example does not apply for applications which must pass ASCII messages back to the host because the software at the host level would have to be modified to accept and correctly interpret ASCII character strings. In this case, it may be wiser to use an AC31 and modify the host software to use the driver. The AC31 is an interface to the OPTOMUX link which adds addressability and checksum protection to ASCII string messages passed to and from standard terminals or ASCII devices. Although, the AC31 uses the OPTOMUX format for START OF MESSAGE, ADDRESS, and CHECKSUM parameters, the imbedded data can be in an ASCII format. However, the LC4 example in this technical note can be applied to applications where the LC4 contains a series of pre-canned messages used to transmit to a display device or even another controller. Each message would then be triggered by the host easily as if it was a discrete position or an analog value.

PROGRAMMING

To allow a programmer to easily create an OPTOMUX shell around a packet of data using the LC2 or LC4, a set of four machine language subroutines have been added to the LC2/LC4 BASIC releases 2 and 3, respectively. These subroutines analyze a received message, send a valid acknowledgement, send an error, and clear the buffer. The routines are accessed using the BASIC CALL statement and are listed as follows:

Routine Location	Description
16	Clear Communications Buffer
31	Receive A Message And Get A Command
34	Send An Acknowledgement (ACK)
37	Send An Error (NAK)

Each of these routine locations must be assigned to a descriptive dummy variable before they are called. The following BASIC statements offer an example:

```
10    CLR.BUF% = 16    'Location Of Clear Buffer Routine
11    GET.CMD% = 31    'Location Of Receive Message Command
12    SEND.ACK% = 34   'Location Of Send Acknowledgement Routine
13    SEND.NAK% = 37   'Location Of Send Error Routine
```

The routines can now be called by using the call statement. An example would be:

```
100   CALL CLR.BUF%   'Go Clear The Buffer
```

The OPTOMUX format can be briefly described as follows:

The start of message character is a greater than sign (>). This character is followed by two ASCII-Hex characters representing the address of the OPTOMUX unit. This address is in the range of 00 to FF Hex (0 to 255 decimal).

The next part of the message contains a command character followed by a positions bit-mask and any data (depending with the command).

The final part of the message is a two character checksum followed by a carriage return.

USEFUL LC2 And LC4 BASIC STATEMENTS

With LC2 and LC4, the host port is always open and can be considered logical file number 0. Therefore, the INPUT #0, INPUT$, and LOC(0) statements can be used to get a specific number of characters from the buffer.

The Clear Host Communications Buffer Routine

This routine is used to easily remove unwanted messages in the host communications buffer. No parameters are passed to this routine.

Sample Call Statement

```
CALL CLR.BUF%
```

where CLR.BUF% = 16

The Receive Message/Get Command Routine

This routine is used to receive OPTOMUX type command messages at the HOST communications port of LC2 or LC4. The routine should be called when a carriage routine is received at the host port. This can be accomplished by using the ON KEY statement in BASIC. A single integer parameter containing the address must be passed to the routine. This address is what the subroutine will use to match with the address in the received message.

Sample Call Statement

```
CALL GET.CMD%(A%)
```

where GET.CMD% = 31 and A% = the address you wish to match.

NOTE: The returned value will be placed in the parameter which held the address (A% in the above example) when the call was made. Therefore, only use variables that are temporary when passing parameters to this subroutine.

After the routine is executed one of the following three conditions will exist:

1. The routine will return a value of zero if the message was for an address different than the one that was passed. In this case, the host communications buffer will NOT have been cleared.

2. The routine will return a negative value if the message had the correct address, but a checksum error was detected. In this case, the communications buffer will NOT have been cleared.

3. The routine will return the ASCII value of the command character if the address and the checksum verify. In this case, the start of message character, two address characters, the command character, and the two checksum characters will have been removed from the input buffer.

The Send Acknowledge Routine

This routine is used to return data to the host. A single string variable must be passed to the routine. The routine will append the "A" character at the beginning of the string and place two checksum characters and a carriage return at the end of the string. If a null string is passed to the routine, a simple acknowledge will be returned (Acr).

Sample Call Statement

```
CALL SEND.ACK%(D$)
```

where SEND.ACK% = 34 and D$ is the data string.

The Send Error Routine

This routine is used to return error messages. A single integer parameter must be passed to the routine and be within the range of 0 to 255. The integer value will be returned as the error code to the host with the "N" character at the beginning and a carriage return at the end. The OPTOMUX error messages do not use a checksum.

Sample Call Statement

```
CALL SEND.NAK%(E%)
```

where SEND.NAK% = 37 and E% is the error number from 0 to 255.

Below is a list of OPTOMUX error messages which can be used to simulate an OPTOMUX type error to the existing host programs which act on these errors.

