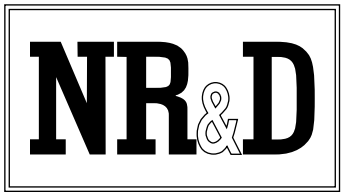
UCM

Installation and Programming Manual

This Manual describes the UCM1 and UCM4 Universal Communication Module, its uses and set up. It also describes the use of the programming software and compiler.

Effective: 6 October, 1999



Niobrara Research & Development Corporation
P.O. Box 3418 Joplin, MO 64803 USA

Telephone: (800) 235-6723 or (417) 624-8918
Facsimile: (417) 624-8920
Internet: http://niobrara.com

SY/MAX and Square D are registered trademarks of Square D Company.

PowerLogic is a trademark of Square D Company.

Allen-Bradley, A-B, and Data Highway are trademarks of Allen-Bradley Company.

Subject to change without notice.

Contents

5 UCM Language Statements 39

Figures

Tables

1
Introduction

The Niobrara Universal Communication Module (UCM) is a SY/MAX® compatible communication coprocessor which most people think of as a very fast D-LOG with the ability to read and write PLC rack addressed registers. The UCM is used to communicate between a peripheral, or a network of peripherals, and the SY/MAX family of controllers through the register rack. The UCM is programmed using a language developed by Niobrara especially for communications between serial devices and PLC registers.

The UCM is available in the following configurations:

- One (1) RS-422 port, the UCM1-D
- Four (4) RS-422 ports, the UCM4-D
- Four (4) RS-485 (2-wire or 4-wire) ports, the UCM4-M
- Four (4) RS-232 ports, the UCM4-S

See Figure 1-2 for the front panel configuration of the UCM1.
See Figure 1-3 for the front panel configuration of the UCM4.

The UCM is field programmed to the exact protocol of the attached device using a compiled "BASIC"-like programming language. This language offers rack addressable registers for use as variables, convenient commands for transmitting and receiving data strings, checksum calculations, and a variety of flow control commands. The UCM configuration file is written using any text editor and then compiled using the COMPILE.EXE utility supplied with the module. (COMPILE runs on MS-DOS® compatible personal computers.) The compiled code is loaded into the UCM using another utility, UCMLOAD.EXE, which communicates to the module using the SY/MAX protocol.

The ports on the UCM are able to function in two modes: SY/MAX and UCM mode. When the UCM is not running the user program on a port, that port is placed in SY/MAX mode allowing programs to be downloaded and registers to be viewed or modified. When a port is running the user program, that port will attempt to communicate with the attached devices according to the user's UCM program. The user pro-

gram has the ability to change the mode to SY/MAX for runtime SY/MAX communication with external SY/MAX compatible devices. The READ, WRITE, and PRINT commands allow full featured SY/MAX compatibility including dynamic routes.

The SY/MAX capabilities of UCM make the module ideal for remote rack polling of SY/MAX devices such as PowerLogic Circuit Monitors. With a very simple program, a network of CMs may be monitored with the data placed in rack addressable PLC registers. This presentation of the data as registers is perfect for multiple PLC backup systems.

In order for the PLC to be able to control the running of the program it must allocate at least 1 register for the rack slot used by the UCM. This first register controls the running of the user program. It is likely that additional registers (up to a total of 2,048) will be assigned to the UCM by the PLC for sharing of data and for providing runtime error codes and program line numbers for debugging of the user program. Figure 1-1 gives a quick visual overview of the modules registers and their accessibility. Note: the UCM1 operates the same as a UCM4 with only port 1 being operational.

There are four separate program memory segments inside the UCM4. Programs that are too large to fit in the space of one memory segment may be placed in consecutive ports upon download. For example if the program for Port 2 is too large to fit in Port 2's program area after compilation, it may be downloaded into Port 2 and Port 3. Of course, Port 3 may not have an operational program but may still be used for SY/MAX communication.

Unused areas of the program memory may be accessed by UCM applications for additional non-volatile storage with the READ PROGRAM and WRITE PROGRAM instructions. This may be quite useful for small data bases and recipe storage.

This manual provides information on the UCM programming language and utilities, and installation and operation of the module.

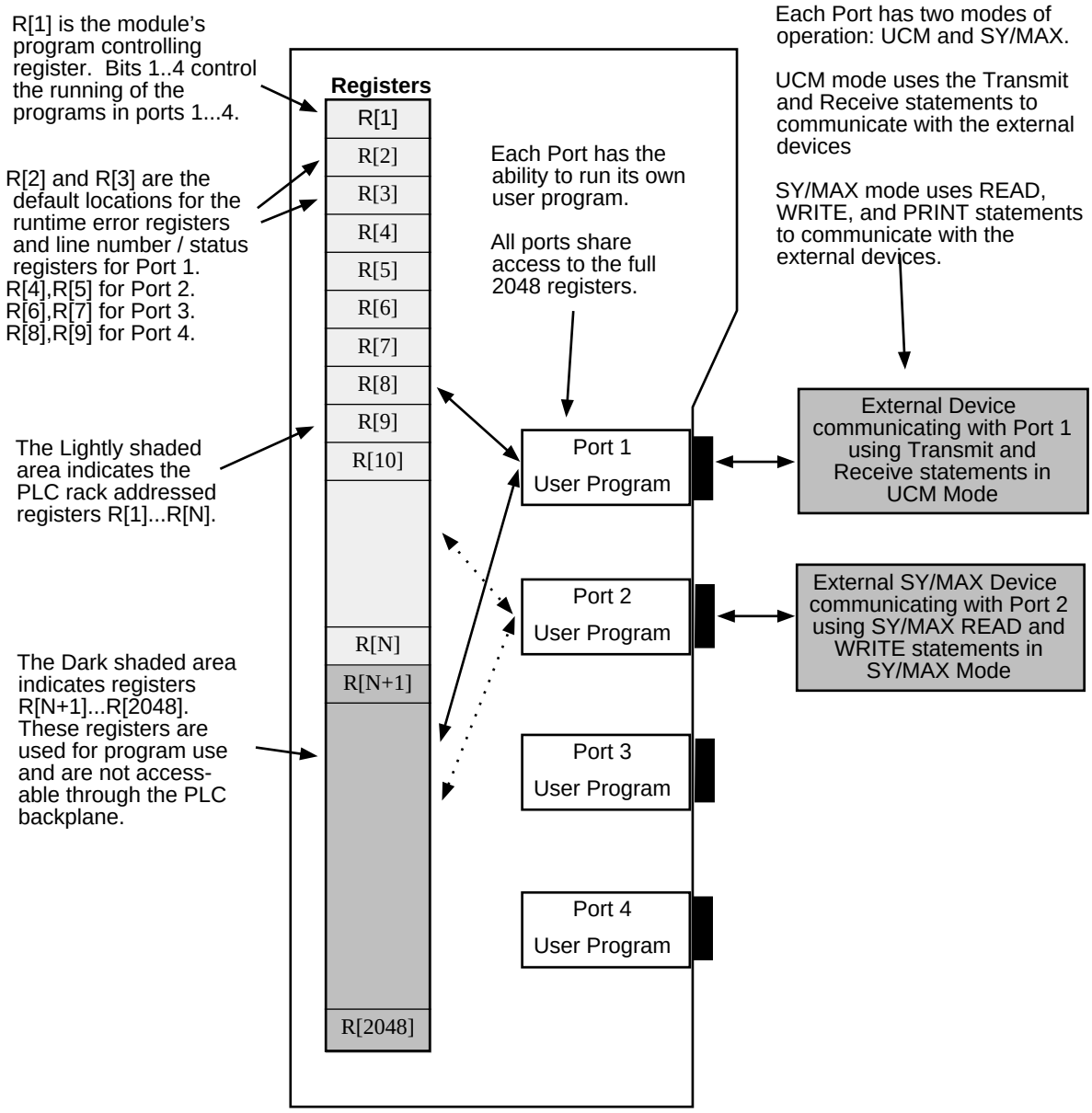
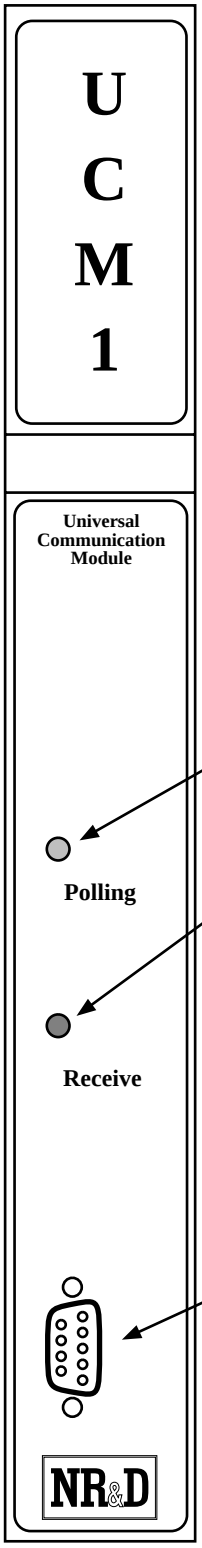


Figure 1-1 UCM Module Overview



Status Indicators (LEDs)

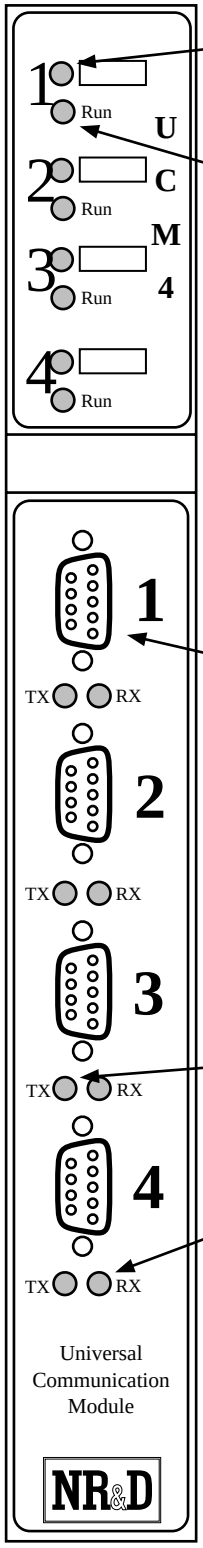
Polling (Green) - When on, the UCM1 is running its user program.

Receive (Red) - When on, the UCM1 is receiving data from the external device.

Communication port

RS-422 or RS-485 Serial Differential Port for connecting to programming and peripheral equipment.

Figure 1-2 UCM1 Front Panel



Program Indicator Lights

User Light - RED user programmed light accessed in the program by SET LIGHT ON, SET LIGHT OFF, or TOGGLE LIGHT. A blank area on the label is provided for a custom description of the RED light.

Run - GREEN Run light is illuminated while the program is actually running.

Communication Ports

Male RS-232 - UCM4-S
Female RS-422 - UCM4-D
Female RS-485 - UCM4-M

Ports 1-4
Serial communication ports for connection of peripheral devices desiring input and output. All ports feature full RTS/CTS hardware handshaking, and support all available modes.

Communication Indicators (LEDs)

TX (Yellow) - Lights when data is transmitted from port.

RX (Yellow) - Lights when data is received at port.

Figure 1-3 UCM4 Front Panel

2
Installation

1 Ensure that the power supply on the register rack will support the current draw of all modules in the rack, including the 1.5 amps drawn by the UCM.

2 Mount the UCM in an available slot in the register rack

3 Connect your PLC programmer to the PLC.

4 Apply power to the rack. The top lights on the UCM1 and UCM4 will strobe during startup. The yellow RX/TX lights beneath ports 1-4 on the UCM4 and above the port on the UCM1 illuminate only when data is passing through the port.

5 With the PLC programmer, allocate registers for the UCM module. At least one register is needed to access the UCM control register, more registers may be needed for the user programs.

Note: The scan time of the processor is directly related to the number of externally addressed registers. Allocating more registers than needed can have an adverse impact on the speed of the system. See the manual for the processor for more information on optimizing scan speed.

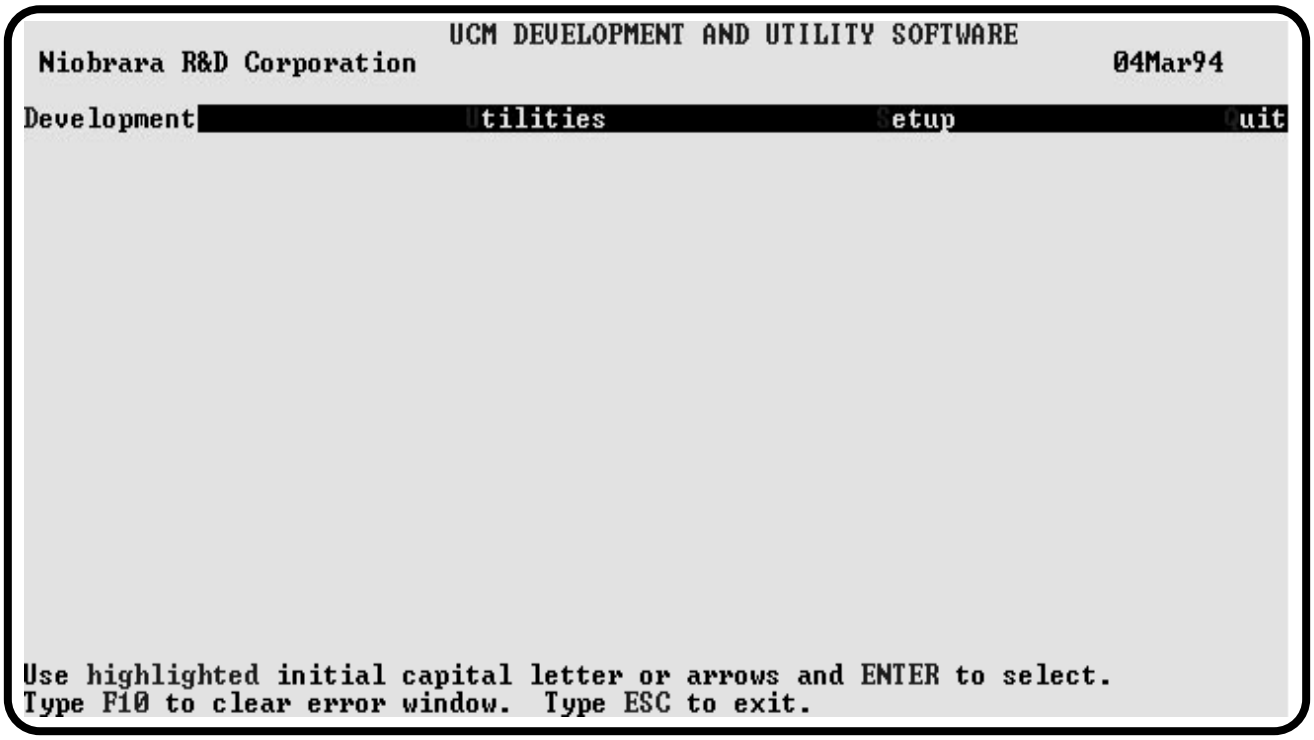
6 Install the UCMSW configuration software from the distribution diskette. Insert the floppy into a drive and run the INSTALL program from that drive. Select UCMSW from the list of softwares to install. It is recommended that a directory UCM be used for the installation on your hard drive. Once INSTALL is finished copying and expanding the files, start up the UCMSW with the following commands:

C:

CD \UCM

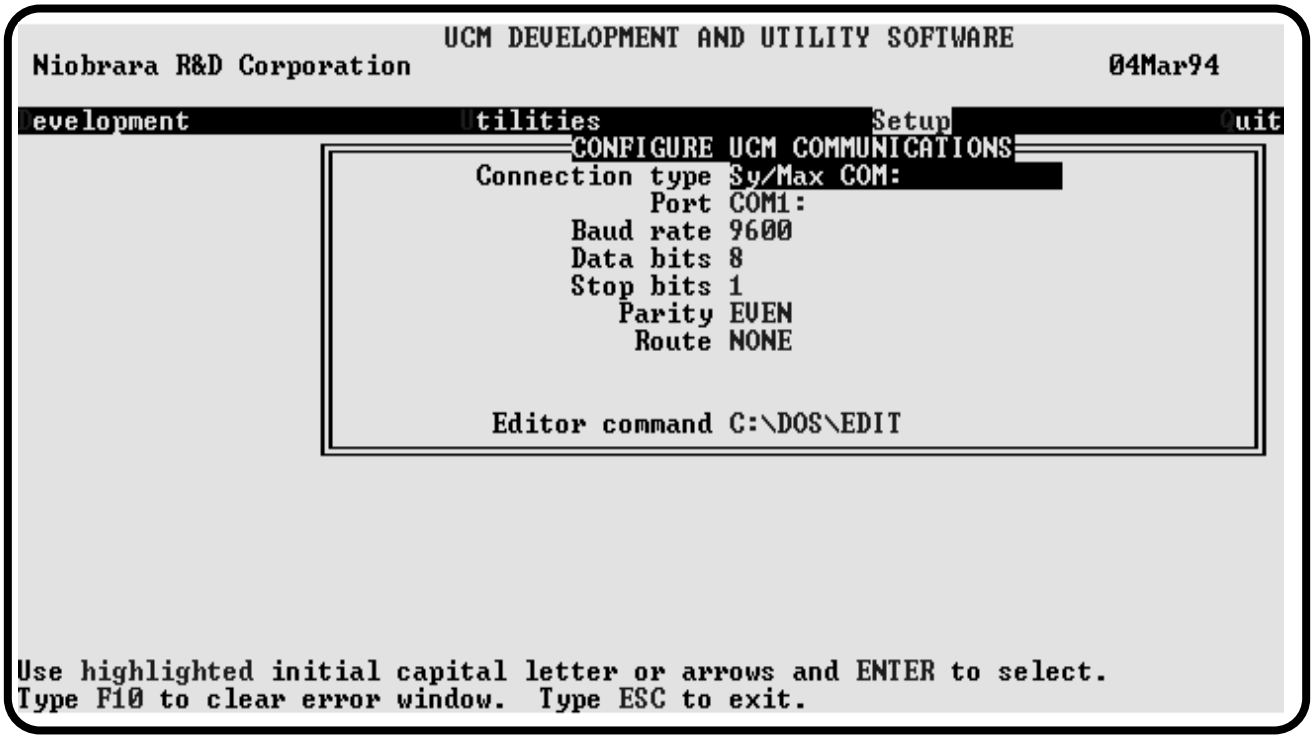
UCMSW

The following screen should appear:



7 Select S for Setup and choose the proper setting for the SY/MAX connection. The default settings for the ports on the UCM are SY/MAX mode, 9600 baud, 8 data bits, EVEN parity, and 1 stop bit.

8 Connect your personal computer to the UCM using the connections specified in the SY/MAX Setup Screen. To test this connection, select Utility then Register View. You should see a screen similar to below:



9 To write UCM programs on your PC that implement the communications protocol of your devices, select D for Development and D for eDit. See chapters 4 through 8 for UCM language definitions, syntax, and syntax examples and Appendix A through E for demo programs. Remember to save the file as you exit from your editor.

10 Use the COMPILE selection (Chapter 9) to compile your UCM source code (UCM$WORK.UCM) into UCM executable code (Filename.UCC).

11 Select Yes when prompted to download the compiled program into the UCM. Select the port that the program will run on.

12 Attach one or more devices to the UCM ports. Refer to Chapters 12 and 13 for port and cable pinout diagrams. An RS-232 or RS-422 breakout box is useful in verifying the correct connection of input devices.

13 Turn on your programs by setting the correct bits of the command register of the UCM from the PLC. The UCM is now running your programs.

3

Quick Start

This chapter is for getting a quick start on operating the UCM. It takes the user through three simple UCM programs to "get the feel" of the module. It is assumed that the UCM software has been loaded onto the personal computer in the C:\UCM directory. It is also assumed that the DOS EDIT text editor is used in these examples. Of course the user may use whatever editor he/she wishes.

These examples work best if a personal computer for programming the UCM and a separate ASCII terminal are available. If an ASCII terminal is not available, a second personal computer running a terminal emulator (such as the terminal emulator in UC-MSW) will work just fine.

This example setup is shown in Figure 3-1 for the UCM4-D. For the UCM4-S, use proper RS-232 cabling. For the UCM1, the personal computer will have to be switched between the PLC and port 1 of the UCM when loading the program.

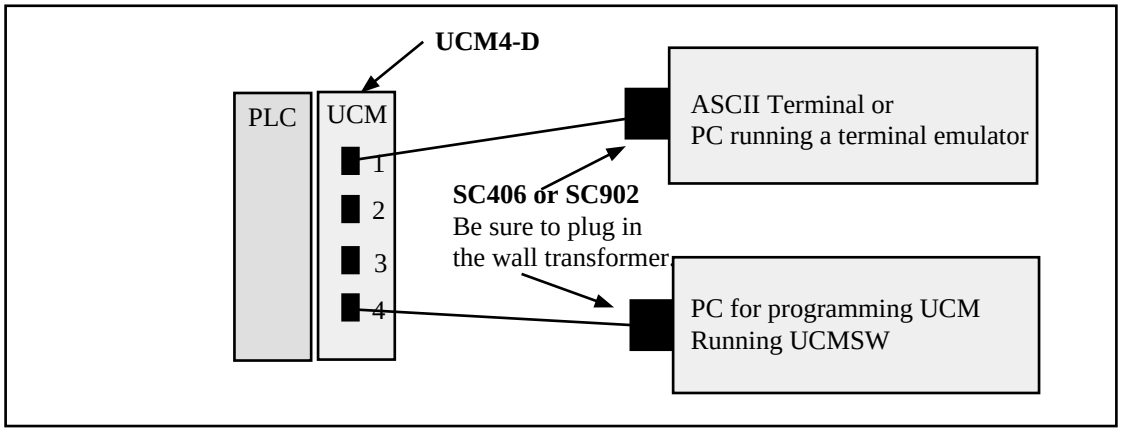


Figure 3-1 Quick Start Setup

HELLO.UCM

To quickly have the UCM do something, perform the following tasks:

1 Change to the UCM directory using the following command:
> CD \UCM

2 Start the UCMSW development software by entering:
> UCMSW

3 Select Development from the main menu.

4 Select Read source from disk.

5 Type HELLO into the File to read and press Enter. What happens here is that the text file HELLO.UCM has been copied to the file named UCM$WORK.UCM. This is the file that is edited during the course of working in the UCMSW environment. Before the UCMSW environment is left, you should select Development, Write to disk to actually save the work under the HELLO.UCM filename.

6 Select Development again.

7 Select D for eDit to edit the UCM file. This will call up the text editor selected in the SY/MAX setup. It is assumed that the MS-DOS EDITOR is being used. The screen should look like Figure 3-2.

Figure 3-2 MS-DOS EDIT of HELLO.UCM

8 Examine the program. The top portion sets up the parameters for the port. The TRANSMIT statement will send the word Hello, followed by a carriage return (0D hex) and a line feed (0A hex). The program will end after the transmit is performed.

9 Exit from EDIT by pressing Alt key and the F key at the same time to show the FILE menu. Press the X key to exit.

10 Compile the HELLO.UCM program by selecting Development Compile. If no errors occur during the compile, a file called HELLO.UCC will be generated. This file is the code that is loaded into the UCM.

11 When the compile is successful, a CONFIRM window will open and ask if you want to download the program into the UCM. Select Y for yes.

12 A DOWNLOAD WORK FILE window will appear. Enter the Module channel for the program to be loaded. (1 for this example) Press Enter to accept the value of 2 for the Status Register and press Enter to accept NO for the Auto Start.

13 UCMSW will now attempt to download the compiled code into the UCM. If an ERROR window appears, a communication setup problem may exist. Check the cabling and SETUP. Press F10 to clear any errors. Remember, the external power supply must be used to connect an SC406 or SC902 to one of the ports on a D or M module.

If the download is successful, the main menu will appear on the screen.

14 Set up the TERM by running UCMSW and selecting the Terminal Emulator from the Utility menu. Be sure to have the proper setup selected in the Setup, Terminal emulator screen.

15 From the programming UCMSW Main menu, select Utility, View Registers. This will show the values in the registers in the module. The screen should appear as shown in Figure 3-3.

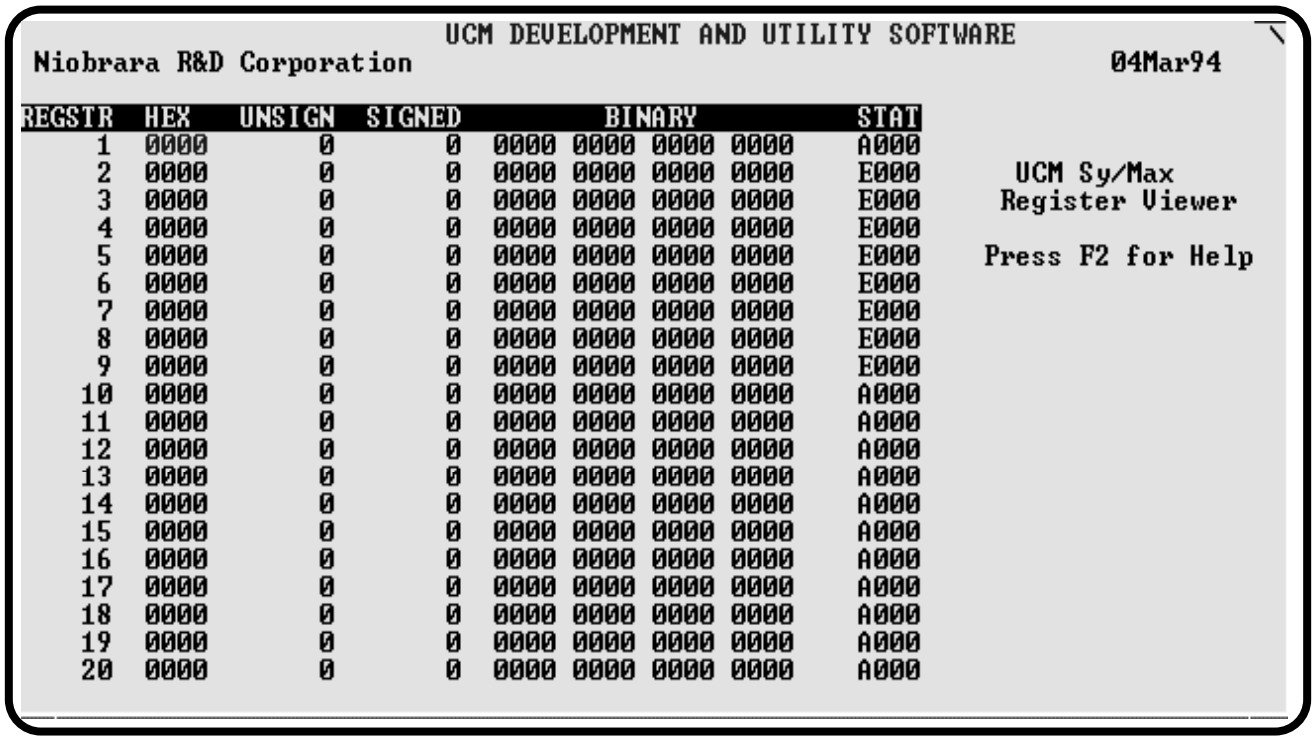


Figure 3-3 Register Viewer

16 To start the program that was just loaded into Port 1, it is necessary to set bit 1 in register 1. With the cursor located on one of the data fields for register 1, Press 1 Return. Several things will happen now:

a. R[1] will change from a PLC output (A000 hex) to a PLC input (E000 hex). This means that the PLC may now only read this register and may no longer write data to it. The reason that this register changed status is because it was written to by the an external source (or the UCM). Once a register has been changed to E000 it will remain in that state until power is cycled on the UCM.

b. R[3] changes from 0 to 8000 (hex) to indicate that the program on port 1 is running without errors.

c. "HELLO<0D><0A>" is transmitted from port 1 to the TERM. This string should appear on the TERM. The TX light on port 1 should flash as this string is transmitted.

d. The program then waits 1 second at the DELAY 100 statement. This delay is used so the user may see the transition of R[2] from 0 to 8000 and back to 0. Without the delay, the program ends too quickly to observe this transition.

e. The program ends at the STOP statement.

f. As the program ends, R[2] changes to 1 to indicate that the program has stopped without error. R[3] changes to 7 to indicate that the program stopped on line 7 of the source code. The View register screen should appear as in Figure 3-4.

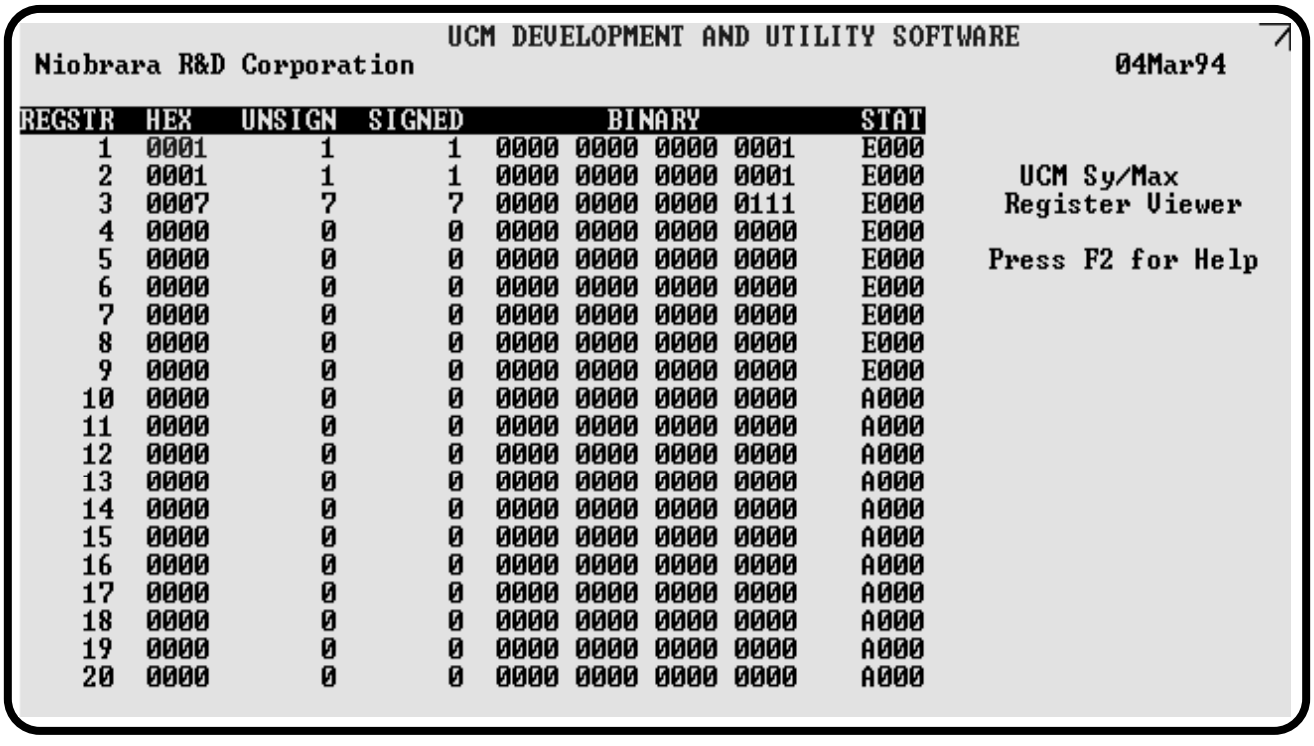


Figure 3-4 Register Viewer after running the program

17 To restart the program it is necessary to clear the command bit for the program and then set the bit again. Using the register viewer, enter a 0 in R[1]. Now enter a 1 in R[1] to start the program again.

18 The TERM should now display something similar to Figure 3-5

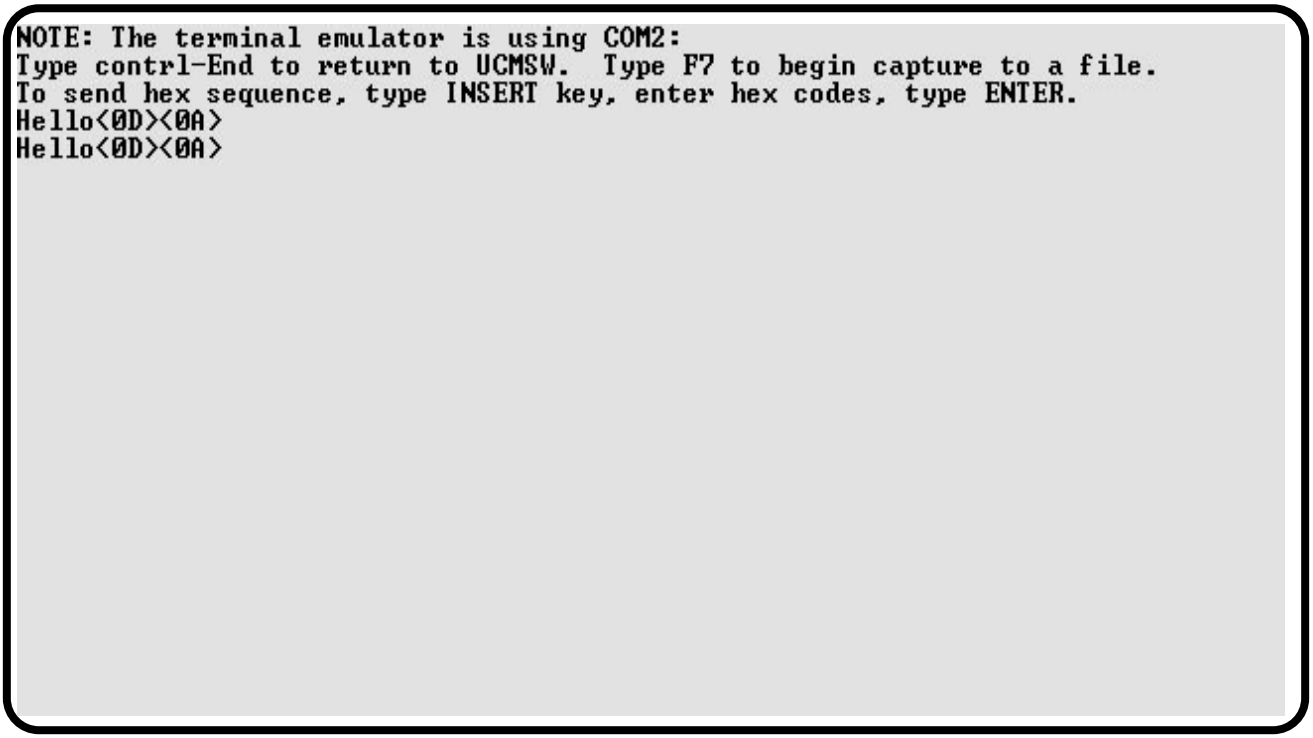


Figure 3-5 TERM screen after running the HELLO.UCM program.

DEC.UCM

For something a little more interesting, try the example program DEC.UCM. From the main menu of UCMSW, select Development. Select Read source from disk and choose the file DEC. Select eDit to view the file; it should appear as in Figure 3-6.

This program provides a somewhat more typical application of the features of the UCM programming language. As in HELLO.UCM, the first four lines featuring the SET statement determine the operating parameters of the port.

Two labels are used in this example. (Start: and Good_Receive:) Labels are alpha-numeric strings that must start with a alpha character, contain no spaces, and end with a colon. Labels are used provide a target for program flow control statements like GOTO and GOSUB.

The flow of the program is as follows:

1 The control bit for the port is set, starting the program.

2 The UCM port is configured as per the SET commands.

3 The string "Enter 3 digits followed by ENTER key.\0D\0A" is sent.

4 The program waits indefinitely until three digits and a carriage return are received. Note: Data in registers defined in the ON RECEIVE is not valid until the entire ON RECEIVE is matched.

5 Once the ON RECEIVE is matched, flow is sent to the label Good_Receive:.

6 The string "Received this number: ":DEC(R[10],3):"\0d\0a" is sent.

7 Register 11 is set equal to Register 10 plus one.

8 The string "Modified to be this number: "DEC(R[11],3):"\0d\0a"

9 Flow returns to START (Step 3 above).

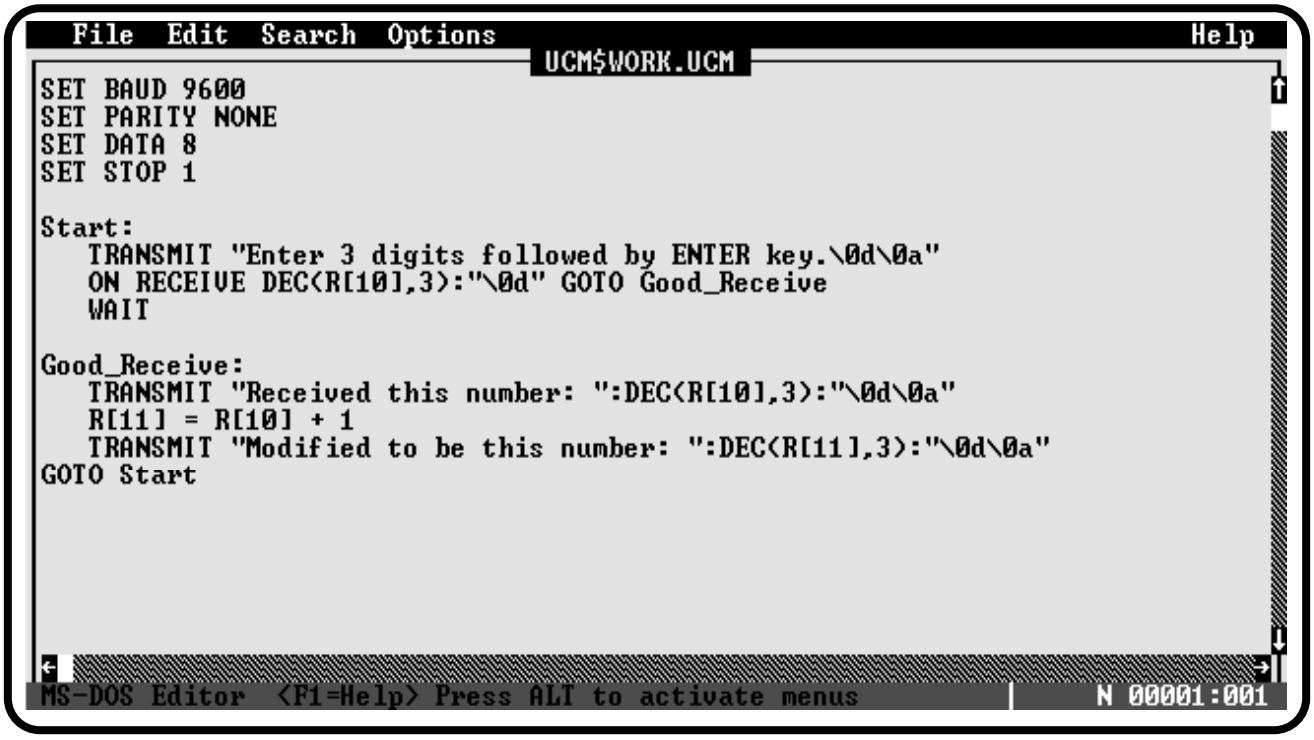


Figure 3-6 MS-DOS EDIT of DEC.UCM

Compile this program, download it into port 1 of the UCM, and run it by setting bit 1 in R[1]. After a few operations, the TERM should look something like the Figure 3-7.

Notice the values in R[10] and R[11] during operation of this program with the Register Viewer. Enter the values 12345 followed by a carriage return. The expected value in R[10] when the <0D> is received and the ON RECEIVE is matched will be 345. If close attention is paid the values 123, 234, and 345 will appear in R[10] as the characters are entered. **But the only the 345 value is correct.** ***This fact should be remembered for all applications.*** See Figure 3-8.

The UCM is a very powerful computer and these examples have just scratched the surface of its capabilities. It is hoped that the user will take these example programs and "play" with them gain a better understanding of how other functions in the UCM work.

```
NOTE: The terminal emulator is using COM2:
Type contrl-End to return to UCMSW.  Type F7 to begin capture to a file.
To send hex sequence, type INSERT key, enter hex codes, type ENTER.
Enter 3 digits followed by ENTER key.<0D><0A>
Received this number: 123<0D><0A>
Modified to be this number: 124<0D><0A>
Enter 3 digits followed by ENTER key.<0D><0A>
Received this number: 524<0D><0A>
Modified to be this number: 525<0D><0A>
Enter 3 digits followed by ENTER key.<0D><0A>
Received this number: 456<0D><0A>
Modified to be this number: 457<0D><0A>
Enter 3 digits followed by ENTER key.<0D><0A>
Received this number: 222<0D><0A>
Modified to be this number: 223<0D><0A>
Enter 3 digits followed by ENTER key.<0D><0A>
Received this number: 345<0D><0A>
Modified to be this number: 346<0D><0A>
Enter 3 digits followed by ENTER key.<0D><0A>
```

Figure 3-7 Terminal output of DEC.UCM

```
                        UCM DEVELOPMENT AND UTILITY SOFTWARE
 Niobrara R&D Corporation                                          04Mar94

REGSTR  HEX   UNSIGN  SIGNED        BINARY          STAT
     1  0001       1       1  0000 0000 0000 0001  E000
     2  8000   32768  -32768  1000 0000 0000 0000  E000      UCM Sy/Max
     3  0000       0       0  0000 0000 0000 0000  E000     Register Viewer
     4  0000       0       0  0000 0000 0000 0000  E000
     5  0000       0       0  0000 0000 0000 0000  E000    Press F2 for Help
     6  0000       0       0  0000 0000 0000 0000  E000
     7  0000       0       0  0000 0000 0000 0000  E000
     8  0000       0       0  0000 0000 0000 0000  E000
     9  0000       0       0  0000 0000 0000 0000  E000
    10  0159     345     345  0000 0001 0101 1001  E000
    11  015A     346     346  0000 0001 0101 1010  E000
    12  0000       0       0  0000 0000 0000 0000  A000
    13  0000       0       0  0000 0000 0000 0000  A000
    14  0000       0       0  0000 0000 0000 0000  A000
    15  0000       0       0  0000 0000 0000 0000  A000
    16  0000       0       0  0000 0000 0000 0000  A000
    17  0000       0       0  0000 0000 0000 0000  A000
    18  0000       0       0  0000 0000 0000 0000  A000
    19  0000       0       0  0000 0000 0000 0000  A000
    20  0000       0       0  0000 0000 0000 0000  A000
```

Figure 3-8 Register View of DEC.UCM

4

UCM Language Definitions

The UCM language is its own unique structured language although the user will probably notice similarities with BASIC, PASCAL, and C. Labels are used to control program flow. Line numbers are not required. The following definitions apply through this manual:

Constant Data Representation - <const>

If numeric data is to remain the same during the entire operation of the UCM program then they should be treated as constants. The UCM supports unsigned decimal integers, signed decimal integers, hexadecimal integers, boolean constants, and a few reserved constants. The use of a constant is referred to as <const> in this manual.

Table 4-1 Constant Data Types

Constant Data Type	Range	Prefix Symbol
Decimal	0...65,535	NA
Signed Integer	-32768...32767	NA
Hexadecimal Integer	0...FFFF	x
Boolean Constants	TRUE, FALSE	NA
Reserved Constants	EVEN,ODD,NONE	NA

Decimal Integers

Decimal integers are defined as the unsigned whole numbers within the range from 0 through 65,535. The following are examples of decimal integers:

0
32114
59
65311

Signed Integers

Signed integers are defined as the whole numbers within the range from -32768 through 32767. The following are examples of signed integers.

```
-514
0
31
-1
```

Hexadecimal Integers

Hexadecimal integers are defined as the hexadecimal representation of the whole numbers within the range from 0 through FFFF. Hexadecimal numbers are defined by the prefix x. The following are examples of hexadecimal constants:

```
x12AB
xf34c
x15
```

Boolean Constants

There are two predefined boolean constants: TRUE and FALSE. The following are valid uses of the boolean constants:

```
SET CAPITALIZE FALSE
SET DEBUG TRUE
```

Reserved Constants

The following constants are reserved for the use in the SET PARITY statement: EVEN, ODD, and NONE. The following are valid uses of the reserved constants:

```
SET PARITY EVEN
SET PARITY ODD
SET PARITY NONE
```

Variable Data Representation - R[<expr>] and $

There are two types of variables used in the UCM: the special variable $ and register variables. The $ variable is a special variable which contains the current index position in a message description. All other variable data in the UCM language is stored in registers. There are 2,048 SY/MAX processor equivalent registers available for variable storage. The syntax R[<expr>] refers to these registers and as <expr> implies, the subscript may be an expression that evaluates to a constant. The valid range for these registers is R[1] through R[2048]. The following are valid variable registers.

```
R[345]
R[(12 +  3 )/ 5]
R[R[1245]]   (The value of R[1245] must fall within the range 1...2,048.)
```

These registers, or a portion of these registers, may be rack addressed by the PLC. Only those registers which are included in the rack address are able to be read or written by the PLC through the backplane. For instance, the PLC has rack addressed the slot for the UCM for registers 251 through 300. These registers will correspond to the internal registers of the UCM 1 through 50. The PLC will not be able to "see" regis-

ters 51 through 2,048 within the UCM module. Each register that is rack addressed by the PLC increases the scan time of the processor; therefore it is good programming practice to only rack address the minimum number of registers that the PLC must use for the particular application. Use the registers above the rack addressed space for miscellaneous variable storage.