Error	Description
00	Power Up Clear Expected A command other than an "A" was attempted after a power-up or power fail condition. Once the error is returned, it is unnecessary to execute a Power Up Clear command, the next command will execute normally.
01	Undefined Command The command character was not a valid command character.
02	Checksum Error The checksum received did not match the sum of the characters in the command.
03	Input Buffer Overrun The received command contained more than 71 characters for analog or 16 characters for digital OPTOMUX messages.
04	Non-printable ASCII Character Received Only characters from 21 Hex to 7F Hex are permitted in OPTOMUX messages.
05	Data Field Error Not enough characters were received.
06	Communications Link Watchdog Time Out Error
07	Specific Limits Invalid

SUBROUTINE EXAMPLES

The following LC2/LC4 BASIC program segments give examples of using each command. This is not a complete program, and only illustrates how these subroutines may be used.

```
'
'            USEFUL CONSTANTS
'
110   CLR.BUF% = 16          'Clear Buffer Routine Location
120   GET.CMD% = 31          'Receive Message Location
130   SEND.ACK% = 34         'Location Of Send Acknowledgement Routine
140   SEND.NAK% = 37         'Location Of Send Error Routine
150   BOARD% = 255           'Phony OPTOMUX Address To Respond To
160   CKSM.ERR% = 2          'OPTOMUX Checksum Error
170   ACK$ = ""                      'A Simple Acknowledge
'
'     INITIALIZE TO INTERRUPT ON A RECEIVED MESSAGE
'
610   ON KEY(CHR$(13)) GOSUB 800             'Set A Jump On A Carriage Return
620   KEY(CHR$(13)) ON                       'Activate Interrupt
'
'            HANDLE THE RECEIVED MESSAGE
'
800   A% = BOARD%                    'Put Address In Temporary Variable
810   ACK$ = ""                              'Simple Acnowledge Acr
820   CALL GET.CMD%(A%)                      'Pass The Temporary Variable
830   IF A% = 0 THEN RETURN          'Command Not For Us
840   IF A% < 0 THEN GOTO 2000       'Checksum Error, Handle It
850   IF A% = 74 THEN GOSUB 3000             'Handle If A% = ASCII J - Write Outs
860   CALL CLR.BUF%                  'Clear The Buffer
870   CALL SEND.ACK%(ACK$)           'Return A Simple Acknowledge
880   RETURN
'.
'     RESPONSE TO A CHECKSUM ERROR
'.
2000  CALL CLR.BUF%                  'Clear Out The Junk
2010  CALL SEND.NAK%(CKSM.ERR%)      'Return A Checksum Error
2020  RETURN
'.
'     HANDLE A "J" COMMAND - WRITE OUTPUTS (Analog)
'.
3000  AMASK% = VAL("&H" + INPUT$(4,0)) 'Get The Bitmask
3010  ADATA% = VAL("&H" + INPUT$(3,0)) 'Get Value
3020  RETURN
```

SAMPLE PROGRAM

```
10    '
20    '       LCMUX - PROGRAM TO MAKE AN LC4 LOOK LIKE A
30    '               DIGITAL AND AN ANALOG OPTOMUX.
40    '
50    '       The LC4 will respond to commands sent to the
60    '       following OPTOMUX addresses:
70    '
80    '       DIGITAL - 255
90    '       ANALOG - 239
100   '
110   '       This program will answer to the following commands:
120   '
130   '       A       - Power Up Clear (digital and analog)
140   '       B       - Reset (digital and analog)
150   '       C       - Turn Around Delay (acknowledge but do nothing)
160   '       D       - Watchdog Delay (acknowledge but do nothing)
170   '       E       - Protocol (acknowledge but do nothing)
180   '       F       - Identify OPTOMUX (digital and analog)
190   '       G       - Configure Positions (acknowledge but do nothing)
200   '       H       - Configure Inputs (acknowledge but do nothing)
210   '       I       - Configure Outputs (acknowledge but do nothing)
220   '       J       - Write Outputs (digital and analog)
230   '       K       - Activate Outputs - Digital, Read Outputs -  Analog
240   '       L       - Deactivate Outputs - Digital, Read Inputs - Analog
250   '       M       - Read ON/OFF Status - Digital
260   '       S       - Update Outputs - Analog
270   '
280   '
290   '       This program will respond with the following errors:
300   '
310   '       00      - Power Up Clear Expected
320   '       01      - Undefined Command Error
330   '       02      - Checksum Error
340   '       05      - Insufficient Or Incorrect Data Error
350   '
360   '
370   '       Two arrays are used for the analog values that are
380   '       passed.  These arrays are labeled AINS%( ) for the inputs
390   '       and AOUTS%( ) for the outputs.
400   '
410   '       Integer values are used for the digital values; DINS% for
420   '       inputs and DOUTS% for the outputs.
430   '
440   '       For demonstration purposes, the main program loop fills
450   '       the AINS%( ) and DINS% variables with values, so that a
460   '       host program can poll and receive changing values.
470   '       In an actual application, the actual field device would
480   '       be interrogated, and its values placed in the appropriate
490   '       variables.
```

```
500     '
510     '
520     '         NOTE:  Be careful when making the polling routines to the
530     '         slave device such that they don't tie up the LC4 interrupts.
540     '         The LC4 needs to respond quickly to the host polling program
550     '         so as not to cause a turn around delay error at the host.
560     '         Avoid using commands such as INPUT, DELAY, and WAIT which
570     '         tie up LC4 and prevent it from servicing the ON COM and
580     '         ON TIMER commands.
590     '
600     '*********************************************************************
610     '
620     '         VARIABLE ASSIGNMENTS
630     '
640     DIM AINS%(15)              'Analog Input Variables
650     DIM AOUTS%(15)  'Analog Output Variables
660     CLR.BUF% = 16    'Clear Buffer Routine Location


670     GET.CMD% = 31   'Get Command If There Is One Locations
680     SEND.ACK% = 34  'Location Of Acknowledge Routine
690     SEND.NAK% = 37  'Location Of Error In Acknowkedge Routine
700     BRD.ADR.A% = 239           'Address Of Analog
710     BRD.ADR.D% = 255           'Address Of Digital
720     DINS% = 0                  '16 Bit Digital Input Varialbe
730     DOUTS% = 0                 '16 Bit Digital Output Variable
740     '
750     '         COMMAND CONSTANTS
760     '
770     PUC% = 0                   'Power Up Clear Command
780     RESET% = 1                 'Reset Command
790     CONFIGURE% = 8 'Configure Outputs Command
800     ACTIVATE% = 10  'Activate Digital Outputs Command
810     DEACTIVATE% = 11           'Deactivate Digital Outputs Command
820     READ.ANL% = 37  'Read Analog Outputs Command
830     WRITE.ANL% = 35 'Write Analog Outputs Command
840     ACK$ = ""                  'Null String For Simple Acknowledge
850     DIG.ID$="00"               'ID If You want To Be A Digital Board
860     ANL.ID$="01"               'ID If You Want To Be An Analog Board
870     CKSM.ERR% = 2   'Checksum Error Code
880     BAD.CMD.ERR% = 1           'Invalid Command Error Code
890     BAD.DAT.ERR% = 5           'Invalid Command Error Code
900     PUC.ERR% = 0               'Power Up Clear Error Code
910     A.PWR.FLAG% = 0'Power Up Clear Flag - Analog
920     D.PWR.FLAG% = 0'Power Up Clear Flag - Digital
930     DEFAULT% = 0               'Default Analog Output Value
940     '
950     '    MASK CONSTANTS
960     '
970     DIM MASK%(15)   'Dimension The Array
980     MASK%(0) = &H0001          'Position 0 Mask Bit