UCM4 users NOTE: The 2,048 registers in the module are shared by all programs running within the module. A program on port 1 may modify registers used by the program on port 3 and so on. This is a very powerful feature of the UCM to allow sharing of data between applications, but care must be taken to prevent unwanted data manipulations by other ports. Again it should be noted that all PLC rack addressed registers should be located in registers 1...n and should be kept to a minimum.

These registers, R[1...2048], may also be accessed by external devices such as another PLC or a personal computer using a register viewer through one of the front panel ports on the UCM which is in SY/MAX mode.

The status of these registers as PLC inputs and PLC outputs is determined by the UCM. Upon power-up, all registers are PLC outputs until they are written to by the UCM user program or by an external device connected through one of the ports. Once a register has been written by the UCM or external device it is considered to be a PLC input until the power is removed from the module.

There are 3 reserved registers in the UCM1 and 9 reserved registers in the UCM4 which may not be used by the user program for variable storage. These registers provide the command bits to start the user programs and provide information on the error status and the line number on which the program is stopped.

Table 4-2 Reserved Registers

Register	Description	UCM Port	Normal Status	Notes
R[1]	Command register		PLC Output	
R[2]*	Error Code	Port 1	PLC Input	
R[3]*	Line Number	Port 1	PLC Input	
R[4]*	Error Code	Port 2	PLC Input	UCM4 ONLY
R[5]*	Line Number	Port 2	PLC Input	UCM4 ONLY
R[6]*	Error Code	Port 3	PLC Input	UCM4 ONLY
R[7]*	Line Number	Port 3	PLC Input	UCM4 ONLY
R[8]*	Error Code	Port 4	PLC Input	UCM4 ONLY
R[9]*	Line Number	Port 4	PLC Input	UCM4 ONLY

* - These are the default location of these registers. They can be relocated using the download command..

R[1] Command Register

Register 1 is the command register for the UCM. Bits 1 through 4 determine the operation of ports 1 through 4 on the UCM4. (Only bit one has meaning on the UCM1.) The command register is always register 1 of the UCM and cannot be moved.

When the bit for the port is 0 that port is in SY/MAX mode. In this mode compiled UCM programs may be downloaded into the module from the personal computer and

the internal registers may be viewed or modified by external SY/MAX compatible devices.

When the bit for the port is set to 1, that port is running the user program.

Run Time Error Registers

The run time error registers indicate the status of the user program The default run time error registers are: R[2] port1, R[4] port 2, R[6] port 3 and R[8] port 4. These registers may be moved using the Load utility within UCMSW. The run time error registers always directly follow the associated line number registers. If a run time error has occurred, the following values will be placed in the error register for the specific port.

Table 4-3 Run Time Errors

Error Value	Definition of Error
0	Halted by clearing RUN bit
1	Halted by STOP or RETURN statement
2	Execution of invalid instruction (Program corrupt)
3	Division by zero
4	No memory for ON CHANGE
5	No memory for ON RECEIVE
6	Illegal run time call (module version does not support compiler)
7	Value out of bounds (register < 1 or > 2048, buffer index out of range, SET parameter bad, output/input too long (>256), width specification < 0 or > 64)
8	Checksum error in downloaded code

Line Number Registers

The line number registers indicate the user program line number which was being executed at the time that the program was halted. This halt may occur when the command bit is zeroed or when a run time error has occurred. The default registers for each port is: R[3] port 1, R[5] port 2, R[7] port 3 and R[9] port 4. These registers may be moved using the Load utility within UCMSW.

Program Area Registers

Only the registers within the range of R[1] through R[2048] are directly accessible to the UCM application program. The UCM also contains four large areas of memory for program storage. UCM registers R[2049] through R[7049] provide access to each of these four areas. Register 2049 provides a pointer to determine which Port's program area is being accessed by the external device. The actual program is stored in registers 2050 through 7049. Only one program area may be accessed by an external device at a time, but the running applications have access to all program areas simultaneously.

The UCM applications may access the program areas by the use of the READ PROGRAM and WRITE PROGRAM instructions. These instructions allow access to large blocks of registers for non-volatile storage. Unlike the normal user register 1-2048, the program areas are retained during power loss.

NOTICE: Great care should be exercised when accessing the program areas since altering a program may cause the program to halt or other programs within the UCM to halt and not be able to restart without reloading the entire program.

UCM programs are downloaded starting at register 2051. The length of the program is stored in register 2050 and is expressed in bytes. To determine the area above the application available for user use simply use the following formula:

Starting Point = R[2050]/2 + 2050

For more information on READ and WRITE PROGRAM see pages 45 and 51.

Arithmetic Expressions - <expr>

Numeric expressions, referred as <expr> in this manual, involve the operation of variables and constants through a precedence of operators and functions.

Numeric Operators

Table 4-4 Numeric Operators

Numeric Operator	Description	Example
+	Addition	R[25] + 5
-	Subtraction	R[25] - 5
*	Multiplication	R[25] * 5
/	Division	R[25] / 5
%	Modulus	R[25] % 5
&	Bitwise AND	R[25] & x100
|	Bitwise OR	R[25] | x100
^	Bitwise Exclusive OR	R[25] ^ x100
-	Unary Negation	-R[25]
~	Unary Bitwise Complement	~R[25]
()	Parentheses	(R[25] + 5) * 3

Precedence of Operators

The order of precedence of numeric operators is as follows:

1 Sub expressions enclosed in parentheses

2 Unary Negation or Unary Complement

3 *, /, &, %, ^ From left to right within the expression.

4 +, -, | From left to right within the expression.

Numeric Functions

The UCM supports a group of seven checksum calculating functions to be used only within message descriptions:

Table 4-5 Checksum Functions

Function	Description
CRC(<expr>,<expr>,<expr>)	Cyclical Redundancy Check (CCITT Standard)
CRC16(<expr>,<expr>,<expr>)	Cyclical Redundancy Check
CRCAB(<expr>,<expr>,<expr>)	Special CRC16 for A-B applications
LRC(<expr>,<expr>,<expr>)	Longitudinal Redundancy Check by byte
LRCW(<expr>,<expr>,<expr>)	Longitudinal Redundancy Check by word
SUM(<expr>,<expr>,<expr>)	Straight Sum by byte
SUMW(<expr>,<expr>,<expr>)	Straight Sum by word

The first <expr> is the starting index. The next <expr> is the ending index. The last <expr> is the initial value usually 0 or -1.

These additional functions are also provided:

Table 4-6 Additional Functions

Function	Description	Example R[45]=x1234, R[46]=xABCD
MIN(<expr>,<expr>)	Provides a result of the <expr> which evaluates to the smaller of the two expressions.	R[44] = MIN(R[45],R[46]) results in R[44] = x1234
MAX(<expr>,<expr>)	Provides a result of the <expr> which evaluates to the larger of the two expression.	R[44] = MAX(R[45]*xA,R[47]) results in R[44] = x65E0
SWAP(<expr>)	Reversed the byte order of the register.	R[44] = SWAP(R[46]) results in R[44] = xCDAB

Labels - <label>

The UCM supports alphanumeric labels for targets of GOTO and GOSUB functions. The label consists of a series of characters ended with a colon. Labels must start with a alphabetic character, numbers are not allowed as the first character in a Label. Labels may not be the exact characters in a UCM language reserved word. The label TIMEOUTLoop: is valid while TIMEOUT: is not valid.

Logical Expressions - <logical>

The UCM supports the following logical operators and relational operators. These are referred to as <logical> elsewhere in this manual.

Logical Operators

Table 4-7 Logical Operators

Logical Operator	Definition	Example
AND	Result TRUE if both TRUE	IF <expr> AND <expr> THEN
OR	Result TRUE if one or both TRUE	IF <expr> OR <expr> THEN
NOT	Inverts the expression	IF NOT(<expr>) THEN

Relational Operators

Table 4-8 Relational Operators

Relational Operator	Definition	Example
<	LESS THAN	IF <expr> < <expr> THEN
>	GREATER THAN	IF <expr> > <expr> THEN
<=	LESS THAN or EQUAL	IF <expr> <= <expr> THEN
>=	GREATER THAN or EQUAL	IF <expr> >= <expr> THEN
=	EQUAL	IF <expr> = <expr> THEN
<>	NOT EQUAL	IF <expr> <> <expr> THEN

Message Descriptions - <message description>

The <message description> refers to the actual serial data that is transmitted from the UCM port or expected data that is to be received by the port. The <message description> may include literal strings, results of various message functions and the concatenation of the above.

Literal String - <string>

A literal string is a string enclosed in quotes. "This is a literal string."

Literal strings may include hexadecimal characters by form \xx where xx is the two digit hex number of the character. This is useful for sending non-printable characters. "This is another literal string.\0D\0A" will print the message with a carriage return (0D) and a line feed (0A).

Embedded quotation marks may be included in literal strings by the insertion of \" in the location of the embedded quote. "This will print a \"quote\" here."

Embedded \ characters may similarly be inserted by using \\.

Message Functions

The UCM can perform a variety of functions on transmitted and received data. When the UCM is using these functions for transmitting, register data and expressions are turned into strings according to the function's rules. When the UCM is using these functions for receiving, incoming strings are either matched to the strings that the UCM expected to receive or they are translated into data and stored in registers.

The following is a list of message functions, each function is described in more detail on pages 56 through 58.

Table 4-9 Message Functions

Functions	Description
BCD(<expr>)	Binary Coded Decimal conversion
BYTE(<expr>)	Least Significant (low) byte conversion
DEC(<expr>,<expr>)	Decimal conversion (base 10) -32768 to 32767
HEX(<expr>,<expr>)	Hexadecimal conversion (base 16)
IDEC(<expr>,<expr>)	IDEC format hexadecimal conversion
OCT(<expr>,<expr>)	Octal conversion (base 8)
RAW(R[<expr>],<expr>)	Sends/Receives high byte then low byte of a register(s)
RWORD(<expr>)	Sends/Receives low byte of an expression
TON(<expr>)	Turn on translation of one string to another
TOFF(<expr>)	Turn off translation of one string to another.
UNS(<expr>,<expr>)	Unsigned decimal conversion (base 10) 0 to 65,535
WORD(<expr>)	Sends/Receives high byte then low byte of an expression

The message functions that take the form *FUNC*(<expr>,<expr>) use the following rules: When using these functions with TRANSMIT, the first <expr> is the data to be translated and transmitted. When using these functions with ON RECEIVE, replace the first <expr> with R[<expr>] to have the incoming string translated and placed into the register R[] or use (<expr>) to have the expression evaluated and matched to the incoming string. The second <expr> in the these functions is the number of characters either to transmit or to receive. An error will be generated at compile or run time if this expression evaluates to less than zero.

RAW takes the form RAW(R[<expr>],<expr>). In this case the first <expr> is the starting register number and the second <expr> is the number of characters. Always uses the high byte first and then the low byte..

The message functions that take the form *FUNC*(<expr>) have fixed character lengths. BYTE transmits one character, the least significant byte, while WORD and RWORD each transmit two characters. WORD transmits the most significant byte and then the least significant byte while RWORD reverses the order, least significant then most significant. As in the previous message functions, when transmitting use <expr> and when receiving either use R[<expr>] to receive and place in a register or (<expr>) to evaluate and match. For examples of the message functions see Chapter 8 - Examples.

In all of the message functions, only characters from the valid character set for that command can be used.

Variable Fields

The width field of any transmit or receive element (that has a width) may be replaced with either of two constructions. (Transmit RAW is an exception as shown below.) The first is just the word VARIABLE, i.e. TRANSMIT DEC(R[10],VARIABLE). The second is VARIABLE followed by a register reference, i.e. TRANSMIT HEX(R[11],VARIABLE R[10]) which will write the actual width to the specified register

Transmit usage of Variable length

A variable field in a TRANSMIT statement means one encoded with only the necessary number of digits (no leading zeros).

For example, if R[11] = 1234 then

TRANSMIT "$":DEC(r[11], variable R[10]):"#"

would send out the string $1234# and R[10] would have the value 4. If R[11] = 89 then the string $89# would be transmitted and R[10] would equal 2.

This type of transmit structure applies to the BCD, UNS, DEC, HEX, OCT, and IDEC formats. The TRANSMIT RAW variable structure requires a terminator byte of 00 hex at the end of the raw string. The transmit raw variable sends up to but not including the null terminator. The optional count register does not include the terminator in the count.

For example, if R[11]=x486F, R[12]=x7764, and R[13]=x7900 then

TRANSMIT "$":RAW(R[11], VARIABLE R[10]):"#"

would send the string $Howdy# and R[10] would equal 5. If R[12]=x0000 then the string $Ho# would be transmitted and R[10] would equal 2.

On Receive usage of Variable length

A variable field in an ON RECEIVE statement must be followed by a literal field such as "\0d". The first character of the literal field works as a terminator.

For example, A device sends a variable length number with a fixed number of decimal points such as $125.01 or $3.99; the decimal point may be used as a terminator and it could be handled as follows:

ON RECEIVE "$":dec(r[100],variable):".":dec(r[101],2)

In the case of $125.01, register R[100] = 125 and R[101] = 1. For $3.99, register R[100] = 3 and R[101] = 99.

The ON Receive raw variable writes an extra zero byte to the registers following the received data. In the case of an odd number of characters, the last register contains the final character in the MSB and a zero in the LSB. In the case of an even number of characters, all 16 bits of the register following the last two characters are set to zero. This null terminator is not included in the count optionally reported.

For example: A device transmits a variable length error message terminated with a carriage return and line feed.

ON RECEIVE RAW(R[500], variable R[200]):"\0d\0a"

will accept the message and place it in packed ASCII form starting at register 500. Register 200 would hold the number of characters (bytes) accepted in the string not including the carriage return or line feed.

Message Assignments

It is sometimes convenient to apply the message descriptions of a TRANSMIT message and store the message in registers in the UCM rather than transmit the string. This is possibly by simply using the assignment character = to a starting register.

R[<exp>] = <message>

The message will be placed in packed ASCII form starting in register R[<exp>]. Any valid transmit message may be stored in this manner.

For example:

```
R[400] = "Hello!\0d\0a"
```

would result in registers 400 though 403 having the following values:
R[400]=x4865, R[401]=x6C6C, R[402]=x6F21, and R[403]=x0D0A.

Something more obviously useful might be:

```
R[501] = byte(r[35]):"\03":word(r[36]):word(r[37]):rword(crc16(1,$-1,0))
```

which would place the reversed word of the checksum in register in R[504].

5

UCM Language Statements

The UCM language statements are described in this chapter. Statements control the operation of the UCM by determining the flow of the program.

The format of these statements includes the definitions from Chapter 4 - UCM Language Definitions. Whenever one of these definitions is referenced in a statement it is enclosed in brackets <>. For example, whenever a statement requires an expression it will appear as <expr>. The words statement and command are used interchangeably.

The word *newline* means a carriage return, line feed or both, whatever your text editor requires. Most commands do not require newlines but those that do use the word *newline*. Since most commands do not requires newlines, multiple statements can be placed on a single line. A whole program could be written on a single line if no statements that require a *newline* are used. For readability, newlines between statements can be used without penalty.

Also note that, except in strings, capitalization in the UCM program is ignored by the UCM and its compiler. The label Tom: is the same as the label TOM:. In literal strings, which are enclosed in quotes "", the capitalization is maintained by the UCM. The command SET CAPITALIZE can affect the way the UCM handles ASCII characters on transmitting and receiving.

Program flow is sequential, from the first statement to the second statement to the third statement etcetera, unless a program flow control statement is reached. Program flow statements can be jumps (GOTO or GOSUB), loops or conditionals (IF...THEN ...ELSE...ENDIF). After a jump, program flow is still sequential starting with the statement immediately after the label. Loops can be accomplished with FOR...NEXT, REPEAT ...UNTIL, or WHILE...WEND.

Assignments

The UCM language allows for the assignment of values to registers and bits of registers. These assignments are similar to the BASIC LET statement.

R[<expr>]=<expr>

This statement sets the register number specified by the first <expr> to the value obtained by the second <expr>. The valid range of register numbers in the first <expr> is 1 through 2,048. The valid range of the second <expr> is x0000 through xFFFF.

R[<expr>].<const>=<logical>

This statement sets a single bit of a register to be one (TRUE) or zero (FALSE). The <expr> can have the values 1 through 2,048. The <const> can have the values 1 through 16 and the <logical> can have the values TRUE or FALSE.

R[<expr>].(<expr>)=<logical>

This statement sets the bit of a register to be the evaluation of the <logical> segment.

R[<expr>].R[<expr>]=<logical>

This statement sets the bit of a register to be the evaluation of the <logical> segment

R[<expr>]=<message description>

This statement sets the register number specified by the <expr> and the following registers to the packed ASCII values obtained by evaluation of the <message description>. The valid range of register numbers in the first <expr> is 1 through 2,048. The <message description> may be any valid message used in a TRANSMIT command.

BAUD

See **SET BAUD** on page 46.

CAPITALIZE

See **SET CAPITALIZE** on page 46.

CLEAR

CLEAR[<expr>].<const> or **CLEAR[<expr>].(<expr>)**

The CLEAR statement sets a single bit of a register to zero. The register number <expr> is in the range 1 through 2,048. The bit number <const> is in the range 1 through 16. The (<expr>) must evaluate to a number within the range 1 through 16 and must be enclosed in parenthesis. To force a single bit of a register to be set to one use the SET R[] statement.

DATA

See **SET DATA** on page 46.

DEBUG

See **SET DEBUG** on page 46.

DEFINE

DEFINE <macro>=<replacement string> ***newline***

The DEFINE statement is a compiler instruction for a global find and replace. When the UCM program is compiled the compiler finds every string <macro> and replaces it with the the string <replacement string>. Both <macro> and <replacement string> are type <string>. A *newline* is required to define the end of the replacement string. Use of this statement can help the readability of the user program and also make the program easier to write.

DELAY

DELAY <expr>

The DELAY statement forces the UCM to pause in its execution of other instructions until a period of time equal to <expr> times 10 mS has expired. Valid range is 0 to FFFF hex.

DUPLEX

See **SET DUPLEX** on page 46.

FOR...NEXT

The FOR ... NEXT statement provides the ability to execute a set of instructions a specific number of times. The variable R[<expr>] is incremented from the value of the first <expr> to the value of the second <expr>. Once the variable is greater than the second <expr>, control passes to the next program statement following the NEXT. If the optional STEP expression is included, the variable R[<expr>] is incremented by the value equal to the STEP <expr>. If the STEP <expr> is not present a step of 1 is assumed.

FOR R[<expr>]=<expr> TO <expr>
one or more statements
NEXT

FOR R[<expr>]=<expr> TO <expr> STEP <expr>
one or more statements
NEXT

FOR ... NEXT loops may be constructed to decrement from the first <expr> to the second <expr> using the DOWNTO function. The STEP <expr> must be a negative number. If STEP <expr> is not present a step of -1 is assumed.

FOR R[<expr>]=<expr> DOWNTO <expr>
one or more statements
NEXT

FOR R[<expr>]=<expr> DOWNTO <expr> STEP <expr>
one or more statements
NEXT

FOR...NEXT loops may be nested any number of levels.

GOSUB...RETURN

GOSUB <label>

The GOSUB statement turns control of a program to another area of code while expecting to get control back from a RETURN statement. It is useful for program flow control where one section of code may be used several times. Somewhere in the program flow following <label> needs to be a RETURN statement. The RETURN statement returns program control back to the GOSUB statement that caused the jump. After a RETURN the UCM will continue running using the statement immediately following the GOSUB.

GOTO

GOTO <label>

The GOTO statement turns program control over to another area of code.

IF...THEN...ELSE...ENDIF

The IF ... THEN statement is used to control the program flow based upon the logical evaluation of the expression in <logical>. When <logical> is true, the statements following the THEN are executed. If <logical> is false the statements following the ELSE are executed.

IF <logical> THEN one or more statements followed by *newline*

IF <logical> THEN one or more statements ELSE one or more statements followed by a *newline*

When more statements are required for an IF ... THEN, the statements may be placed on additional lines below the IF ... THEN. The ENDIF statement indicates the termination of the IF statement.

IF <logical> THEN *newline*
one or more statements
ENDIF

IF <logical> THEN *newline*
one or more statements
ELSE
one or more statements
ENDIF

LIGHT

See SET LIGHT on page 47.

MOVE

Reserved instruction for a special NR&D motion control application. Must not be used in user application.

MULTIDROP

See **SET MULTIDROP** on page 47.

ON CHANGE

ON CHANGE R[<expr>] GOTO <label>

ON CHANGE R[<expr>] RETURN

ON CHANGE R[<expr>] & <expr> GOTO <label>

ON CHANGE R[<expr>] & <expr> RETURN

The ON CHANGE statement functions within a WAIT loop (like an ON RECEIVE or ON TIMEOUT), and performs the GOTO or RETURN depending upon the result of the value of R[<expr>]. When the value in R[<expr>] is modified by another source, the ON CHANGE statement is performed.

ON RECEIVE

ON RECEIVE <message description> GOTO <label>

ON RECEIVE <message description> RETURN

The ON RECEIVE statement functions within a WAIT loop and performs the GOTO or RETURN depending upon whether the incoming string exactly matches the <message description>.

ON TIMEOUT

ON TIMEOUT <expr> GOTO <label>

ON TIMEOUT <expr> RETURN

The ON TIMEOUT statement functions within a WAIT loop (like an ON RECEIVE or ON CHANGE), and performs the GOTO or RETURN depending upon the elapsed time between incoming characters on the port. The result of the <expr> must fall within the range 0 to FFFF hex. Like the DELAY function, the ON TIMEOUT <expr> waits for a period of time equal to <expr> times 10mS..

PARITY

See **SET PARITY** on page 48.

PRINT (SY/MAX)

PRINT <port> (<drop>,...) <message description> **(static route)**

PRINT <port> R[<expr>] <message description> **(dynamic route)**

The PRINT statement allows a UCM program to generate SY/MAX network print messages from any port on the UCM.

The <port> is an expression which evaluates to a valid UCM port i.e. 1, 2, 3, or 4 for a UCM4 or only 1 for a UCM1. The port number does not have to be the same port that the program is running on. The port which is selected will change to SY/MAX mode, if not already in that mode, and emit the PRINT packet. The port will wait up to 5 seconds for a valid reply from the SY/MAX device before timing out. The reply status will appear in the line number register for the program running port.

There are two types of routing schemes used: static and dynamic.

The **static route** includes the required network drops separated by commas and enclosed in parenthesis. Example: (103, 055). The route will consist of the drop number of the network port that the UCM is connected, any Net-to-Net ports, and finally the network target port. The target port must be a NIM port in Peripheral mode, or an SPE4 port in Peripheral, Transparent, Share, or Gateway mode.

The **dynamic route** includes a register which contains a pointer to another register which contains the number of drops in the route, N. The next N registers following the number of drops register contain the drop numbers for the route. Example: PRINT 1 R[1045] "Hello\0D\0A". If R[1045] contains the decimal value 2040, then the PRINT message will look at R[2040] for the number of drops in the route. If R[2040] = 3 then the value of R[2041] will be the first drop number, R[2042] will be the second drop number, and R[2043] will be the target drop.

The <message description> is the same used in a TRANSMIT statement.

READ (SY/MAX)

READ <port> (route) <local>, <remote>, <count>

READ <port> R[<expr>] <local>, <remote>, <count>

The READ statement allows a UCM program to generate SY/MAX network priority read messages from any port on the UCM.

The <port> is an expression which evaluates to a valid UCM port i.e. 1, 2, 3, or 4 for a UCM4 or only 1 for a UCM1. The port number does not have to be the same port that the program is running on. The port which is selected will change to SY/MAX mode, if not already in that mode, and emit the READ packet. The port will wait up to 5 seconds for a valid reply from the SY/MAX device before timing out. The reply status will appear in the line number register for the program running port.

There are two types of routing schemes used: static and dynamic.

The **static route** includes the required network drops separated by commas and enclosed in parenthesis. Example: (103, 055). The route will consist of the drop number of the network port that the UCM is connected, any Net-to-Net ports, and finally the network target port.

The **dynamic route** includes a register which contains a pointer to another register which contains the number of drops in the route, N. The next N registers following the number of drops register contain the drop numbers for the route. Example: READ 1 R[1045] 5, 6, 8. If R[1045] contains the decimal value 2040, then the READ message will look at R[2040] for the number of drops in the route. If R[2040] = 3 then the value of R[2041] will be the first drop number, R[2042] will be the second drop number, and R[2043] will be the target drop.

The **<local>** is an expression which evaluates to a register number within the range of 1...2048. This value is the register within the UCM where the data from the READ will be placed. If <count> value is greater that 1 then the <local> is the starting register in the UCM for the READ data.

The **<remote>** is an expression which evaluates to a register number within the remote SY/MAX device. This value must be within the valid register range of the

external device, usually within the range of 1...8191. Consult the manual for the external device to determine valid register numbers. If the <count> is greater than 1 then the <remote> value is the starting register of the multiple register read.

The **<count>** is an expression which evaluates to the number of consecutive registers to be read with the READ statement. The default value is 1. The maximum value is typically 128 but may vary with the external device.

EXAMPLE: READ 2 (101, 112) 145, 3084, 25

This READ statement will send a SY/MAX network read out UCM port 2 with a static route into network port 101 and out network port 112. The READ will transfer the data from registers 3084 through 3109 in the external device to registers 145 through 170 of the UCM.

READ PROGRAM

READ PROGRAM <port> <local>, <remote>, <count>

The READ PROGRAM statement allows a UCM program to read a group of registers from the program area of one of the UCM's ports to the user register area.

The <port> is an expression which evaluates to a valid UCM port i.e. 1, 2, 3, or 4 for a UCM4 or UCM1.

The <local> is an expression which evaluates a number within the range 1 through 2048.

The <remote> is an expression which evaluates to a number within the range 2050 through 7049.

The <count> is an expression which evaluates to a number within the range 1 through 128.

EXAMPLE: READ PROGRAM 3 50,3478,33

would copy 33 registers from Port 3's program area, register 3748 to the user area R[50].

REPEAT...UNTIL

REPEAT
program statements
UNTIL <logical>

The REPEAT statement starts a loop based upon the evaluation of the <logical> condition located in the UNTIL statement. The loop will only be performed as long as the <logical> is TRUE. When the <logical> is FALSE, program execution jumps to the statement following the UNTIL.

Note: The program statements will execute at least once regardless of the condition of <logical>. This is different than the WHILE...WEND or FOR...NEXT loops which will not execute the program statements within their boundaries if the <logical> is FALSE.

RETURN

See **GOSUB...RETURN** on page 42.

SET

The SET statement allows the initialization of the UCM for the following parameters: Baud rate, Capitalization of incoming characters, Data bits, Parity, Stop bits, and Debug mode.

SET BAUD <const>

The SET BAUD statement sets the baud rate of the port for the value. Any decimal value may be chosen for the baud rate. The UCMSW utility may be used to determine the actual baud rate produced by the SET BAUD statement. Example: SET BAUD 9600

SET CAPITALIZE <const>

The SET CAPITALIZE statement performs a translation on incoming ASCII alphabet characters from the lower case to the upper case. Example: SET CAPITALIZE TRUE or SET CAPITALIZE FALSE.

SET DATA <const>

The SET DATA statement sets the number of data bits for the operation of the port. Valid range is 5,6,7, or 8 bits. Example: SET DATA 8

SET DEBUG <const>

The SET DEBUG statement determines the operation of the UCM port in the event of a run time error. If SET DEBUG TRUE is used, the UCM program will halt upon a run time error and display the error number and line number in the appropriate registers. If SET DEBUG FALSE is used, the UCM program will halt upon a run time error and immediately restart the program from the beginning.

SET DUPLEX <const>

SET DUPLEX controls the behavior of the receiver in a -M module and has two modes, HALF and FULL:

SET DUPLEX FULL is the normal mode just like a -D or -S module. In full duplex, the receiver is listening all of the time and data can be received even during a transmit (it is buffered until an ON RECEIVE is executed.) If you would tie the transmit and receive pairs together in this mode, the receiver would receive all transmitted data.

SET DUPLEX HALF is intended for use with half duplex links such as two wire RS-485 or echoing modems, radios, or other DCE. In half duplex mode, the receiver is disabled when there is data to transmit. That is, the receiver is turned off and RTS is turned on at the start of a TRANSMIT statement and stays disabled until the last stop bit of the last character transmitted is emitted. At the end of the transmission, RTS is negated and the receiver is re-enabled. Any receive data buffered before the transmission is still in the buffer.

RTS behavior and CTS behavior are unaffected by SET DUPLEX. SET DUPLEX HALF is only effective when a port is in UCM mode. In SY/MAX mode (either because the program is stopped or because of executing a READ, WRITE, PRINT, or SET MODE SYMAX) the duplex operation is full. If a TRANSMIT, RECEIVE, or SET MODE UCM is subsequently executed on the SY/MAX mode

port, the duplex mode in effect before the change to SY/MAX mode is restored. At the beginning of a program, the duplex mode is full. Executing a SET DUPLEX HALF on a -D module results in a run time error 7 (value out of bounds). SET DUPLEX FULL can be executed by a -D and there is no effect.

SET LIGHT <const>

The SET LIGHT statement is used to determine the state of the RED indicator light for the port. SET LIGHT ON turns on the light while SET LIGHT OFF turns off the light. See also TOGGLE LIGHT on page 48.

SET MODE <const>

The SET MODE statement determines the operating mode of the port. Valid entries are UCM or SYMAX. UCM mode allows the use of the TRANSMIT and RECEIVE statements to communicate with the external device. SYMAX mode allows the use of READ, WRITE, and PRINT statements to communicate with external SY/MAX devices. The default mode is UCM if no mode is selected.

SET MULTIDROP <const>

SET MULTIDROP controls the transmit driver disable in the -M module. (The -D module transmit driver is always enabled.) It has two settings, TRUE and FALSE.

SET MULTIDROP FALSE causes the transmit driver to be enabled continuously (just like a -D). The CTS pair controls the transmitter restraint only.

SET MULTIDROP TRUE causes the transmit driver enable to be controlled by CTS. When CTS is active, the driver is enabled (and the transmitter is unrestrained). When CTS is inactive, the transmit driver is disabled and the TX+ and TX- leads are placed in a high-impedance state suitable for paralleling with other transmitters.

Like multidrop false mode, CTS inactive also restrains the transmitter and will freeze a TRANSMIT statement. Unlike SET DUPLEX, the state of SET MULTIDROP is effective in SY/MAX mode as well as UCM mode. This is intended to alleviate multidrop contention from modules that halt and enter SY/MAX mode in some cases. Note that RTS is still asserted in SY/MAX mode and if tied to the CTS pair, will enable the transmitter. When a channel on a -M module is stopped or at the beginning of a program SET MULTIDROP TRUE is in effect. Therefore it is necessary to include an explicit SET MULTIDROP FALSE at the beginning of any program for a -M module that does not need the transmit driver disabled. RTS should be looped to CTS fro normal SY/MAX operations on a -M module (i.e. use an SC406, SC902, DC1, or CC100 as usual). It should be noticed that most applications will have RTS tied to CTS which will defeat the above settings (in SY/MAX mode the driver will be enabled).

Executing SET MULTIDROP TRUE on a -D module will result in a run time error 7 (value out of bounds). Note that the default behavior of the -M , which is SET MULTIDROP TRUE, is different than the -D which has no multidrop capability. This is the only difference in the default behavior between the two modules.

SET PARITY <const>

The SET PARITY statement determines the parity of the port. Valid entries are EVEN, ODD, or NONE. Example: SET PARITY EVEN

SET STOP <const>

The SET STOP statement determines the number of stop bits for the port. Valid entries are 1 or 2. Example: SET STOP 2

SET TIMER R[<expr>] <const>

The SET TIMER statement sets a timeout value into an internal UCM register, which may be tested at a later time to check whether the <const> amount of time has elapsed. The <const> amount of time is in 1/100ths of a second, thus
SET TIMER R[100] 100
would load a value into R[100] which would make that timer expire 1.00 seconds after the SET TIMER instruction was executed. The timer register may be tested any time after the SET TIMER command, using the EXPIRED(R[<expr]) function, which returns FALSE if the timer has not expired (the <const> amount of time has not elapsed), or TRUE if the timer has expired (the <const> amount of time, or more, has elapsed).

SET (bit)

SET R[<expr>].<const> or SET R[<expr>].(<expr>)

The SET statement sets a single bit of a register to ONE. The register number <expr> is in the range 1 through 2,048. The bit number <const> is in the range 1 through 16. The (<expr>) must evaluate to a number within the range 1 through 16 and must be enclosed in parenthesis. To clear a single bit of a register to be set to one use the CLEAR R[] statement.

STOP

The STOP statement causes the UCM program to halt upon its execution. The program may be restarted by clearing and then setting the command bit for the program.

STOP (BITS)

See **SET STOP** on page 48.

TOGGLE

TOGGLE R[<expr>].<const> or TOGGLE R[<expr>].(<expr>)

The TOGGLE statement changes the state of a single bit of a register. The register number <expr> is in the range 1 through 2,048. The bit number <const> is in the range 1 through 16. The (<expr>) must evaluate to a number within the range 1 through 16 and must be enclosed in parenthesis. To force a single bit of a register to be set to zero use the CLEAR R[] statement. To force a single bit of a register to be set, use the SET R[] statement.

TOGGLE LIGHT

TOGGLE LIGHT

The TOGGLE LIGHT statement changes the state of the RED user light for the port that the program is running on. If the light was off when the toggle light statement is executed then the light will be turned off. If the light was on then the light will be turned off. See also the SET LIGHT command on page 47.

TRANSLATE

TRANSLATE <const>:<string1>=<string2>

Translate assigns the function of translating the message of string2 to string1 to a constant number between 1 and 8. The Translate statement is used in conjunction with TON(<const>) and TOFF(<const>). Example: TRANSLATE 1: "\10\10" = "\10"

The string being received is modified whenever <string1> is encountered so that <string2> replaces string 1. Suppose TRANSLATE 2: "LL" = "L" is defined. The UCM port receives the string "HELLO". If the translation is activated for the entire string, ON RECEIVE TON(2):RAW(R[10],4):TOFF(3) RETURN,the string processed by the ON RECEIVE would be "HELO".

The translation works in the opposite direction for TRANSMIT. If the above translation is used, TRANSMIT TON(2):"YELOW":TOFF(2) would result in the sending of the string "YELLOW".

The translation has the following effects upon the checksum calculations: TRANSMIT, the checksum is calculated based upon the pre-translation message; ON RECEIVE, the checksum is calculated based upon the post-translation message.

TRANSMIT

TRANSMIT <message description>

The TRANSMIT statement allows serial communication to be emitted from the port. The exact string evaluated from the <message description> will be emitted.

WAIT

The WAIT statement follows a group of ON RECEIVE, ON CHANGE, and ON TIMEOUT statements. The WAIT statement causes a loop to occur until one of the ON RECEIVE, ON CHANGE, or ON TIMEOUT conditions has occurred. Program flow will be directed by the ON RECEIVE, CHANGE, or TIMEOUT statement.

WHILE...WEND

WHILE <logical>
program statements
WEND

The WHILE statement starts a loop based upon the evaluation of the <logical> condition. The loop will only be performed as long as the <logical> is TRUE. When the <logical> is FALSE, program execution jumps to the statement following the WEND.

WRITE (SY/MAX)

WRITE <port> (route) <local>, <remote>, <count>

WRITE <port> R[<expr>] <local>, <remote>, <count>

The WRITE statement allows a UCM program to generate SY/MAX network priority WRITE messages from any port on the UCM.

The <port> is an expression which evaluates to a valid UCM port i.e. 1, 2, 3, or 4 for a UCM4 or only 1 for a UCM1. The port number does not have to be the same port that the program is running on. The port which is selected will change to SY/MAX mode, if not already in that mode, and emit the WRITE packet. The port will wait up to 5 seconds for a valid reply from the SY/MAX device before timing out. The reply status will appear in the line number register for the program running port.

There are two types of routing schemes used: static and dynamic.

The **static route** includes the required network drops separated by commas and enclosed in parenthesis. Example: (103, 055). The route will consist of the drop number of the network port that the UCM is connected, any Net-to-Net ports, and finally the network target port.

The **dynamic route** includes a register which contains a pointer to another register which contains the number of drops in the route, N. The next N registers following the number of drops register contain the drop numbers for the route. Example: WRITE 1 R[1045] 5, 6, 8. If R[1045] contains the decimal value 2040, then the WRITE message will look at R[2040] for the number of drops in the route. If R[2040] = 3 then the value of R[2041] will be the first drop number, R[2042] will be the second drop number, and R[2043] will be the target drop.

The **<local>** is an expression which evaluates to a register number within the range of 1...2048. This value is the register within the UCM where the data from the WRITE will be placed. If <count> value is greater that 1 then the <local> is the starting register in the UCM for the WRITE data.

The **<remote>** is an expression which evaluates to a register number within the remote SY/MAX device. This value must be within the valid register range of the external device, usually within the range of 1...8191. Consult the manual for the external device to determine valid register numbers. If the <count> is greater than 1 then the <remote> value is the starting register of the multiple register read.

The **<count>** is an expression which evaluates to the number of consecutive registers to be read with the WRITE statement. The default value is 1. The maximum value is typically 128 but may vary with the external device.

EXAMPLE: WRITE 3 R[143] 2000, 133, 7

Where R[143] = 1123

R[1123] = 4

R[1124] = 5

R[1125] = 118

R[1126] = 102

R[1127] = 8

This WRITE statement will send a SY/MAX network write out UCM port 3 with a dynamic route into network port 005, through the Net-to-Net ports 118, 102, and out network port 008. The WRITE will transfer the data from UCM registers 2000 through 2007 to the external device registers 133 through 140

WRITE PROGRAM

WRITE PROGRAM <port> <local>, <remote>, <count>

The WRITE PROGRAM statement allows a UCM program to copy a group of registers from the user register area to the program area of one of the UCM's ports.

The <port> is an expression which evaluates to a valid UCM port i.e. 1, 2, 3, or 4 for a UCM4 or UCM1.

The <local> is an expression which evaluates a number within the range 1 through 2048.

The <remote> is an expression which evaluates to a number within the range 2050 through 7049.

The <count> is an expression which evaluates to a number within the range 1 through 128.

EXAMPLE: WRITE PROGRAM 2 2000,6101,5

would copy 5 registers from user area R[2000] to Port 2's program area, register 6101.

6
UCM Language Functions

The UCM language includes a variety of commonly used functions to facilitate message generation and reception, and other program flow areas.

Checksum Functions

CRC

Form: CRC(<expr>,<expr>,<expr>)

The CRC function calculates the Cyclical Redundancy Check (CCITT standard) upon a message. The first <expr> is the starting index. This value is number of the character in the message where the CRC16 is to start. The second <expr> is the ending index, usually the $ or $-1 location. The final <expr> is the initial value for the checksum, usually a 0 or -1.

CRC16

Form: CRC16(<expr>,<expr>,<expr>)

The CRC16 function calculates the Cyclical Redundancy Check upon a message. The first <expr> is the starting index. This value is number of the character in the message where the CRC16 is to start. The second <expr> is the ending index, usually the $ or $-1 location. The final <expr> is the initial value for the checksum, usually a 0 or -1.

The CRC16 is a variation of the CCITT standard CRC and is sometimes called a CRC. The MODBUS RTU protocol uses the CRC16.

CRCAB

Form: CRCAB(<expr>,<expr>,<expr>)

The CRCAB function calculates the CRC16 Check upon a message while leaving out the $-2 character. The first <expr> is the starting index. This value is number of the character in the message where the CRC16 is to start. The second <expr> is the ending index, usually the $ location. The final <expr> is the initial value for

the checksum, usually a 0.

The CRCAB is a variation of the CRC16 customized for use with the Allen-Bradley protocols.

LRC

Form: LRC(<expr>,<expr>,<expr>)

The LRC function calculates the Longitudinal Redundancy Check upon a message. The first <expr> is the starting index. This value is number of the character in the message where the LRC is to start. The second <expr> is the ending index, usually the $ or $-1 location. The final <expr> is the initial value for the checksum, usually a 0 or -1.

The LRC operates upon each byte of the message and the result of the function is a byte.

LRCW

Form: LRCW(<expr>,<expr>,<expr>)

The LRCW function calculates the Longitudinal Redundancy Check upon a message. The first <expr> is the starting index. This value is number of the character in the message where the LRCW is to start. The second <expr> is the ending index, usually the $ or $-1 location. The final <expr> is the initial value for the checksum, usually a 0 or -1.

The LRCW operates upon each word of the message and the result of the function is a word.

SUM

Form: SUM(<expr>,<expr>,<expr>)

The SUM function calculates the straight hex sum of a message. The first <expr> is the starting index. This value is number of the character in the message where the SUM is to start. The second <expr> is the ending index, usually the $ or $-1 location. The final <expr> is the initial value for the checksum, usually a 0 or -1.

The SUM function operates upon each byte of the message and returns a byte.

SUMW

Form: SUMW(<expr>,<expr>,<expr>)

The SUMW function calculates the straight hex sum of a message. The first <expr> is the starting index. This value is number of the character in the message where the SUMW is to start. The second <expr> is the ending index, usually the $ or $-1 location. The final <expr> is the initial value for the checksum, usually a 0 or -1.

The SUMW function operates upon each word of the message and returns a word.

Message Description Functions

BCD - Binary Coded Decimal conversion

Usual Format: BCD(Register location, byte count)

or BCD(Register location, VARIABLE)
or BCD(Register location, VARIABLE, Register location)

Valid characters: hexadecimal 00 through 09, 10 through 19 ... 90 through 99.

Transmitting: Converts an expression into its decimal representation, breaks the decimal number into pairs of digits and then translates each pair of digits into its BCD character.

TRANSMIT format: BCD(<expr>,<expr>)

Receiving: Converts BCD characters into pairs of decimal digits, assembles the pairs into a 16 bit decimal number and then compares the number to an expression or places the number into an UCM register.

ON RECEIVE formats: BCD(R[<expr>],<expr>) or BCD((<expr>),<expr>)

Note: The UCM port must be set for 8 bit for BCD to work correctly.

BYTE - Single (lower) byte conversion

Usual Format: BYTE(Register location)

Valid characters: hexadecimal 00 through FF

Transmitting: Converts an expression into its hexadecimal representation and transmits the lower 8 bits as a hexadecimal character.

TRANSMIT format: BYTE(<expr>)

Receiving: Interprets hexadecimal characters as 8-bit hexadecimal numbers and then compares the numbers to an expression or places the numbers into the lower byte of UCM registers and zeros the upper byte of these registers.

ON RECEIVE formats: BYTE(R[<expr>]) or BYTE((<expr>))

Note: If the UCM port is set to 7 bit then bit 8 will always be zero.

DEC - Decimal conversion

Usual Format: DEC(Register location, byte count)
or DEC(Register location, VARIABLE)
or DEC(Register location, VARIABLE, Register location)

Valid characters: ASCII + (plus sign), - (minus sign) and 0 through 9

Transmitting: Converts an expression into its signed decimal representation, breaks the signed decimal number into its sign and its digits and then translates each digit into its ASCII character.

TRANSMIT format: DEC(<expr>,<expr>)

After the significant digits the UCM pads the front of the string with ASCII zeros. Does not transmit the plus (+) sign for positive numbers but does transmit a minus sign (-) on negative numbers.

Receiving: Converts ASCII characters into decimal digits with a sign, assembles the sign and digits into a 16 bit decimal number and then compares the number to an expression or places the number into an UCM register.

ON RECEIVE formats: DEC(R[<expr>],<expr>) or DEC((<expr>),<expr>)

Total number of registers that can be affected: 1

Positive numbers can have a plus (+) sign preceding them but it is not required. Negative numbers must have a minus (-) sign preceding them.

HEX - Hexadecimal conversion

Usual Format: HEX(Register location, byte count)
or HEX(Register location, VARIABLE)
or HEX(Register location, VARIABLE, Register location)

Valid characters: ASCII 0 through 9 and A through F

Transmitting: Converts an expression into its hexadecimal representation, breaks the hexadecimal number into its digits and then translates each hex digit into its ASCII character.