990     MASK%(1) = &H0002                          'Position 1 Mask Bit
```

```
1000  MASK%(2) = &H0004                  'Position 2 Mask Bit
1010  MASK%(3) = &H0008                  'Position 3 Mask Bit
1020  MASK%(4) = &H0010                  'Position 4 Mask Bit
1030  MASK%(5) = &H0020                  'Position 5 Mask Bit
1040  MASK%(6) = &H0040                  'Position 6 Mask Bit
1050  MASK%(7) = &H0080                  'Position 7 Mask Bit
1060  MASK%(8) = &H0100                  'Position 8 Mask Bit
1070  MASK%(9) = &H0200                  'Position 9 Mask Bit
1080  MASK%(10) = &H0400                 'Position 10 Mask Bit
1090  MASK%(11) = &H0800                 'Position 11 Mask Bit
1100  MASK%(12) = &H1000                 'Position 12 Mask Bit
1110  MASK%(13) = &H2000                 'Position 13 Mask Bit
1120  MASK%(14) = &H4000                 'Position 14 Mask Bit
1130  MASK%(15) = &H8000                 'Position 15 Mask Bit
1140  '
1150  '
1160  '       INITIALIZE HOST INTERRUPT
1170  '
1180  ON KEY(CHR$(13)) GOSUB 1410    'Interrupt On Carriage Return
1190  KEY(CHR$(13)) ON                   'Activate Interrupt
1200  '
1210  '*****************************************************************
1220  '
1230  '               MAIN PROGRAM LOOP GOES HERE
1240  '
1250  '*****************************************************************
1260  '
1270  FOR I% = 0 TO 15               'Go Through All 16 Positions
1280  AINS%(I%) = 0                      'Initialize To 0
1290  NEXT
1300  FOR P% = 0 TO 4095                 'Go Through A Large Range Of Values
1310  FOR I% = 0 TO 15               'Do All 15 Analog Positions
1320  AINS%(I%) = P%                 'Set Equal To Outer Loop Value
1330  NEXT


1340  DINS% = INT(TIMER/3)               'Set The Digital To A Sequential Value
1350  NEXT
1360  GOTO 1270                          'Do It Again
1370  END
1380  '
1390  '       HOST INTERRUPT ROUTINE (UPON RECEIPT OF CARRIAGE RETURN)
1400  '
1410  C% = BRD.ADR.A%                    'Set Analog Address
1420  CALL GET.CMD%(C%)                  'Routine To Check For Analog Board
1430  IF C% = 0 THEN GOTO 1460       'Command Not For Analog
1440  IF C% > 0 THEN GOTO 1620       'Command For Analog Else Checksum Error
1450  GOTO 2720
1460  C% = BRD.ADR.D%                    'Set Digital Address
1470  CALL GET.CMD%(C%)                  'Check For Digital Board
1480  IF C% = 0 THEN GOTO 1510       'Command Not For Digital Either
1490  IF C% > 0 THEN GOTO 1560       'Command For Digital Else Checksum Error
```

```
1500  GOTO 2720
1510  CALL CLR.BUF%
1520  RETURN
1530  '
1540  '          CHECK FOR POWER UP ON DIGITAL
1550  '
1560  IF C% = 65 THEN GOTO 2010                       'Command Is An A, Power Up Clear
1570  IF D.PWR.FLAG% > 0 THEN GOTO 1700        'See If We Lost Power
1580  GOTO 2850
1590  '
1600  '          CHECK FOR POWER UP ON ANALOG
1610  '
1620  IF C% = 65 THEN GOTO 2240                       'Command Is An A, Power Up Clear
1630  IF A.PWR.FLAG% > 0 THEN GOTO 2100        'See If We Lost Power
1640  GOTO 2780
1650  '
1660  '*******************************************************************
1670  '
1680  '          PROCESS DIGITAL COMMANDS
1690  '
1700  IF C% < 65 THEN GOTO 2600                       'Command Not Valid
1710  ON C% - 65 GOTO 2370, 2600, 2600, 2600, 2310
1720  ON C% - 70 GOTO 2180, 2180, 2180, 1770, 1830, 1890, 1950
1730  GOTO 2600                                       'If You Get Here, Its Bad
1740  '
1750  '          DIGITAL WRITE COMMAND
1760  '
1770  DOUTS% = VAL("&H" + INPUT$(LOC(0),0))     'Grab New Bitmask
1780  CALL SEND.ACK%(ACK$)                      'Send Acknowledge
1790  RETURN
1800  '
1810  '          TURN DIGITAL OUTPUTS ON
1820  '
1830  DOUTS% = DOUTS% OR VAL("&H" + INPUT$(LOC(0),0)) 'Get Bitmask
1840  CALL SEND.ACK%(ACK$)                      'Send Acknowledge
1850  RETURN
1860  '
1870  '          TURN DIGITAL OUTPUTS OFF
1880  '
1890  DOUTS% = DOUTS% AND (NOT(VAL("&H" + INPUT$(LOC(0),0))))  'Get Mask
1900  CALL SEND.ACK%(ACK$)                      'Send Acknowledge
1910  RETURN
1920  '
1930  '          READ DIGITAL STATUS
1940  '
1950  RESPONSE$ = RIGHT$("0000" + HEX$(DINS%),4)                'Build Message
1960  CALL SEND.ACK%(RESPONSE$)                 'Send It
1970  RETURN
1980  '
1990  '          RESPONSE TO A POWER UP CLEAR COMMAND (DIGITAL)
2000  '
```

```
2010   CALL CLR.BUF%                 'Clear Out The Junk
2020   CALL SEND.ACK%(ACK$)          'Send Acknowledge
2030   D.PWR.FLAG% = 1               'Set The Power Up Flag
2040   RETURN
2050   '
2060   '**********************************************************
2070   '
2080   '        PROCESS ANALOG COMMANDS
2090   '
2100   IF C% < 65 THEN GOTO 2600              'Command Not Valid