TRANSMIT format: HEX(<expr>,<expr>)

Maximum number of characters that can be sent:

Receiving: Translates ASCII characters into hexadecimal digits, assembles the digits into 16 bit hex numbers and then compares the numbers to an expression or places the numbers into UCM registers.

ON RECEIVE formats: HEX(R[<expr>],<expr>) or HEX((<expr>),<expr>)

Total number of registers that can be affected: 16 (64 characters)

HEXLC - Lower Case Hexadecimal conversion

Usual Format: HEXLC(Register location, byte count)
or HEXLC(Register location, VARIABLE)
or HEXLC(Register location, VARIABLE, Register location)

Valid characters: ASCII 0 through 9 and a through f

Transmitting: Converts an expression into its hexadecimal representation, breaks the hexadecimal number into its digits and then translates each hex digit into its ASCII character. Functions the same as HEX but accepts lower case characters a through f.

TRANSMIT format: HEXLC(<expr>,<expr>)

Maximum number of characters that can be sent: 4

Receiving: Translates ASCII characters into hexadecimal digits, assembles the digits into 16 bit hex numbers and then compares the numbers to an expression or places the numbers into UCM registers. Transmits the hex alpha characters as lower case a through f.

ON RECEIVE formats: HEXLC(R[<expr>],<expr>) or HEXLC((<expr>),<expr>)

Total number of registers that can be affected: 1 (4 characters)

IDEC conversion

Usual Format: IDEC(Register location, byte count)
or IDEC(Register location, VARIABLE)
or IDEC(Register location, VARIABLE, Register location)

Valid characters: ASCII 0 through 9 and : ; < = > ?

Transmitting: Converts an expression into its hexadecimal representation, breaks the hexadecimal number into its digits and then translates each hex digit into its pseudo-ASCII character. In pseudo-ASCII, hex digits 0 through 9 are there normal ASCII characters while hex digits A through F are replaced by the hex characters 3A through 3F which are the ASCII characters : ; < = > and ?.

TRANSMIT format: IDEC(<expr>,<expr>)

Receiving: Converts pseudo-ASCII characters into hexadecimal digits, assembles the digits into 16 bit hexadecimal numbers and then compares the numbers to an expression or places the numbers into UCM registers.

ON RECEIVE formats: IDEC(R[<expr>],<expr>) or IDEC((<expr>),<expr>)

Note: This is the format that the IDEC processors and other devices use to pass register values. If communicating to an IDEC processor, a Square D Model 50 or Micro-1, or any other devices that use this pseudo-ASCII protocol this is a useful function.

OCT - Octal conversion

Usual Format: OCT(Register location, byte count)

Valid characters: ASCII 0 through 7

Transmitting: Converts an expression into its octal representation, breaks the octal number into its digits and then translates each digit into its ASCII character.

TRANSMIT format: OCT(<expr>,<expr>)

Receiving: Converts ASCII characters into octal representation.

ON RECEIVE formats: OCT(R[<expr>],<expr>) or OCT((<expr>),<expr>)

RAW - Raw register conversion

Usual Format: RAW(Register location, byte count)
or RAW(Register location, VARIABLE)
or RAW(Register location, VARIABLE, Register location)

Valid characters: hexadecimal 00 through FF

Transmitting: Converts registers into their hexadecimal representation and translates each 16-bit hexadecimal number into a pair of 8-bit hexadecimal characters.

TRANSMIT format: RAW(R[<expr>],<expr>)

Receiving: Interprets hexadecimal characters as 8-bit hexadecimal numbers, assembles each pair of 8-bit numbers into a 16-bit hexadecimal number, high byte then low byte, and then compares the numbers to an expression or places the numbers into UCM registers.

ON RECEIVE formats: RAW(R[<expr>],<expr>) or RAW((<expr>),<expr>)

Note: If the UCM port is set to 7 bit then bit 8 and bit 16 will always be 0. RAW is an expanded version of SY/MAX packed ASCII and can be used to transmit and receive packed ASCII characters as well as 8-bit characters.

RWORD

Usual Format: RWORD(Register location)

Valid characters: hexadecimal 00 through FF

Transmitting: Converts an expression into its 16-bit hexadecimal representation, translates the 16-bit number into a pair of 8-bit hexadecimal numbers and transmits the lower eight bits and then the upper 8 bits as hexadecimal characters.

TRANSMIT format: RWORD(<expr>)

Receiving: Interprets two hexadecimal characters as two 8-bit hexadecimal numbers, assembles the two 8-bit numbers into a 16-bit number, first number low byte and second number high byte, and then compares the number to an expression or places the number into an UCM register.

ON RECEIVE formats: RWORD(R[<expr>]) or RWORD((<expr>))

Note: Like WORD but in the reverse order, low byte then high byte.

TON - Translate on

The commands TON and TOFF work with the TRANSLATE command. The TRANSLATE command defines a string that is to be translated into another string. This is used when a character has reserved meaning but could also be used in the translation of data. Up to 8 TRANSLATE strings can be contained in an UCM program.

An example: the escape character (hex 1B) could be used to interrupt a transmission but hex 1B might also be valid data. When the remote process wants to interrupt transmission it sends a single hex 1B. But when the remote process wants to send data containing hex 1B it sends 1B1B and the UCM is responsible for interpreting two hex 1Bs as a single 1B instead of as an escape. In this case the translate command would be:

```
TRANSLATE 1:"\1B\1B" = "\1B"
```

and the command for receiving data that might contain a hex 1B:

```
ON RECEIVE TON(1):RAW(R[12],15):TOFF(1)
```

The TON command turns on translation during an ON RECEIVE or TRANSMIT. The format for turning translation on is TON(<expr>) where <expr> is the translation number and must evaluate to be between 1 and 8. The TON is usually followed by a TOFF.

TOFF - Translate off

The TOFF command turns off translation during an ON RECEIVE or TRANSMIT. The format for turning translation off is TOFF(<expr>) where <expr> is the translation number and must evaluate to be between 1 and 8.

UNS - Unsigned decimal conversion

Usual Format: UNS(Register location, byte count)
or UNS(Register location, VARIABLE)
or UNS(Register location, VARIABLE, Register location)

Valid characters: ASCII 0 through 9

Transmitting: Converts an expression into its unsigned decimal representation, breaks the unsigned decimal number into its digits and then translates each digit into its ASCII character.

TRANSMIT format: UNS(<expr>,<expr>)

Receiving: Converts ASCII characters into decimal digits, assembles the digits into a 16 bit unsigned decimal number and then compares the number to an expression or places the number into an UCM register.

ON RECEIVE formats: UNS(R[<expr>],<expr>) or UNS((<expr>),<expr>)

Total number of registers that can be affected: 1

WORD

Usual Format: WORD(Register location)

Valid characters: hexadecimal 00 through FF

Transmitting: Converts an expression into its 16-bit hexadecimal representation, translates the 16-bit number into a pair of 8-bit hexadecimal numbers and transmits the upper eight bits and then the lower 8 bits as hexadecimal characters.

TRANSMIT format: WORD(<expr>)

Receiving: Interprets two hexadecimal characters as two 8-bit hexadecimal numbers, assembles the two 8-bit numbers into a 16-bit number, first number the high byte and second number the low byte, and then compares the number to an expression or places the number into an UCM register.

ON RECEIVE formats: WORD(R[<expr>]) or WORD((<expr>))

Note: Like RWORD but always high byte then low byte. Also like RAW(R[<expr>],2).

Other Functions

CHANGED

Format: CHANGED(R[<expr>]) or CHANGED(R[<expr>] & <expr>)

The CHANGED function provides a boolean result dependent upon whether the evaluated register or mask of the register has been altered from the last operation of this function. The first occurrence of the CHANGED function will result in a FALSE regardless of the state of the evaluated register.

The CHANGED function is used in any place referred to as <logical>, such as:

```
IF CHANGED(R[56]) THEN GOTO reply
```

The CHANGED function is similar to the ON CHANGE statement, but the CHANGED function allows program execution to continue running instead of pausing to wait for the change to occur.

EXPIRED

Format: EXPIRED(<expr>)

The EXPIRED function tests a timer register and returns a logical true if the previously set timer has expired. (A timer is set using the SET TIMER command). For

Example:

IF EXPIRED(R[201]) THEN GOTO TIMEOUT

would test the status of the timer regiser in R[201], and jump if that timer has expired.

FLOAT

Format: FLOAT(R[<expr>])

The FLOAT function converts an integer register value to a floating point number.

The result of the FLOAT function is used as a floating point number; either to assign to a floating point variable, or in a floating point calculation:
F[101] = FLOAT(R[10]) / 10.0

Remember that Floating point numbers occupy two register locations, as they do in the SY/MAX processor. For example, F[101] would use both registers R[101] and R[102].

MAX

Format: MAX(<expr>,<expr>)

The MAX function provides a result of the <expr> which evaluates to the larger of the two expressions.

MIN

Format: MIN(<expr>,<expr>)

The MIN function provides a result of the <expr> which evaluates to the smaller of the two expressions.

SWAP

Format: SWAP(<expr>)

The SWAP function reverses the byte order of the result of the <expr>. If R[44] = xABCD then SWAP(R[44]) would bring the result xCDAB.

PORT

PORT returns the UCM port number that the program is running. Values are 1 through 4 on an UCM4 and only 1 on an UCM1.

RTS

RTS is a variable which may be used to control the state of the Request to Send line for a UCM port. RTS = TRUE will assert the RTS line. RTS = FALSE will negate the RTS line. Any action which takes control over the port hardware such as a SY/MAX operation, will override the setting of RTS.

TRUNC

Format: TRUNC(F[<expr>])

The TRUNC function converts a floating point register value to an integer number, by truncating the fractional portion of the number.

The result of the TRUNC function is used as an integer value; either to assign to a register variable, or in an integer calculation:
R[10] = TRUNC(F[101]) * 10

Remember that Floating point numbers occupy two register locations, as they do in the SY/MAX processor. For example, F[101] would use both registers R[101] and R[102].

CTS

CTS is a variable which gives the current state of Clear to Send on the UCM port. IF CTS = TRUE then CTS is asserted by the external device. If CTS = FALSE then CTS is negated.

7
Configuration Software UCMSW

UCMSW

The UCMSW software program is provided free of charge to UCM users. This software is used to program the operation of the UCM.

The startup screen of UCMSW is shown in Figure 7-1 on page 64. The operational modes are selected by the highlighted menu bar on the fourth line. Selection can be made by moving the cursor to the desired option using the arrow keys and pressing **ENTER**. A short cut is provided, simply type "**D**" for Development, "**U**" for Utility, "**S**" for setup or "**Q**" to quit.

UCMSW also contains two convenient utilities for general use, a SY/MAX Register Viewer and a Terminal Emulator. The Register Viewer uses the same setup as the Sy/Max functions. The Terminal Emulator has its own setup characteristics.

Data Entry Keys

Whenever data entry is allowed by the program, certain keys can be used to facilitate data entry. They are:

BACKSPACE	Move cursor left and remove character there
LEFT ARROW	Move cursor to the left one character
RIGHT ARROW	Move cursor to the right one character
DEL	Remove the character under the cursor
INS	Change between insert and overstrike modes of entry
HOME	Move cursor to the left edge of the field
END	Move cursor to the end of the data
Control-F	Move cursor right (Forward) one word
Control-R	Move cursor left (Reverse) one word

Control-D	Delete from the cursor to the end of the field
Control-U	Delete from cursor to the beginning of the field
Control-Y	Delete all characters in the field
ESC	Exit the field without modifying it
ENTER	Accept the contents of the field

When a field is opened for input, the cursor is positioned at the left side of the field. If data is already present in the field, typing any character other than those listed above will cause the field to be blanked allowing entry of new data without first deleting the old. If it is desired to retain the previous data for editing, make sure the first key you type is an editing key such as a left or right arrow.

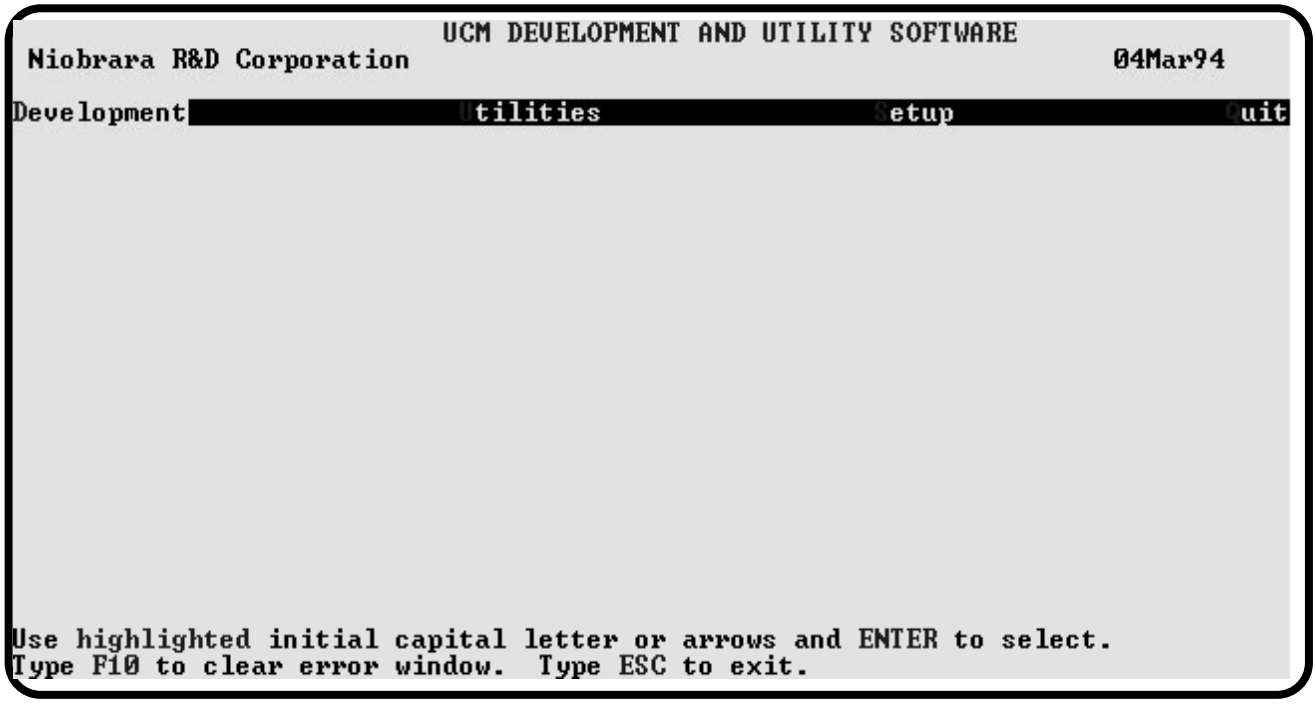


Figure 7-1 Startup Screen

Development Functions

By selecting the Development menu item it is possible to choose from a variety of options to assist the development of UCM applications. The UCMSW development system operates in the following manner:

The user selects a file name for the application, which will now be referred to as <filename>.UCM. If <filename>.UCM exists, the user will load it to memory, otherwise the user will be prompted to create the file. Upon selection, and loading of the <filename>.UCM, the UCMSW performs all editing not on <filename>.UCM but UCM$WORK.UCM. This allows the original <filename>.UCM to remain unchanged during editing and compiling. All changes will be made on UCM$WORK.UCM. This original <filename>.UCM will not be altered unless the "Write source to disk" is

selected. This method permits changes to be made while keeping the original source code intact.

Upon selection of Development, a new menu as shown in Figure 7-2 is displayed. Each menu item is described below.

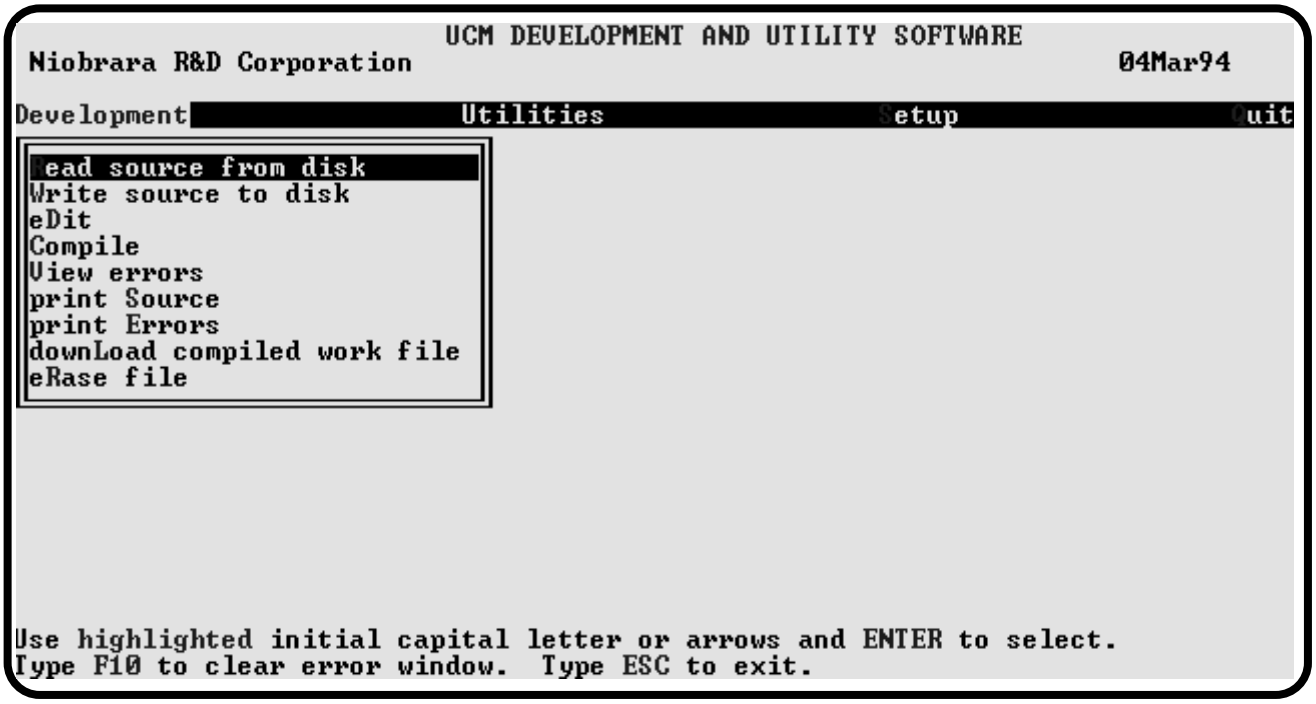


Figure 7-2 Development Menu

"Read source from disk"

This function reads an UCM program from disk into the work file UCM$WORK.UCM. The file to be read should have been created by the "Write source to disk" function described below and must have a .UCM extension. When "Read source from disk" is selected, a window will open and ask for the name of the file to read. The bottom part of the screen will show a list of all files with the extension .UCM in the current directory. Any subdirectories, or drives, will be shown in square brackets. The parent directory (of which the current directory is a subdirectory) is shown by the word "parent" in square brackets. You may either type the name of the file to read or you may use the arrow keys to move the highlight to the desired filename. Pressing **ENTER** with the highlight on a filename will select that file for reading. Pressing **ENTER** with the highlight positioned on a directory name (either a

subdirectory or [parent]) will change the current directory to that directory and will show the .UCM files in the new directory. If there are more files than will fit on the screen, pressing the right arrow with the highlight at the right edge of the screen will scroll the display sideways to show more files. Typing the **ESC** key will return to the offline function menu without loading a file.

"Write source to disk"

This function saves a copy of the UCM$WORK.UCM text file to a disk file. "Write source to disk" uses the same point and shoot file selection described for "Read source from disk" above. To create a new file you must type the name. The name should be a valid MS/DOS filename but should not include any path name or extension. The program will append an extension of .UCM to the name and the file will be placed in the directory which is shown in the bottom half of the screen. To create a file in a directory other than the current one, use the arrow and **ENTER** keys to traverse the directory tree until a listing of the desired directory is shown in the bottom half of the screen. Then type in the file name and press **ENTER**. If you specify (either by typing or by pointing) a file that already exists, you will be prompted for approval before that file is overwritten.

"eDit"

The eDit selection will open a new DOS shell with the text editor selected in the SY/MAX Setup menu. This editor will open loaded with the file UCM$WORK.UCM. Use the editor in the normal fashion to make changes to the text file. Upon completion of the editing, save the work. After saving the file, exit the editor. Upon exiting the editor, the main UCMSW menu will appear.

It is important to remember to save the file before exiting or the changes will not be made.

"Compile"

Use this function when you wish to compile your program. The compiler will compile the UCM$WORK.UCM file and generate the "source filename".UCC if the compile is successful. When the compile is finished and successful the user is prompted to download the compiled code into the UCM. If "Y" is selected the Download window will open and the user is prompted for the port number, status register location, and Auto-start values.

If any errors occur in the compile, the "source filename.".UCC will not be generated and a file "UCM$WORK.ERR" will be made.

"View errors"

This function allows the viewing of UCM$WORK.ERR to see where the errors occurred during the compile. This is accomplished by opening the same editor used in eDit with the file UCM$WORK.ERR. When finished viewing the errors simply exit the editor to return to UCMSW.

"print Source"

This function will produce a report showing the source code in UCM$WORK.UCM. When this function is selected, you will be prompted for an output filename with the default value of PRN shown. To send the report to the PRN device (normally the par-

allel printer port), simply press **ENTER**. To send the report to a different port or to a file, type the name and then press **ENTER**. Online configurations may be printed with the **F1** print screen key.

"print Errors"

This function will produce a report showing the error list in UCM$WORK.ERR. When this function is selected, you will be prompted for an output filename with the default value of PRN shown. To send the report to the PRN device (normally the parallel printer port), simply press **ENTER**. To send the report to a different port or to a file, type the name and then press **ENTER**. Online configurations may be printed with the **F1** print screen key.

"downLoad compiled work file"

This routine allows the downloading of the compiled result of the file selected with "Read source from disk". The source code must be compiled before this function is used. This routine is called also at the end of a successful compile automatically.

The download window prompts the user for the UCM port to load the program into. Valid ports are 1, 2, 3, or 4 on an UCM4 or only 1 on an UCM1.

The location of the status register pair is also prompted for. The default values are 2 for port 1, 4 for port 2, 6 for port 3 and 8 for port 4. This value must fall within the range of 2 through 2047. Also each port must have its own unique value for the status pair.

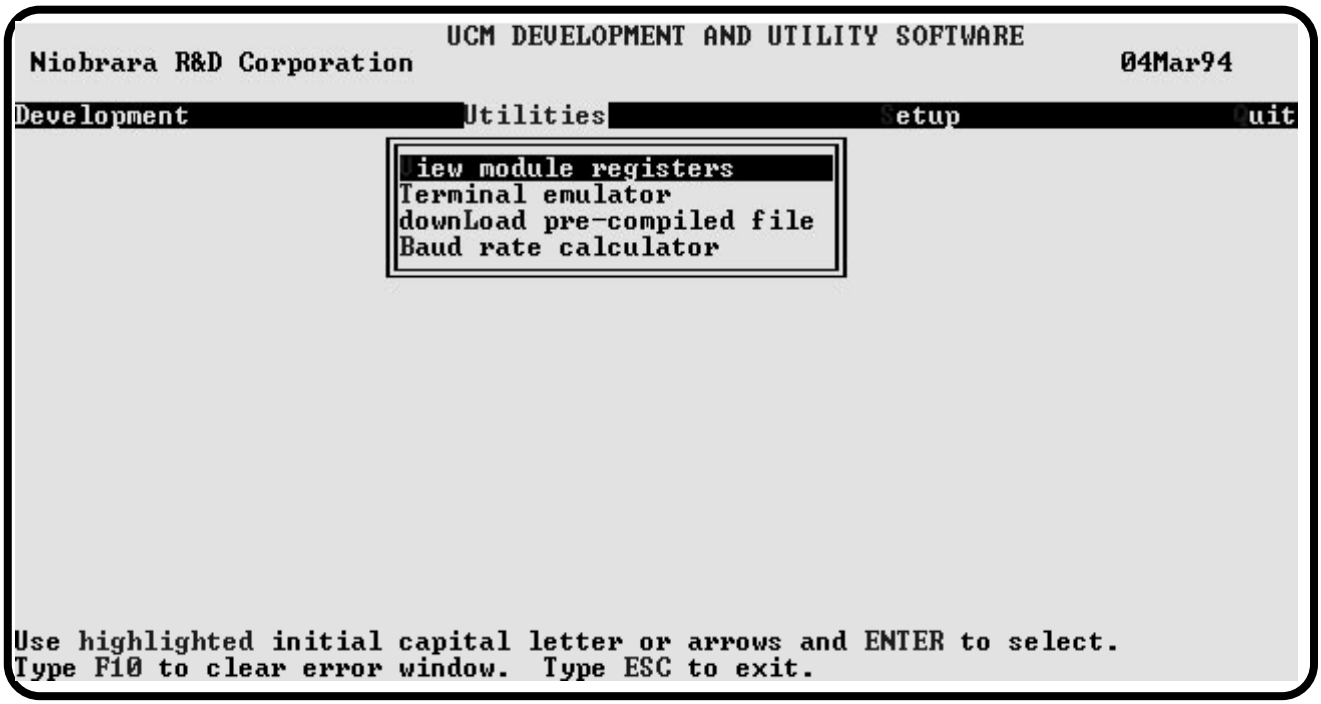
The Auto-start feature allows a port to automatically start running upon power-up. Selecting the Auto-start feature on any of the ports forces the command register (register 1) of the UCM to be a PLC input register with a status E000 hex. This automatically prevents the PLC from controlling the running of any of the programs by setting the command bits directly. The Auto-start feature should only be used for stand alone applications.

"eRase file"

This function removes the selected UCM source file from the disk. "eRase file" uses the same point and shoot file selection described for "Read source from disk" above. The name should be a valid MS/DOS filename but should not include any path name or extension. The program remove the file from the disk as well as removing it from the bottom of the screen. To remove a file in a directory other than the current one, use the arrow and **ENTER** keys to traverse the directory tree until a listing of the desired directory is shown in the bottom half of the screen. Then type in the file name and press **ENTER**.

Utilities

The Utilities menu provides access to useful maintenance and testing functions of the UCMSW software.



View module registers

Selecting the View module registers menu item will invoke a SY/MAX register data viewer/modifier. This viewer continuously performs a block read of 20 registers and displays the contents of those registers in hex, unsigned integer, signed integer, and binary. The status register associated with the data register is also displayed in hex. The register viewer is dependent on the values located in the SETUP Sy/Max menu. Mode, Baud rate, Parity, Route, etc. must be properly set for proper communication.

The Up and Down arrow keys are used to move from register to register.

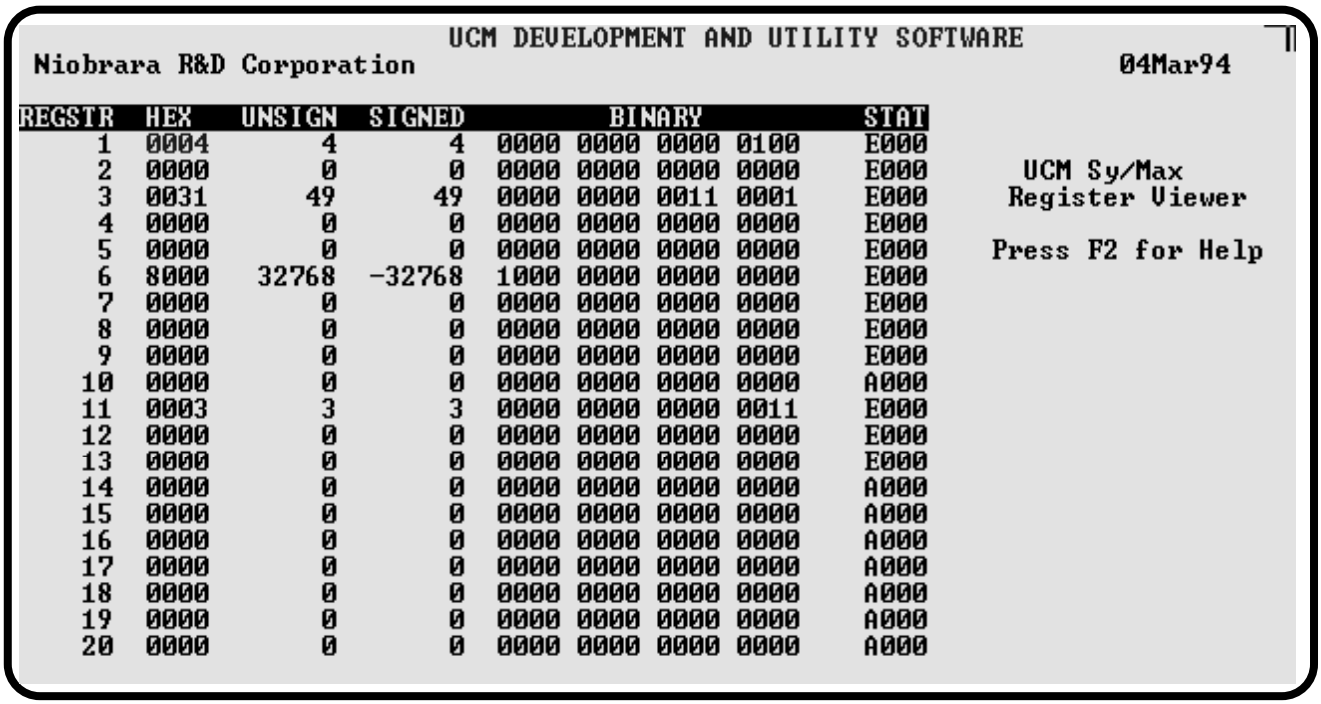
The Page Up and Page Down keys move in increments of 10 registers.

The Left and Right arrows move from column to column on the same register.

This register viewer is highly useful in that it allows easy editing of the data in the register being viewed. By pressing 0..9 in the decimal fields or 0..9, or A..F in the hex field, an editing mode is entered. New data may be entered at this time. Pressing the Enter key or moving to a new field with the arrow keys will cause the new data to be written to the edited register. If the curser is located in the REGISTER column the block of registers being viewed may be adjusted by entering a new register number. To edit the binary values, press HOME when on the binary field. Move the cursor to the desired bit and enter a ’0’ or a ’1’ and press enter to accept.

Pressing Esc will exit from the Register viewer and return to the main menu. Pressing Esc while editing a data field will result in canceling the edit and the modified data will not be written to the register.

The STAT field displays the status register associated with the data register. The STATUS field is a read only display and can not be modified by the Register Viewer. Two common values are E000 and A000. A000 is the hex representation that the PLC recognizes as a PLC OUTPUT register. E000 is for a PLC INPUT register. This allows easy recognition of registers used by the UCM as inputs and used by the PLC as outputs.



REGSTR	HEX	UNSIGN	SIGNED	BINARY	STAT
1	0004	4	4	0000 0000 0000 0100	E000
2	0000	0	0	0000 0000 0000 0000	E000
3	0031	49	49	0000 0000 0011 0001	E000
4	0000	0	0	0000 0000 0000 0000	E000
5	0000	0	0	0000 0000 0000 0000	E000
6	8000	32768	-32768	1000 0000 0000 0000	E000
7	0000	0	0	0000 0000 0000 0000	E000
8	0000	0	0	0000 0000 0000 0000	E000
9	0000	0	0	0000 0000 0000 0000	E000
10	0000	0	0	0000 0000 0000 0000	A000
11	0003	3	3	0000 0000 0000 0011	E000
12	0000	0	0	0000 0000 0000 0000	E000
13	0000	0	0	0000 0000 0000 0000	E000
14	0000	0	0	0000 0000 0000 0000	A000
15	0000	0	0	0000 0000 0000 0000	A000
16	0000	0	0	0000 0000 0000 0000	A000
17	0000	0	0	0000 0000 0000 0000	A000
18	0000	0	0	0000 0000 0000 0000	A000
19	0000	0	0	0000 0000 0000 0000	A000
20	0000	0	0	0000 0000 0000 0000	A000

Figure 7-3 View Registers

Terminal Emulator

Selecting the Terminal emulator from the Utilities menu will invoke a terminal emulator according to the setup selected in the Setup menu. The terminal emulator opens as shown in Figure .

The terminal sends the ASCII code for the alpha-numeric characters out the selected COM port. Functions keys F1 through F4 and the keypad arrows send ANSI (i.e. VT100) codes. F7 is reserved for starting a file capture. F8 will close the capture file. The backspace key sends ASCII BS (08 hex). The Delete key sends and ASCII DEL (7F hex). The Insert key allows the transmission of ASCII hexadecimal characters directly from the hex numbers separated by spaces.

The terminal displays printable ASCII characters which are received on the port. Non-printable characters are displayed as the hexadecimal value enclosed in <>, such as <0D><0A> indicates the carriage return, line feed characters. The terminal is always in this "monitor" mode and therefore ANSI output emulation is not provided.

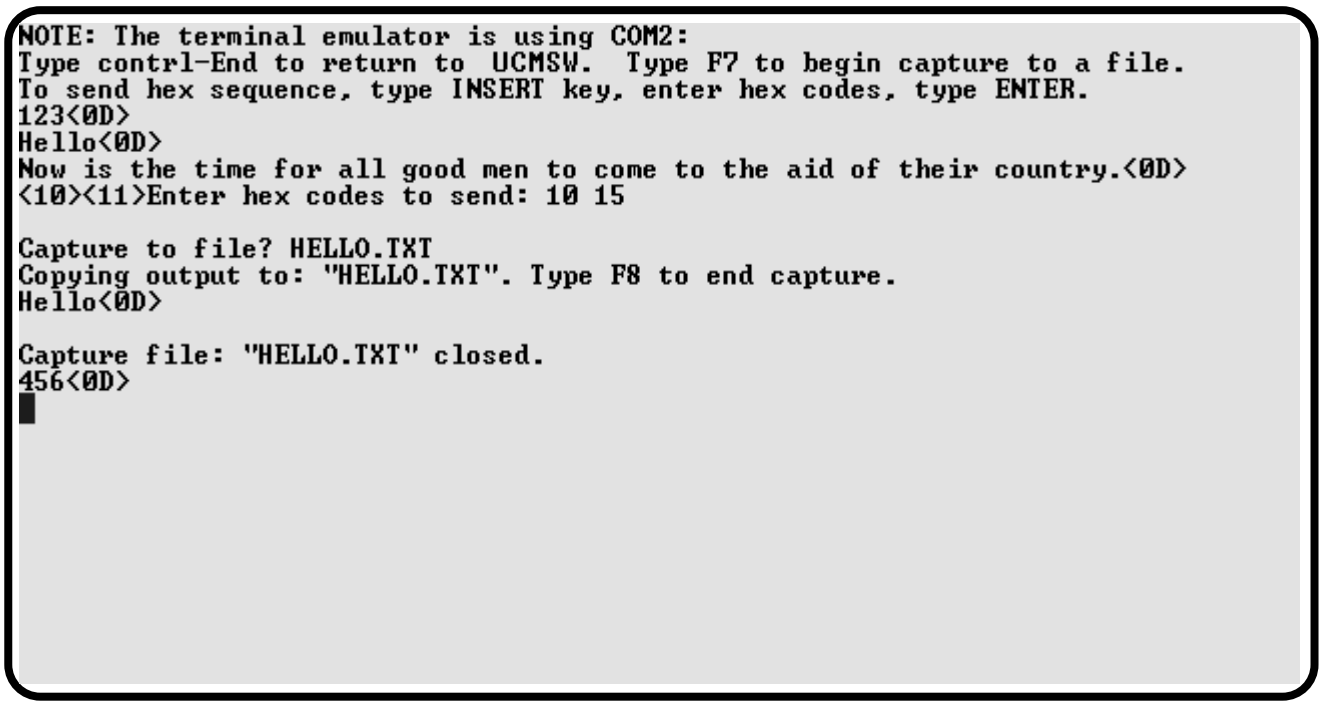


Figure 7-4 Terminal Emulator

Download Pre-compiled file

The downLoad pre-compiled file selection will open the download file window. (Figure 7-5) This allows the downloading of a UCM program which has already been compiled (filename.UCC). The files which have the extension .UCC will be displayed. Use the cursor to select the file or type in the name.

The Download window will then appear and allow the selection of UCM port parameters including status registers and Auto-start.

Baud Rate Calculator

The UCM module calculates the actual baud rate that is generated by the SET BAUD command. This value will depend upon the actual hardware of the module. The UCM1 will calculate a different baud rate for the same setting as the UCM4. The Baud Rate Calculator routine is given to provide an actual baud rate value which will be generated with the SET BAUD statement. Select the module type and enter the baud value to be used in the SET BAUD statement and the actual baud rate that will be generated by the UCM hardware will be displayed as in Figure 7-6.

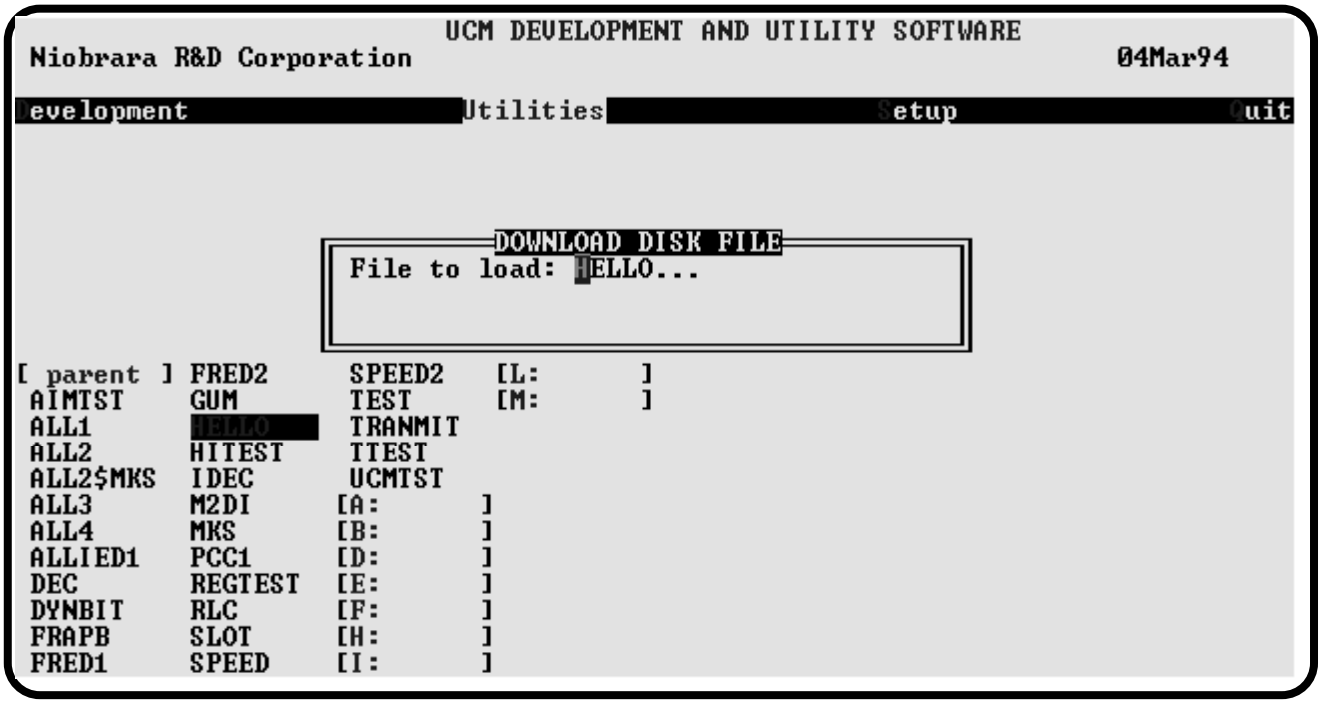


Figure 7-5 Download pre-compiled file

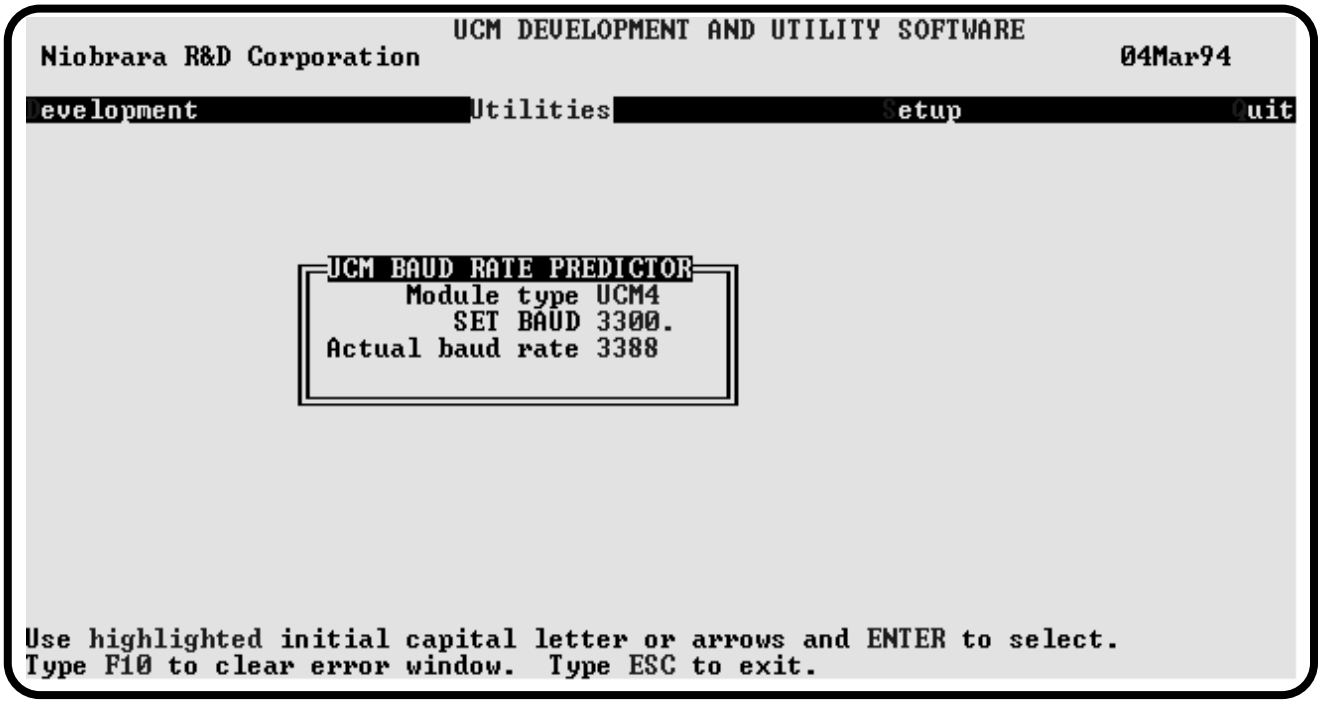


Figure 7-6 Baud Rate Calculator

SETUP

The setup menu accesses the setup parameters for the personal computer to enable it to communicate with the UCM and the terminal emulator. The parameters chosen will depend on the exact equipment involved in making the connections.

SY/MAX SETUP

The connection type is mainly determined by the method of connection to outside world and may be broken into two groups: the personal computer's COM: port and the SY/LINK Network Interface Card.

Personal Computer COM: port

If the connection from the personal computer is made through one of its serial ports then the Connection type should be one of the following:

- Sy/Max COM:
- Net-to-Net COM:
- Gateway COM:

Sy/Max COM: is the default and most likely will be the one used. In this mode the personal computer will communicate through one of its COM: ports as though it were a SY/MAX device such as a PLC. The full SY/MAX protocol is supported including routing so SY/MAX COM: may be used through SY/MAX mode ports on NIMs and SPE4s with appropriate routing. This mode is to be used when a direct connection from the personal computer COM: port is made to the UCM. In most cases an RS-232<>RS-422 conversion is required and the Niobrara SC406 (or SC902) cable makes this conversion very convenient.

Net-to-Net COM: is used when connecting to a NIM or SPE4 that is set to Net-to-Net mode. The first drop number in the route will be that of the address of the NIM.

Gateway COM: is used when connecting to an SPE4 port that is in Gateway mode. For more information about Gateway mode see the SPE4 instruction manual.