2110   ON C% - 65 GOTO 2510, 2600, 2600, 2600, 2450
2120   ON C% - 70 GOTO 2180, 2180, 2180, 2920, 3270, 3140
2130   IF C% = 83 GOTO 3040                   'Update Analog Out Command
2140   GOTO 2600                              'If You Get Here, Its Bad
2150   '
2160   '        SIMPLE ACKNOWLEDGE
2170   '
2180   CALL CLR.BUF%                 'Clear Out The Junk
2190   CALL SEND.ACK%(ACK$)          'Send Acknowledge
2200   RETURN
2210   '
2220   '        RESPONSE TO A POWER UP CLEAR COMMAND
2230   '
2240   CALL CLR.BUF%                 'Clear Out The Junk
2250   CALL SEND.ACK%(ACK$)          'Send Acknowledge
2260   A.PWR.FLAG% = 1               'Set The Power Up Flag
2270   RETURN
2280   '
2290   '        RESPONSE TO AN IDENTIFY OPTOMUX COMMAND (DIGITAL)
2300   '
2310   CALL CLR.BUF%                 'Clear Out The Junk
2320   CALL SEND.ACK%(DIG.ID$)       'Send Acknowledge
2330   RETURN
2340   '
2350   '        RESPONSE TO A RESET COMMAND (DIGITAL)
2360   '
2370   DINS% = 0                              'Set Digital Value To 0
2380   DOUTS% = 0                             'Set Digital Value To 0
2390   CALL CLR.BUF%                 'Clear Out The Junk
2400   CALL SEND.ACK%(ACK$)          'Send Acknowledge
2410   RETURN
2420   '
2430   '        RESPONSE TO AN IDENTIFY OPTOMUX COMMAND (ANALOG)
2440   '
2450   CALL CLR.BUF%                 'Clear Out The Junk
2460   CALL SEND.ACK%(ANL.ID$)                'Send Acknowledge
2470   RETURN
2480   '
2490   '        RESPONSE TO A RESET COMMAND (ANALOG)
```

```
2500 '
2510 FOR C% = 0 TO 15
2520 AOUTS%(C%) = DEFAULT%
2530 NEXT
2540 CALL CLR.BUF%                  'Clear Out The Junk
2550 CALL SEND.ACK%(ACK$)           'Send Acknowledge
2560 RETURN
2570 '
2580 '       RESPONSE TO A BAD COMMAND (ERROR)
2590 '
2600 CALL CLR.BUF%                  'Clear Out The Junk
2610 CALL SEND.NAK%(BAD.CMD.ERR%) 'Return An Error
2620 RETURN
2630 '
2640 '       RESPONSE TO BAD DATA (ERROR)
2650 '
2660 CALL CLR.BUF%                  'Clear Out The Junk
2670 CALL SEND.NAK%(BAD.DAT.ERR%) 'Return An Error
2680 RETURN
2690 '
2700 '       RESPONSE TO A CHECKSUM ERROR
2710 '
2720 CALL CLR.BUF%                  'Clear Out The Junk
2730 CALL SEND.NAK%(CKSM.ERR%)      'Return The Error
2740 RETURN
2750 '
2760 '       RESPONSE TO A PUC - ANALOG
2770 '
2780 CALL CLR.BUF%                  'Clear Out The Junk
2790 CALL SEND.NAK%(PUC.ERR%)               'Return The Error
2800 A.PWR.FLAG% = 1                'We Powered Up
2810 RETURN
2820 '
2830 '       RESPONSE TO A PUC - DIGITAL
2840 '
2850 CALL CLR.BUF%                  'Clear Out The Junk
2860 CALL SEND.NAK%(PUC.ERR%)               'Return The Error
2870 D.PWR.FLAG% = 1                'We Powered Up
2880 RETURN
2890 '
2900 '       WRITE ANALOG OUTPUTS COMMAND
2910 '
2920 IF LOC(0) > 7 THEN GOTO 2660   'Too Many Characters For This Command
2930 AMASK% = VAL("&H" + INPUT$(4,0)) 'Get The Bitmask
2940 ADATA% = VAL("&H" + INPUT$(3,0)) 'Get The Data Value
2950 FOR S% = 15 TO 0 STEP -1       'Check All 15 Positions
2960 IF (AMASK% AND MASK%(S%)) = 0 THEN GOTO 2980   'If Not This, Next
2970 AOUTS%(S%) = ADATA%            'Get The Value
2980 NEXT
2990 CALL SEND.ACK%(ACK$)           'Send Acknowledge
```

```
3000  RETURN
3010  '
3020  '                UPDATE ANALOG OUTPUTS
3030  '
3040  AMASK% = VAL("&H" + INPUT$(4,0)) 'Get The Bitmask
3050  FOR S% = 15 TO 0 STEP -1          'Check All 15 Positions
3060  IF (AMASK% AND MASK%(S%)) = 0 THEN GOTO 3080            'If Not This, Next
3070  AOUTS%(S%) = VAL("&H" + INPUT$(3,0))                    'Get The Value
3080  NEXT
3090  CALL SEND.ACK%(ACK$)              'Send Acknowledge
3100  RETURN
3110  '
3120  '                READ ANALOG INPUTS COMMAND
3130  '
3140  AMASK% = VAL("&H" + INPUT$(LOC(0),0))      'Get The Bitmask
3150  AVALUE$ = ""
3160  ANSWER$ = ""
3170  FOR O% = 15 TO 0 STEP -1          'Check All 15 Positions
3180  IF (AMASK% AND MASK%(O%)) = 0 THEN GOTO 3210            'If Not This, Next
3190  AVALUE$ = "0000" + HEX$(AINS%(O%) + 4096)               'Add 1000 Hex Offset
3200  ANSWER$ = ANSWER$ + MID$(AVALUE$,LEN(AVALUE$) - 3)      'Get The Value
3210  NEXT
3220  CALL SEND.ACK%(ANSWER$)                    'Send Acknowledge
3230  RETURN
3240  '
3250  '                READ ANALOG OUTPUTS COMMAND
3260  '
3270  AMASK% = VAL("&H" + INPUT$(LOC(0),0))      'Get The Bitmask
3280  AVALUE$ = ""
3290  ANSWER$ = ""
3300  FOR M% = 15 TO 0 STEP -1          'Check All 15 Positions
3310  IF (AMASK% AND MASK%(M%)) = 0 THEN GOTO 3340            'If Not This, Next
3320  AVALUE$ = "000" + HEX$(AOUTS%(M%))                      'Do Not Add 1000 Hex Offset
3330  ANSWER$ = ANSWER$ + MID$(AVALUE$,LEN(AVALUE$) - 2)      'Get The Value
3340  NEXT
3350  CALL SEND.ACK%(ANSWER$)                    'Send Acknowledge
3360  RETURN
```