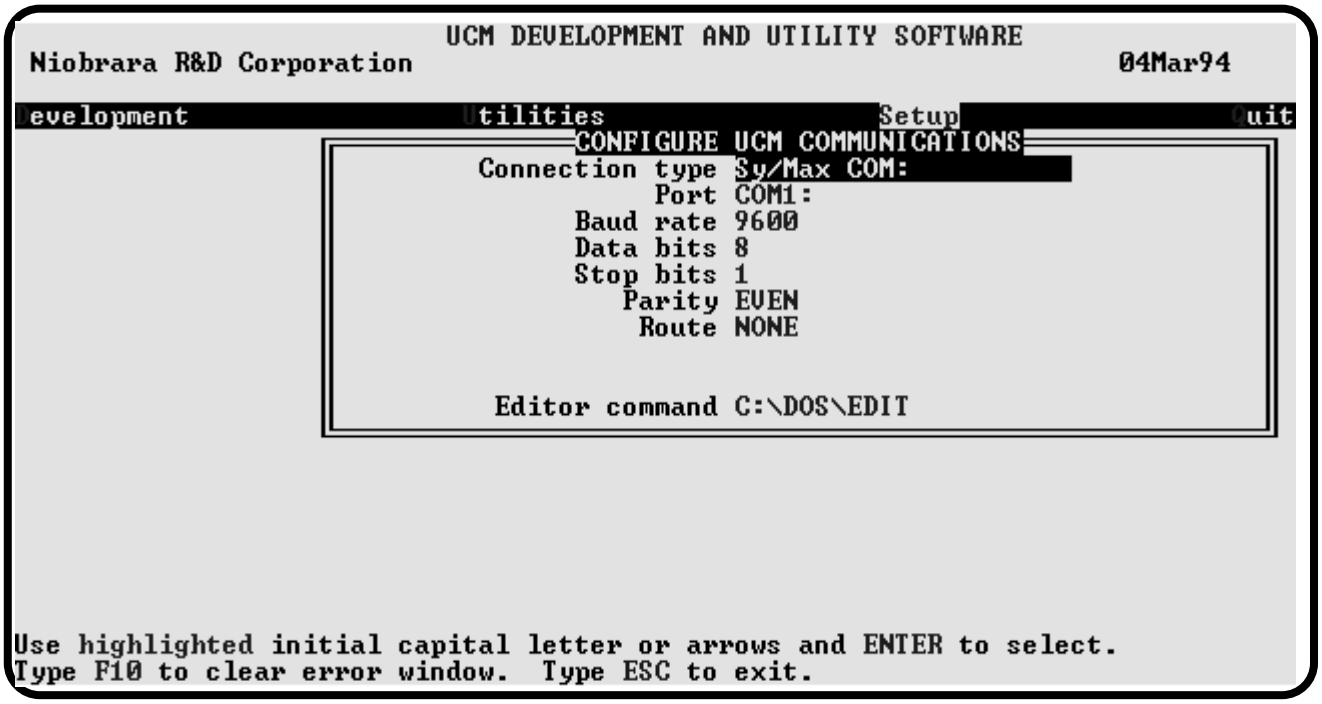


Figure 7-7 SY/MAX Setup Screen

Port - When one of the COM: connection types is selected a particular port of the personal computer must be selected. Available choices are: COM1:, COM2:, COM3:, and COM4:. Select the port which will be used to connect to the UCM.

Baud rate - The Baud rate selected here is the baud rate of the personal computer serial port selected. This value should be set to match the device connected to the personal computer. An UCM has a default baud rate of 9600 and if a direct connection is made to the UCM this is the setting that should be made on the personal computer. If the baud rate of the UCM has been changed this value may need to be adjusted.

Data bits - When in SY/MAX or Net-to-Net modes the data bits is required to be 8 and may not be changed. The SY/MAX protocol requires 8 data bits. The Gateway mode uses ASCII messages which do not require the full 8 data bits and may be set to 8 or 7 depending on the attached device.

Stop bits - The stop bits are normally set to 1 but may be adjusted to 2 for some particular application. The UCM is set for 1 stop bit.

Parity - SY/MAX and Net-to-Net modes normally use EVEN parity and that is the default for the UCM. Other choices are ODD and NONE.

Route - The route is used to determine the path from the personal computer to the UCM. If a direct connection is made from the personal computer to the UCM, i.e. without going through a SY/NET network or an SPE4, this value is set to NONE by pressing the Delete key. If a SY/MAX connection is made to a SY/MAX mode port on an NIM or SPE4 the first drop will be that of the drop number of the NIM or SPE4 port. If any Net-to-Net drops are included between the port connected to the personal

computer and the port connected to the UCM, they must be included in order of occurrence from the personal computer to the UCM. The last drop number listed will be that of the NIM or SPE4 SY/MAX mode port that is connected to the UCM. Up to 8 total drops are supported by the SY/MAX protocol.

If the personal computer is in Net-to-Net mode the first drop will be that of the Net-to-Net port of the NIM or SPE4 that the personal computer is connected to. Subsequent drops will be included like above.

The Gateway mode route will include the Gateway port on the SPE4 that the personal computer is connected and any subsequent Net-to-Net and SY/MAX drops to reach the UCM.

SY/LINK Connection

UCMSW provides full support of the Square D SY/LINK network interface card. Setup for the network interface is provided along with setup for the RS-422 port on the card.

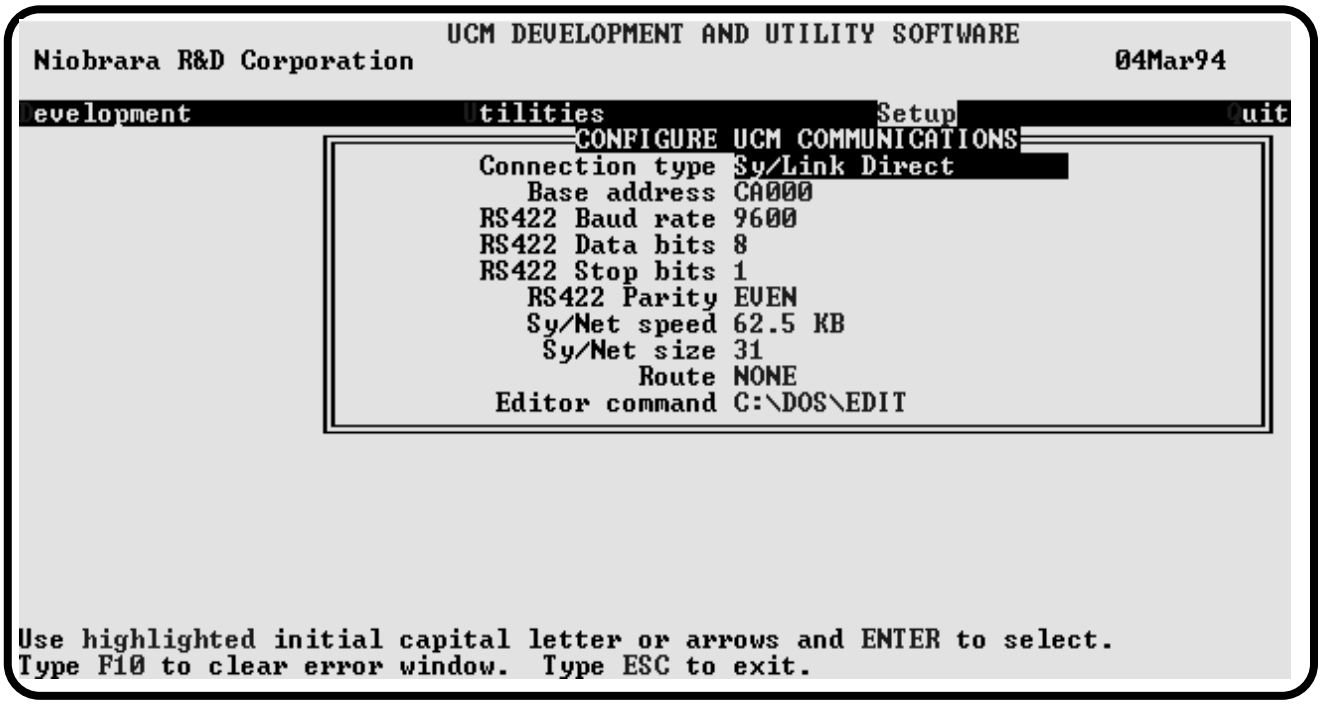


Figure 7-8 SY/LINK Setup Screen

Connection type - The RS-422 port may be set to SY/MAX or Net-to-Net modes. For a direct connection to the UCM from the RS-422 port of the SY/LINK card choose the Sy/Link Direct mode. If an indirect connection from the RS-422 port of the card is made through other Net-to-Net ports choose Sy/Link Net-to-Net. If the RS-422 port is not used and the connection is make through the SY/NET network to another NIM, the choice does not matter.

Base address - This is a hex value that represents the SY/LINK's cards address range selected by DIP switches on the card. Select the same range that is set on the card.

RS422 Baud rate - Select the baud rate to match the external device, normally 9600.

RS422 Data bits - Select the data bits to match the external device, normally 8.

RS422 Stop bits - Select the stop bits to match the external device, normally 1.

RS422 Parity - Select the parity to match the external device, normally EVEN.

Sy/Net speed - Select to match the speed settings of the other devices on the SY/NET.

Sy/Net size - Select to match the setting on the other SY/NET devices.

Route - The first drop in the route defines the network address of the SY/LINK board. Since the personal computer is connected to the SY/LINK card through the edge connector of the card, port 0, the drop number must start with 0. The remaining two digits of that drop should be selected not to match any other device on the SY/NET. For instance, there are three NIMs on the network addressed 01, 02, and 03. It seems logical to make the SY/LINK card be at address 04 so the first drop in the route field will be 004. The next drop will be that of the NIM port connected to the UCM, or another Net-to-Net port. If Sy/Link Direct was selected and the UCM is connected directly to the RS-422 port of the SY/LINK card the full route statement would be 004 104 as the RS-422 port is considered to be port 1.

Terminal Emulator SETUP

The Terminal Emulator setup allows an individual setup for the operation of the terminal emulator. For instance, this separate setup will allow COM1 to be used for the UCM SY/MAX connection and COM2 to be used for a terminal emulator connection.

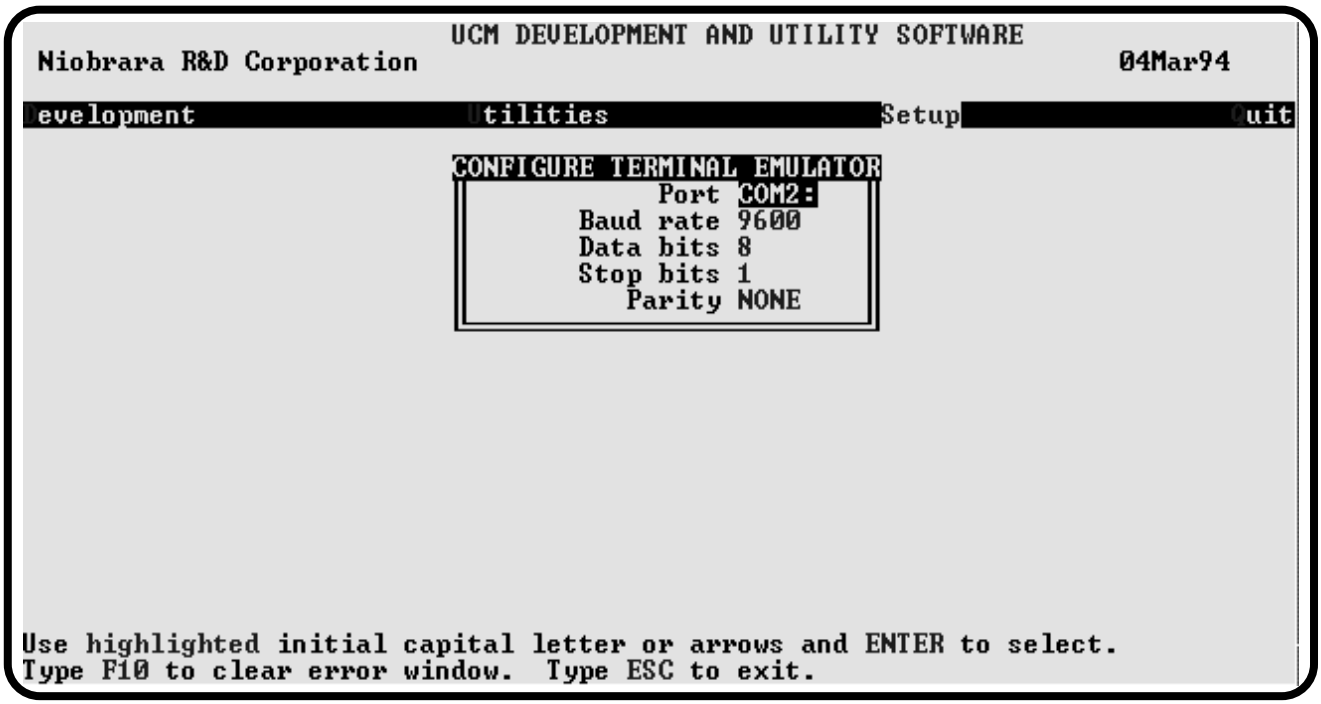


Figure 7-9 Terminal Emulator Setup Screen

Port - Available choices are: COM1:, COM2:, COM3:, and COM4:. Select the port which will be used to connect to the external device.

Baud rate - The Baud rate selected here is the baud rate of the personal computer serial port selected. This value should be set to match the device connected to the personal computer.

Data bits - Selects the number of data bits for the message packet. Choices are 8 or 7 depending on the attached device.

Stop bits - The stop bits are normally set to 1 but may be adjusted to 2 for some particular application.

Parity - Choices are EVEN, ODD, and NONE.

Command Line Parameters

UCMSW may be started from the DOS command line with a sequence of letters which represent the keystrokes necessary to perform any operation. This allows batch processing of various commands such as downloading of stored setup files. The special characters /R, /D, and /E refer to the Return key, Delete key, and Escape key respectively. The parameters are not case sensitive. The following example changes the Route to 102,055; loads the compiled file TEST.UCC into an UCM for operation on Port 1 with the Status Register at 2, then quits. It is assumed that the UCM is connected to the computer and the rest of UCMSW setup is correct.

>UCMSW S/S/R/R/R/R/R/D102,055/R/RYUlTEST/R1/R2/R/RQ

The S selects Setup, S for Sy/Max, five Return keys to get to the Route field, /D for Delete to remove any previous route, 102,055 Return, for the new route, Y for Yes save the setup to disk, U for Utility, L for downLoad pre-compiled file, TEST Return for the filename, 1 Return for the port number, 2 Return for the Status Registers, Return for NO on Autostart, and finally Q for Quit.

Another way of changing the Setup is to copy the setup file to another name in DOS, then copy it back to UCM.STP in the batch file before calling UCMSW.

8

Examples

TRANSMIT message function with register references

In the following TRANSMIT examples the following initial conditions are assumed:

UCM Register	Decimal	Signed Decimal	Hex	Octal	Binary
R[23]	41394	24142	A1B2	120662	1010 0001 1011 0010
R[24]	20318	20318	4F5E	47536	0100 1111 0101 1110

TRANSMIT HEX

Command: TRANSMIT HEX(R[23],4)
ASCII Characters transmitted: A1B2
Decimal values: 65 49 66 50
Hex values: 41 31 42 32

Command: TRANSMIT HEX(R[23],2)
ASCII Characters transmitted: B2
Decimal values: 66 50
Hex values: 42 32

Command: TRANSMIT HEX(R[23],8)
ASCII Characters transmitted: 0000A1B2
Decimal values: 48 48 48 48 65 49 66 50
Hex values: 30 30 30 30 41 31 42 32

Command: TRANSMIT HEX(R[23],VARIABLE)
ASCII Characters transmitted: A1B2
Decimal values: 65 49 66 50
Hex values: 41 31 42 32

Command: TRANSMIT HEX(R[23],VARIABLE R[600])

ASCII Characters transmitted: A1B2
Decimal values: 65 49 66 50
Hex values: 41 31 42 32
R[600] would then equal 4.

TRANSMIT DEC

Command: TRANSMIT DEC(R[23],6)
ASCII Characters transmitted: -24142
Decimal values: 45 50 52 49 52 50
Hex values: 2D 32 34 31 34 32

Command: TRANSMIT DEC(R[23],5)
ASCII Characters transmitted: 24142
Decimal values: 50 52 49 52 50
Hex values: 32 34 31 34 32

Command: TRANSMIT DEC(R[23],12)
ASCII Characters transmitted: -00000024142
Decimal values: 45 48 48 48 48 48 48 50 52 49 52 50
Hex values: 2D 30 30 30 30 30 30 32 34 31 34 32

Command: TRANSMIT DEC(R[23],VARIABLE)
ASCII Characters transmitted: -24142
Decimal values: 45 50 52 49 52 50
Hex values: 2D 32 34 31 34 32

Command: TRANSMIT HEX(R[23],VARIABLE R[600])
ASCII Characters transmitted: -24142
Decimal values: 45 50 52 49 52 50
Hex values: 2D 32 34 31 34 32
R[600] would then equal 6.

TRANSMIT UNS

Command: TRANSMIT UNS(R[23],5)
ASCII Characters transmitted: 41394
Decimal values: 52 49 51 57 52
Hex values: 34 31 33 39 34

Command: TRANSMIT UNS(R[23],3)
ASCII Characters transmitted: 394
Decimal values: 51 57 52
Hex values: 33 39 34

Command: TRANSMIT UNS(R[23],8)
ASCII Characters transmitted: 00041394
Decimal values: 48 48 48 52 49 51 57 52
Hex values: 30 30 30 34 31 33 39 34

Command: TRANSMIT UNS(R[23],8)
ASCII Characters transmitted: 00041394
Decimal values: 48 48 48 52 49 51 57 52
Hex values: 30 30 30 34 31 33 39 34

TRANSMIT OCT

Command: TRANSMIT OCT(R[23],6)
ASCII Characters transmitted: 120662
Decimal values: 49 50 48 54 54 50
Hex values: 31 32 30 36 36 32

Command: TRANSMIT OCT(R[23],3)
ASCII Characters transmitted: 662
Decimal values: 54 54 50
Hex values: 36 36 32

Command: TRANSMIT OCT(R[23], VARIABLE)
ASCII Characters transmitted: 120662
Decimal values: 49 50 48 54 54 50
Hex values: 31 32 30 36 36 32

Command: TRANSMIT OCT(R[23], VARIABLE R[600])
ASCII Characters transmitted: 120662
Decimal values: 49 50 48 54 54 50
Hex values: 31 32 30 36 36 32
R[600] would then equal 6.

TRANSMIT BCD

Command: TRANSMIT BCD(R[23],3)
ASCII Characters transmitted: {not ASCII characters}
Decimal values: 4 19 148
Hex values: 04 13 94

Command: TRANSMIT BCD(R[23],1)
ASCII Characters transmitted: {not ASCII character}
Decimal values: 148
Hex values: 94

Command: TRANSMIT BCD(R[23],5)
ASCII Characters transmitted: {not ASCII characters}
Decimal values: 0 0 4 19 148
Hex values: 00 00 04 13 94

Command: TRANSMIT BCD(R[23], VARIABLE)
ASCII Characters transmitted: {not ASCII characters}
Decimal values: 4 19 148
Hex values: 04 13 94

Command: TRANSMIT BCD(R[23], VARIABLE R[600])
ASCII Characters transmitted: {not ASCII characters}
Decimal values: 4 19 148
Hex values: 04 13 94
R[600] would then equal 3.

ON RECEIVE message functions with register references

In the following ON RECEIVE examples it assumed that a WAIT follows immediately after the ON RECEIVE command, there are no other ON RECEIVEs set up for the WAIT and the incoming string is the following group of ASCII characters:

D876543F

Before the WAIT is executed, the following initial conditions are present:

UCM Register	Hex	Unsigned Decimal	Decimal	Octal	Binary
R[23]	A1B2	41394	-24142	120662	1010 0001 1011 0010
R[24]	03F5	1013	1013	1765	0000 0011 1111 0101

Several of the examples have remaining characters. The remaining characters will be received by the UCM and buffered until the next ON RECEIVE is reached by the program. This is not good programming practice unless these characters are meant to be handled elsewhere in the program. If they are not handled correctly, ON RECEIVEs later in the program may give unexpected results.

ON RECEIVE HEX

Command: ON RECEIVE HEX(R[23],4) RETURN

Results after WAIT:
Characters used: D876
Translated to: hex D876

	Hex	Unsigned decimal	Decimal	Binary
Register 23	D876	55,414	-10,122	1101 1000 0111 0110
Register 24	03F5	1,013	1,013	0000 0011 1111 1001

Remaining characters: "543F"

Command: ON RECEIVE HEX(R[23],8) RETURN

Results after WAIT:
Characters used: D876543F
Translated to: hex D876 and hex 543F

	Hex	Unsigned decimal	Decimal	Binary
Register 23	D876	55,414	-10,122	1101 1000 0111 0110
Register 24	543F	21,567	21,567	1001 1000 0011 1111

Note: Every character is used by this HEX function. The string was meant for a statement similar to this one, in that it handles all of the characters.

Command: ON RECEIVE HEX(R[23],2) RETURN

Results after WAIT:
Characters used: D8
Translated to: hex D8

	Hex	Unsigned decimal	Decimal	Binary
Register 23	00D8	216	216	0000 0000 1101 1000
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Remaining characters: "76543F"

ON RECEIVE DEC

Command: ON RECEIVE DEC(R[23],4) RETURN

Results after WAIT:
Characters used: D8765
Translated to: decimal 8,765

	Hex	Unsigned decimal	Decimal	Binary
Register 23	223D	8,765	8,765	0010 0010 0011 1101
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The first received character "D" is ignored by the DEC() function. This is all right but if a D is always the leading character then a program statement like ON RECEIVE "D":DEC(R[23],4) may be better.
Remaining characters: "43F"

Command: ON RECEIVE DEC(R[23],5) RETURN

Results after WAIT:
Characters used: D87654
Translated to: decimal 87,654%65,536 = 22,118

	Hex	Unsigned decimal	Decimal	Binary
Register 23	5666	22,118	22,118	0101 0110 0110 0110
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The first "D" is ignored similar to the previous ON RECEIVE..
Remaining characters: "3F"

Command: ON RECEIVE DEC(R[23],2) RETURN

Results after WAIT:
Characters used: D87
Translated to: decimal 87

	Hex	Unsigned decimal	Decimal	Binary
Register 23	0057	87	87	0000 0000 0101 0111
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The "D" is ignored as above.
Remaining characters: "6543F"

ON RECEIVE UNS

Command: ON RECEIVE UNS(R[23],4) RETURN

Results after WAIT:

Characters used: D8765
Translated to: unsigned decimal 8,765

	Hex	Unsigned decimal	Decimal	Binary
Register 23	223D	8,765	8,765	0010 0010 0011 1101
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: the first received character "D" is ignored by the UNS() function.
Remaining characters: "43F"

Command: ON RECEIVE UNS(R[23],5) RETURN
Results after WAIT:
Characters used: D87654
Translated to: unsigned decimal 87,654%65,536 = 22,118

	Hex	Unsigned decimal	Decimal	Binary
Register 23	5666	22,118	22,118	0101 0110 0110 0110
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The "D" is ignored. The next five characters "87654" do not make a valid unsigned decimal number and so the UNS() function takes the incoming number and does a modulus 65,536. In this case the result is 22,118.
Remaining characters: "3F"

Command: ON RECEIVE UNS(R[23],2) RETURN
Results after WAIT:
Characters used: D87
Translated to: decimal 87

	Hex	Unsigned decimal	Decimal	Binary
Register 23	0057	87	87	0000 0000 0101 0111
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The "D" is ignored.
Remaining characters: "6543F"

ON RECEIVE OCT

Command: ON RECEIVE OCT(R[23],5) RETURN
Results after WAIT:
Characters used: D876543
Translated to: octal 76543

	Hex	Unsigned decimal	Decimal	Binary	Octal
Register 23	7D63	32,099	32,099	0111 1101 0110 0011	076543
Register 24	03F5	1,013	1,013	0000 0011 1111 0101	001765

Note: The first two received characters "D8" are not octal digits and are ignored by the OCT() function.