USING OPTO 22'S UTILITY PROGRAMS

The following three programs are located on the OPTOWARE Utility Diskette. The Utility Diskette (P/N 9909) is included with the OPTOWARE Driver Package (P/N 9900). If you need a copy of the Utility Diskette, please call us at 800-854-8851 or 714-891-5861 inside California. Opto 22 also has these programs on our Bulletin Board System; the BBS phone number is 714-892-8375.

HOST.EXE

The HOST program allows the user to send ASCII Hex characters out the serial port to the OPTOMUX network. This is useful in checking out the OPTOMUX hardware and configuration and/or debugging your own OPTOMUX driver. To run the program, type HOST at the DOS prompt. A menu of selections will be displayed on the screen. Below is a copy of the HOST screen.

```
          OPTOMUX HOST PROGRAM        Version 2.00

          PORT: COM1 AT 3f8 , 19200 BAUD
          PROTOCOL: 2 PASS    REPEAT COUNT = 1
          ERROR DETECT: ON    CHECKSUM ENABLE: ON


           PLEASE CHOOSE ONE OF THE BELOW
             1  CHANGE COMMAND BUFFER
             2  ENTER IMMEDIATE MODE
             3  TYPE COMMAND BUFFER
             4  CHANGE REPEAT COUNT
             5  TOGGLE PROTOCOL
             6  TOGGLE ERROR DETECT
             P  CHANGE COM PORT
             B  CHANGE BAUD RATE
             Q  QUIT
          <cr>  SEND BUFFER

     PLEASE ENTER SELECTION: _____
```

The HOST program automatically enables the CAPS LOCK key, because the OPTOMUX commands use mostly upper case alphabets. The program defaults to COM1, 19200 baud, two-pass protocol, repeat count of one, and error detect on and checksum enabled.

The following is a summary of the commands to use HOST:

1 Press 1 to enter OPTOMUX commands into the buffer. The program puts the start of command character (>) at the beginning of the line. If you use the default settings, the program also computes the checksum of that command line. To complete the command line, enter a carriage return. To enter additional commands, type the next OPTOMUX command on the next line. To get back to the main menu, enter a carriage return on a blank line.

2 Press 2 to get into the Immediate mode. In the Immediate mode, the command you enter on the screen will be transmitted immediately when you enter a carriage return. The program puts the start of command character (>) at the beginning of the command and, if the checksum enable is on, computes the checksum of that command line. Press the ESC key will return you to the main menu.

3 Press 3 to display the command(s) in the command buffer on the screen.

4 Press 4 to change the repeat count. This selects how many iterations the command buffer will be send to the OPTOMUX network.

5 Press 5 to toggle between two- and four-pass protocol. This must match the jumper setting of the B10 jumper on the OPTOMUX Brain Board (B1 or B2).

6 Press 6 to cycle through the four possible combinations listed below.

ERROR DETECT	CHECKSUM ENABLE
ON	ON
OFF	OFF
ON	OFF
OFF	ON

With the ERROR DETECT off, the audible error beep and the pause functions are disabled. This is useful when you want to put an oscilloscope on the communication line. With the CHECKSUM ENABLE off, the program do not compute the checksum of the OPTOMUX command; you must compute the checksum or enter two question marks (??).

P Press P to cycle through the six possible serial port selection. The serial port's Hex address is also displayed on the main menu. The fifth and sixth serial port addresses are for communicating from 5

Paragon LC to an LC4 as COM 3 and 4.

B Press B to cycle through the possible baud rate selection. The baud rates are: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115,200.

Q Press Q to quit the HOST program and get back to DOS.

<CR> Press the carriage return key to transmit the command(s) in the command buffer. Some computers label the carriage return key as "Enter" or "Return."

For information on the OPTOMUX commands, please refer to the "OPTOMUX B1 And B2 Digital And Analog Brain Boards Operations Manual," Form 203.

OS.COM

OptoScan is a utility program that reads your OPTOMUX network and responds back with the settings of all the OPTOMUX stations on the network. It also allows you to enter the parameters of a given OPTOMUX station to poll that station for information and allow you to make changes to that station.

The following are tutorial/demos on how to use OptoScan on your OPTOMUX network. To run the program, type OS at the DOS prompt.

Example one:

After receiving your OPTOMUX system, you would like to set your Group A and B jumpers on the Brain Boards.

1. At the main menu, move the cursor to the Baud and enter the desired baud rate.
2. Move the cursor and select the desired protocol (two- or four-pass, [2] or [4]), station address ([0] to [255]), link mode ([M]ultidrop or [R]epeat mode), and last board ([Y]es or [N]o).
3. After making the appropiate selections, the Group A and B Jumpers Sections will display the correct jumper settings for Groups A and B.
4. To get new settings, repeat steps 1 through 3.

Example two:

After completing example one and wiring the OPTOMUX communications network, you would like to test the communications link.

1. At the main menu, move the cursor to PORT and enter the serial port number of the OPTOMUX network (1 thru 4). Press the F4 function key, and enter the desired delay (use the default value). The OptoScan program will communicate to the OPTOMUX network starting at station address 0 and 300 baud. OptoScan will increment the station address from 0 to 255 at 300 baud, sending a Power Up Clear command to the OPTOMUX board address. If the board responds with an acknowledge, it will send an Identify OPTOMUX Type command. However, if the board response with a -28 error (invalid return data), OptoScan will change to four-pass protocol and re-issue the Power Up Clear command. If the board responds with an acknowledge, OptoScan will record the settings for the station address, switch back to two-passs protocol, and increment to the next station address. After incrementing to station address 255, OptoScan will double the baud rate and start at station address 0. OptoScan will increment up to station address 255 at 38400 baud and stop. On the right side of the main menu, OptoScan will display the station address, OPTOMUX type, baud rate, and protocol settings of all the OPTOMUX stations from which OptoScan received an acknowledge response.

2. Check your configuration with the display on the right side of the main menu, if you need to change any of your OPTOMUX station settings refer to example one.

Example three:

After completing examples one and two and all the field wiring, you would like to read from and write to field devices on a particular OPTOMUX station.

1. Set the settings for that OPTOMUX station on the upper left side of the main menu (BAUD, DELAY, PROTOCOL, and ADDRESS). Toggle to AUTO scan by pressing the F3 function key and then press the F2 function key to SCAN the particular OPTOMUX station. The status of all 16 points is displayed in the middle of the main menu.

2. Configure the 16 points using the up and down arrow keys and the F9 function key to toggle between input and output module. OptoScan displays 16 positions whether you use a four, eight, or 16 point I/O mounting rack. For a digital input module, when the field input turns on the LED on the I/O mounting rack will turn on and OptoScan will change from OFF to ON at that module position. For a digital output module, you can turn ON/OFF the output module by pressing the F9 function key with the cursor positioned at the module position under the STATE column. Press the F10 function key to get back to the main menu.

3. For analog modules, step 2 is still valid. However, the VALUE column displays the raw count (0 to 4095) for the analog modules instead of the ON/OFF status. To enter data, you need to press the F9 function key and then an integer number from 0 to 4095 (0 to 100 percent of scale).

4. If you want a record of any errors, press the F5 function key to active the error logging routine. The errors are logged onto a file named ERROR.TXT.

NOTE: Press the F1 function key for help and follow the instructions on the bottom of the screen. Press the F10 function key to get to the previous screen or to exit from OptoScan.

USER.EXE

The USER program allows you to use the OPTOWARE driver to communicate to the OPTOMUX network. The USER program allows you to enter data into the different variables (ERRORS, ADDRESS, COMMAND, POSITIONS, MODIFIERS, and INFO) used by the OPTOWARE driver and then call the driver to send the command to the OPTOMUX network. The following is a demostration of how to use the USER program. To run the program, type USER at the DOS prompt.

1. To use the USER program, you must first set the serial port number and then configure the serial port. To set the serial port number from the main menu, use the arrow keys to move the cursor over to Driver title and press the carriage return. This displays the Driver menu with the list of driver commands. Use the arrow keys to move the cursor down to the Set Port Number command (command 102) and press the carriage return. The OPTOWARE command menu will be displayed. The OPTOWARE Command menu is divided into two parts. The upper half describes the command you have selected and contains an example using the command in IBM BASIC. Use the PgUp and PgDn keys to page up and down the descriptions half of the screen.

   The lower half of the screen contains the variables used by OPTOWARE. Refer to the upper half of the screen or the OPTOWARE manual regarding how to use the command. Use the arrow keys to move the cursor to the different variables and variable array elements. To set the serial port number, use the arrow keys to move the cursor to INFO ARRAY (0) and enter the appropriate serial port number (1 to 4).

   Once the OPTOWARE command is set correctly, press the F2 function key to call the OPTOWARE driver and execute the command. If there is an error in the OPTOWARE driver or communication to OPTOMUX, an error number and error message will be displayed. Otherwise, the ERRORS variable is set to 0 and the Driver Status displays a "Command finished" message.

2. The next command is to configure the serial port. Press the ESC key to get back to the Driver menu and select the Configure Serial Port command (command 104). Press carriage return to get to the OPTOWARE command menu. To configure the serial port, enter the baud rate in INFO%(0) array (i.e., 300, 600, ..., 38400).

   Once the OPTOWARE command is set correctly, press the F2 function key to call the OPTOWARE driver and execute the command. If there is an error in the OPTOWARE driver or communication to OPTOMUX, an error number and error message will be displayed. Otherwise, the ERRORS variable is set to 0 and the Driver Status displays a "Command finished" message.

3. This concludes setting up the serial port for communication to the OPTOMUX network. The next step would be to issue System commands (i.e., Power Up Clear, Set Turn Around Delay, etc.) to each OPTOMUX board. After the System commands, the Configure commands must be issued. The Configure commands informs the OPTOWARE driver which positions are inputs, outputs, or temperature probes.

4. After issuing the Driver, System, and Configure commands, you are ready to read and write to the I/O modules and setup the Latches, Time Delays, Waveforms, etc. commands.

USING THE OPTOWARE DRIVER With Borland's Turbo C++

The OPTOWARE driver can be called from Borland's Turbo C++ by using the following statements in your program.

```
    #include <stdio.h>
    #include <dos.h>

  /*****************************************************************************/
  /*     The following statements are the variable and array             */
  /*     definitions needed to call the driver.  All parameters are      */
  /*     declared as global.                                             */
  /*                                                                     */
  /*     Define Driver Parameters                                        */
  /*                                                                     */
  /*****************************************************************************/

  int near   errors;               /* Driver error status */
  int near   address;              /* OPTOMUX Board Address Range 0-255 */
  int near   command;              /* OPTOMUX Command - See OPTOWARE Manual */
  int near   positions[16];        /* Module positions table - See OPTOWARE Manual:
                                      POSITIONS array */
  int near   modifiers[2];         /* Modifiers table - See OPTOWARE Manual:
                                      MODIFIERS array */
  int near   info[16];             /* Data table - See OPTOWARE Manual: INFO array */

  /*****************************************************************************/
  /*     Declare the driver module as an extern "C" module               */
  /*     so that the compiler does not perform what is known             */
  /*     as name mangling with the Optoware driver name.                 */
  /*                                                                     */
  /*     extern "C" { } -  This declaration tells the compiler to        */
  /*     suppress name mangling of non-C++ modules.                      */
  /*                                                                     */
  /*     void - This keyword specifies no return values.                 */
  /*                                                                     */
  *      far - Aligns all memory models.                                 */
  /*                                                                     */
  /*     pascal - This keyword causes arguments to be pushed on          */
  /*     the stack from left to right (last argument is last pushed).    */
  /*                                                                     */
  /*     int - Integer variable.                                         */
  /*                                                                     */
  /*     near - Passes only the offset address, not the segment.         */
  /*                                                                     */
  /*     *p1, *p2, *p3, ... *p6 - Declares pointers as 16 bits.          */
  /*                                                                     */
  /*****************************************************************************/

  extern "C"
  {
        void far pascal optoware(int near *p1, int near *p2, int near *p3,
            int near *p4, int near *p5, int near *p6);
  }
```

Below is a sample main program which uses the declarations made earlier. The example sets the driver to use COM port 2, configures the serial port, then sends a Power Up Clear command to the OPTOMUX unit at address 255.

```
main()
{
        /* select the com port */
        errors= 0;                  /* initialize errors variable to 0 */
        address= 255;               /* initialize address variable */
        command= 102;               /* command to select com port */
        info[0]= 2;                 /* selects port 2 */
        optoware(&errors, &address, &command, positions, modifiers, info);
        printf("\nThe return error is: %d\n", errors);

        /* configure the serial port */
        command= 104;               /* configure serial port command */
        info[0]= 19200;
        optoware(&errors, &address, &command, positions, modifiers, info);
        printf("\nThe return error is: %d\n", errors);

        /* send power up clear */
        command= 0;                 /* power up clear command */
        info[0]= 0;                 /* initialize info[0] again */
        optoware(&errors, &address, &command, positions, modifiers, info);
        printf("\nThe return error is: %d\n", errors);
}
```

Make sure that you link either DRIVER.OBJ or IDRIVER.OBJ file with your program. This can be easily done by declaring the path and file name in the project file so that Turbo C++ can find the driver when compiling.