Remaining characters: "F"

Command: ON RECEIVE OCT(R[23],2) RETURN

Results after WAIT:
Characters used: D876
Translated to: octal 76

	Hex	Unsigned decimal	Decimal	Binary	Octal
Register 23	003E	62	62	0000 0000 0011 1110	000076
Register 24	03F5	1,013	1,013	0000 0011 1111 0101	001765

Note: The "D" and the "8" are ignored.
Remaining characters: "543F"

Command: ON RECEIVE OCT(R[23],6) RETURN

Results after WAIT:
Characters used: D876543F
Translated to: nothing

	Hex	Unsigned decimal	Decimal	Binary	Octal
Register 23	A1B2	41,394	-24,142	1010 0001 1011 0010	120662
Register 24	03F5	1,013	1,013	0000 0011 1111 0101	001765

Note: Since "D", "8" and "F" are not valid octal characters they are lost by the OCT command. Between the "8" and the "F" the octal characters "76543" were received, which is only 5 characters instead of the 6 required by this ON RECEIVE. Since the next character "F" was not an octal character the previous 5 characters are ignored as not matching 6 octal characters in a row. So, not enough octal characters have been transmitted for this command. If this command is used without an ON TIMEOUT then the program will wait until 6 octal characters in a row are sent before completing this ON RECEIVE. Also note that register 23 has not yet changed.

Remaining characters: None - waiting for 6 octal characters in a row

ON RECEIVE BCD

Command: ON RECEIVE BCD(R[23],2) RETURN

Results after WAIT:
Characters used: D8 (hexadecimal 44 and 38)
Translated to: decimal 4,438

	Hex	Unsigned decimal	Decimal	Binary
Register 23	1156	4,438	4,438	0001 0001 1001 1010
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The first two received characters "D" and "8" are used by the BCD() function. The "D" is a hex character 44 and the "8" is a hex character 38 and so the unsigned decimal value is 4438.

Remaining characters: "76543F"

Command: ON RECEIVE BCD(R[23],4) RETURN

Results after WAIT:
Characters used: D876 (hexadecimal 44 38 37 36)
Translated to: decimal 44,383,736 converted to 15,864

	Hex	Unsigned decimal	Decimal	Binary
Register 23	6AF8	15,864	15,864	0110 1010 1111 1000
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: Both register 23 were changed
Remaining characters: None

ON RECEIVE RAW

Command: ON RECEIVE RAW(R[23],2) RETURN

Results after WAIT:
Characters used: D8 (hexadecimal 44 38)
Translated to: hexadecimal 4438

	Hex	Unsigned decimal	Decimal	Binary
Register 23	4438	17,464	17,464	0100 0100 0011 1000
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The "D" is an hex 44 and the "8" is a hex 38 so register 23 is now 4438
Remaining characters: "76543F"

Command: ON RECEIVE RAW(R[23],1) RETURN

Results after WAIT:
Characters used: D (hexadecimal 44)
Translated to: hexadecimal 4400

	Hex	Unsigned decimal	Decimal	Binary
Register 23	4400	17,408	17,408	0100 0100 0000 0000
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: The RAW function places the first character into the upper bits of the register and zeros the rest of the bits.
Remaining characters: "876543F"

Command: ON RECEIVE RAW(R[23],4) RETURN

Results after WAIT:
Characters used: D876 (hexadecimal 44 38 37 36)
Translated to: hexadecimal 4438 and 3736

	Hex	Unsigned decimal	Decimal	Binary
Register 23	4438	17,464	17,464	0100 0100 0011 1000
Register 24	3736	14,134	14,134	0011 0111 0011 0110

Note: RAW changed both register 23 and 24
Characters remaining: "543F"

ON RECEIVE BYTE

Command: ON RECEIVE BYTE(R[23]) RETURN

Results after WAIT:
Characters used: D (hexadecimal 44)
Translated to: hexadecimal 0044

	Hex	Unsigned decimal	Decimal	Binary
Register 23	0044	68	68	0000 0000 0100 0100
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: Only R[23] is changed.
Characters remaining: "876543F"

ON RECEIVE WORD

Command: ON RECEIVE WORD(R[23]) RETURN

Results after WAIT:
Characters used: D8 (hexadecimal 44, 38)
Translated to: hexadecimal 4438

	Hex	Unsigned decimal	Decimal	Binary
Register 23	4438	17,464	17,464	0100 0100 0011 1000
Register 24	03F5	1,013	1,013	0000 0011 1111 0101

Note: Only R[23] is changed.
Characters remaining: "76543F"

ON RECEIVE RWORD

Command: ON RECEIVE RWORD(R[23]) RETURN

Results after WAIT:
Characters used: D8 (hexadecimal 44, 38)
Translated to: hexadecimal 3843

	Hex	Unsigned decimal	Decimal	Binary
Register 23	3843	14,403	14,403	0011 1000 0100 0011
Register 24	03F5	1.013	1.013	0000 0011 1111 0101

Note: Only R[23] is changed.
Characters remaining: "76543F"

READ Examples

The READ statement allows SY/MAX compatible priority READ commands to be sent from UCM port under UCM program control. The READ statement is sent using static or dynamic routes.

Static Route READ

The static route read is used when it is not necessary to change the target of the read statement.

The statement

READ 1 (101, 45) 100, 130, 5

will send a READ out UCM port 1, into the network port 101, out the network port 045, and read the data in registers 130 through 135 in the attached device. The results of these registers will be placed in the UCM registers 100 through 105.

A convenient use for the static route read might be the following:

FOR R[30] = 1 to 5 STEP 2

READ 2 (101, 77, 24) 100+R[30], 100*R[30], 2

NEXT

This routine would copy registers 100, 101 in the remote device and place these in UCM registers 100 and 101, remote registers 300, 301 into UCM registers 103, 104, and remote registers 500, 501 into UCM registers 105 and 106.

If a route is not needed, such as when a SY/MAX device is directly connected to the UCM port, simply use ().

READ 3 () 2040, 3124, 5

Dynamic Route READ

The dynamic route read is similar to the indirect read in the SY/MAX processor. The UCM register R[<expr] in the route section of the READ provides a pointer to that UCM register which contains the number of drops in the route, N. The next N registers contain the actual drop numbers of the N drops. For example,

Assume: R[200] = 3
R[201] = 55
R[202] = 110
R[203] = 100 when the following READ is performed.

READ 1 R[200] 15, 25, 6

This READ will be sent out UCM port 1, into network port 55, through the Net-to-Net port 110 and out network port 100, read registers 25 through 31 in the remote device and return the data to UCM registers 15 through 21.

This type of READ is very powerful as it allows easy control of the drops and number of drops of a communication. For example, suppose UCM port 2 is connected to a SPE4 port in SY/MAX mode with a drop number of 30. On port 31 of the SPE4 is a network of 6 PowerLogic Circuit Monitors with addresses 01 through 06. (This port is in PLOGIC mode.) Assume R[1000] = 3, R[1001] = 30, and R[1002] = 31. The following loop will read registers 1 through 28 in each CM an place them in registers 11 through 183.

FOR R[10] = 1 to 6

R[1003] = R[10]

```
        READ 2 R[1000] 11+(R[10]-1)*28, 1, 28
NEXT
```

WRITE Examples

The WRITE statement allows SY/MAX compatible priority WRITE commands to be sent from UCM port under UCM program control. The WRITE statement is sent using static or dynamic routes.

Static Route WRITE

The static route write is used when it is not necessary to change the target of the write statement.

The statement

```
        WRITE 1 (101, 45) 100, 130, 5
```

will send a WRITE out UCM port 1, into the network port 101, out the network port 045, and write the data from UCM registers 100 through 105 to registers 130 through 135 in the attached device.

A convenient use for the static route write might be the following:

```
FOR R[30] = 1 to 5 STEP 2
        WRITE 2 (101, 77, 24) 100+R[30], 100*R[30], 2
NEXT
```

This routine would copy registers 100, 101 in the UCM to registers 100 and 101 in the remote device, UCM registers 102, 103 into remote registers 300, 301, and UCM registers 105, 106 into remote registers 500 and 501.

If a route is not needed, such as when a SY/MAX device is directly connected to the UCM port, simply use ().

```
WRITE 3 () 2040, 3124, 5
```

Dynamic Route WRITE

The dynamic route write is similar to the indirect write in the SY/MAX processor. The UCM register R[<expr] in the route section of the WRITE provides a pointer to that UCM register which contains the number of drops in the route, N. The next N registers contain the actual drop numbers of the N drops. For example,

```
Assume:  R[200] = 3
         R[201] = 55
         R[202] = 110
         R[203] = 100 when the following WRITE is performed.

WRITE 1 R[200] 15, 25, 6
```

This WRITE will be sent out UCM port 1, into network port 55, through the Net-to-Net port 110 and out network port 100, copy UCM registers 15 through 21 into remote device’s registers 25 through 31.

PRINT Examples

The network PRINT statement allows easy use of the UCM's message commands for sending data out SY/NET peripheral ports. Any message that may be sent with the TRANSMIT statement may be sent with the PRINT statement.

Suppose a modem is connected to peripheral drop 101. The UCM port 1 is connected to SY/MAX drop 154. To send the Hayes command ATDT5551212 to the modem simply use the command:

PRINT 1 (154, 101) "ATDT5551212"

To hang up the modem later, use the following:

DELAY 100

PRINT 1 (154, 101) "+++"

DELAY 100

PRINT 1 (154,101) "ATH0"

and the modem will return to command mode and hang up.

The PRINT command also has the dynamic route like the READ and WRITE statements.

9
Compiling

COMPILE.EXE

COMPILE.EXE is an MS-DOS compatible program for compiling the UCM configuration text file into machine readable code. All UCM configurations must be compiled before they can be downloaded into the UCM. The downloading is done by another MS-DOS compatible program UCMLOAD.EXE described in a later section of the manual.

The COMPILE command syntax is as follows:

COMPILE filename[*.ext*] [-O*file2*] [-D*macro=string*] [-L*file3*]

Where filename refers to the text file containing the source code for the UCM.

The *.ext* is an optional extension to the filename. If no extension is included then .UCM is assumed by the compiler.

Options can appear in any order.

-O option

The -O option is for specifying an output file other than filename.ucc. If the -O option is not used then COMPILE will create the output file filename.ucc. If the -O option is used then COMPILE will create an output file named *file2*. If an extension is desired for *file2* it needs to be added since no extension is assumed by the compiler.

-D option

The -D option is for specifying DEFINE macros at compile time. This is very useful for compiling one UCM configuration file for 2 or more ports of the same UCM module. The *macro* portion of the -D option is the string inside of the UCM configuration file that is to be found while *string* portion is *macro*'s replacement. It is equivalent to Find what: *macro* Change to: *string* in DOS EDIT.

If, in the configuration file AMAZING.UCM, the word Time has been used and Time needs to have a value of 50 then the DOS command to compile AMAZING with the Time replacement is:

```
COMPILE AMAZING -DTime=50
```

If the compile completes with no errors then the output file AMAZING.UCC will be created. If more than one DEFINE is needed at compile time then they can be added to the end of the COMPILE command as in:

```
COMPILE AMAZING -DTime=50 -DPort=1 -DFlavor=strawberry
```

-L option

The -L option is for telling the compiler to also generate a 68000 source listing. The name of the DOS text file is *file3*. If an extension is desired for *file3* it needs to be added since no extension is assumed by the compiler.

The 68000 source listing, *file3*, is a text file that can be read by your favorite text editor. If you have any questions about the way the compiler generates code for the UCM then you can use the -L option. Most users will not have a use for this option.

Compiler Errors

When the UCM configuration file contains code that the compiler does not recognize, variables out of range, code that is too long or any other error then the compiler generates an error listing. This listing will have the compiler error number, the line number in the .UCM file where the error occurred, a copy of the line in question, and a description of the error. The listing will also summarize the total number of errors detected.

The programmer can use this listing to correct problems in the UCM configuration file. Since no object code is generated if an error occurs during the compile, all errors must be repaired before a valid object file can be made for downloading into the UCM.

Debugging

For debugging purposes the user may want to store the error listing in a file in order to refer to it later. This can be accomplished with the output redirection feature of DOS. For example:

```
COMPILE filename >error.lst
```

The text that normally would go to the screen will now appear in the text file **error.lst**.

A complete listing of the compiler errors appears in Chapter 10 - Compiler Error Listing.

10
Compiler Error Listing

ERROR 1 - Expression expected for function parameter
example: Transmit HEX(,2)
Functions require 1 or more expressions. i.e. HEX(R[20],2).

ERROR 2 - Comma expected after function parameter
example: MIN(R[23] 9)
Functions with 2 or more expressions require commas to separate expressions i.e. MIN(R[23],9).

ERROR 3 - Right parenthesis expected after last function parameter
example: SWAP(R[45]
Functions require parenthesis around the expressions i.e. SWAP(R[45])

ERROR 4 - Parameter must be a register reference
example: RAW(8*4,2)
The RAW function requires a register reference i.e. RAW(R[32],2)

ERROR 5 - Register number expression expected
example: R[]
An expression must go between the parenthesis R[<expr>]

ERROR 6 - Register number not in 1..2048
examples: R[2049] or R[0]
There are only 2,048 registers in the UCM. Do not try to use registers outside of this range.

ERROR 7 - Right bracket missing from register reference
example: R[15] = R[9
Registers take the form R[<expr>] with the brackets [] required.

ERROR 8 - Closing quote missing in literal string
example: TRANSMIT "Hello
Literal strings are enclosed in quotes "<string>". A \" can be used to embed a quote inside the string.

ERROR 9 - Unmatched parenthesis in message description
example:

ERROR 10 - Message description expected
An ON CHANGE or ON RECEIVE has been used without any message.

ERROR 11 - Parameter must be a register for parse mode
The first <expr> of a message function (HEX, DEC, UNS, OCT, BCD) has been used in an ON RECEIVE instead of (<expr>). To generate and match <expr> surround it with parenthesis, i.e. HEX((R[23]),2).

ERROR 12 - Unmatched parenthesis in arithmetic expression
example:

ERROR 13 - GOTO or RETURN expected
An ON RECEIVE or ON CHANGE has been used without a GOTO or RETURN.

ERROR 14 - Label expected after GOTO
A label is missing in a GOTO. Should be GOTO <label>.

ERROR 15 - GOTO undefined label
example: GOTO Jail
A label has been referenced in a GOTO statement that is not in the UCM file. Be sure that the label ends in a colon i.e. Jail:.

ERROR 16 - Timeout value missing
example: ON TIMEOUT GOTO Jail
ON TIMEOUT requires an expression i.e. ON TIMEOUT 10 GOTO Jail

ERROR 17 - Colon missing after label or statement misspelled
example: main
Colons are needed after labels i.e. main:.

ERROR 18 - Arithmetic expression expected in assignment
example:

ERROR 19 - Equal expected in assignment
example:

ERROR 20 - Constant expected for translation number
example:

ERROR 21 - Translation number not in 1..8
example:

ERROR 22 - Colon expected after translation number
example:

ERROR 23 - Equal expected in translation statement

example:

ERROR 24 - Statement expected
example:

ERROR 25 - Translation too long
example:

ERROR 26 - Messages are limited to 256 characters
example:

ERROR 27 - Expression too complex. Use register for intermediate variable
example:

ERROR 28 - Literal translation string expected
example:

ERROR 29 - Expecting non-empty translate from string
example:

ERROR 30 - Expecting translate to string
example:

ERROR 31 - Constant out of range -32768..65535
example:

ERROR 32 - String function width over than 64 characters
example:

ERROR 33 - String function width negative
example:

ERROR 34 - SET statement must be followed by constant
example:

ERROR 35 - Control code in literal string (use \\xx notation) or missing quote
example:

ERROR 36 - SET value is out of range
example:

ERROR 37 - Object code is too large
example:

ERROR 38 - The compiler ran out of memory
example:

ERROR 39 - Output message will be too long
example:

ERROR 40 - Input message will be too long
example:

ERROR 41 - On change must be followed by a register reference
example:

ERROR 42 - Constant register number required
example:

ERROR 43 - Expression expected for bit number
example:

ERROR 44 - Reserved but non-implemented statement

ERROR 45 - SET missing

ERROR 46 - SET not followed by setable parameter
SET needs to be followed by either BAUD, PARITY, STOP, CAPITALIZE, DEBUG or R[<expr>].<const>.

ERROR 47 - ON missing
CHANGE, TIMEOUT and RECEIVE need to be preceded by ON.

ERROR 48 - ON not followed by appropriate parameter
ON needs to be followed by either RECEIVE, CHANGE or TIMEOUT.

ERROR 49 - Logical constant must be FALSE (0) or TRUE (1)
example:

ERROR 50 - Function name should be followed by left parenthesis
example:

ERROR 51 - Expected delay time expression
example:

ERROR 52 - Logical expression required
example:

ERROR 53 - Goto or gosub expected in if statement
example:

ERROR 54 - Statement is unreachable
Code has been written that in a section that does not have a label.

ERROR 55 - IF not followed by GOTO, GOSUB, or THEN
example:

ERROR 56 - Equal expected in FOR
example: FOR R[2] 1 TO 16
An equal sign is required after the register, i.e. FOR R[2] = 1 to 16.

ERROR 57 - Expression expected in FOR
example: FOR R[2] = TO 16
FOR statement requires start and end values, i.e. FOR R[2] = 1 TO 16.

ERROR 58 - TO expected
example: FOR R[2] = 1 16
FOR statements require TO, i.e. FOR R[2] = 1 TO 16.

ERROR 59 - Register reference expected in FOR
example: FOR N = 1 to 16
FOR loops require their counters to be registers. N can be defined by using DEFINE statements in the UCM file, i.e. DEFINE N = R[2], or by using the -D option of the compiler, i.e. COMPILE TEST -DN=R[2]

ERROR 60 - Missing WEND
example: WHILE R[9]<5 R[8]=R[8]*R[8] R[9]=R[9]+1
WHILE <logical> one or more statements WEND is the structure of a WHILE, i.e. WHILE R[9]<5 R[8]=R[8]*R[8] R[9]=R[9]+1 WEND

ERROR 61 - Missing ENDIF
An IF <logical> THEN has been followed by a *newline* but no ENDIF. No ENDIF is required if statements are made on the same line as the IF but if a *newline* is used after the THEN and before a statement then an ENDIF is required.

ERROR 62 - Missing NEXT
A FOR has been used without a NEXT. Each FOR statement requires a NEXT statement.

ERROR 63 - Missing UNTIL
A REPEAT statement has been used without an UNTIL. The format is REPEAT on or more statements UNTIL <logical>.

ERROR 64 - No statement in ELSE clause

ERROR 65 - Duplicate label
A label name has been used twice in the same UCM file. Only the first 20 characters of a string are used by the compiler so strings longer than this are truncated to the first 20 characters and may then match.

ERROR 66 - Bit selector operator (.) not followed by constant or left parenthesis

example:

ERROR 67 - Bit number out of range 1..16
example: TOGGLE R[2].0
Register bits are numbered from 1 to 16. Any constant expression that evaluates outside this range is an invalid bit number.

ERROR 68 - Attempt to DEFINE a reserved word
example: DEFINE END=16
Reserved words can not be used in DEFINE statements.

ERROR 69 - Insufficient memory for macro definition
example:

ERROR 70 - Missing = in macro definition
example:

ERROR 71 - Macro expansion requires too much memory
example:

ERROR 72 - Division by 0
example: R[12]/((2*3)-6)
The compiler evaluates constant expressions at compile time to reduce UCM processor time. If the denominator evaluates to 0 then this error occurs at compile time.

ERROR 73 - Bit operator not followed by register reference
example: CLEAR 1
The bit operators (SET, CLEAR & TOGGLE) must be followed by a register reference and a bit number reference i.e. CLEAR R[2].1.

ERROR 74 - Bit delimiter (.) missing in bit operation
example: SET R[2]1
A period is required by the bit operators (SET, CLEAR & TOGGLE) between the register reference and the bit number, i.e. SET R[2].1.

ERROR 75 - Right parenthesis missing after dynamic bit number
example:

11
Local Registers

Table 11-1 Module Register List

Register	Legal Values	Function
1	0..F (hex)	Command Register Bits 1..4 control Ports 1..4
2	any	Default location for Error Register for Port 1.
3	any	Default location of Line Number Register for Port 1.
4	any	Default location for Error Register for Port 2.
5	any	Default location of Line Number Register for Port 2.
6	any	Default location for Error Register for Port 3.
7	any	Default location of Line Number Register for Port 3.
8	any	Default location for Error Register for Port 4.
9	any	Default location of Line Number Register for Port 4.
10..2048	any	User Variables
2049	0..4	Port Number for program loading.
2050	any	Number of bytes loaded.
2051..7049	any	Program loading area.
7050..8175		Reserved for future use, do not modify
8176	8001..8004 (hex)	Value indicates the UCM Port number to which the reading device is attached. 8000 + Port # (Read only)
8177..8186		Packed ASCII message giving the module's name and firmware revision. (Read only)
8187		Status of CTS on Ports 1..4. (Read only)
8188	999X (hex)	NR&D Module Identifier. The last digit indicates the command register setup for Auto-start upon power up.
8189	1..2047	Pointer to the location of the Status Register Pair for Port 1.
8190	1..2047	Pointer to the location of the Status Register Pair for Port 2.
8191	1..2047	Pointer to the location of the Status Register Pair for Port 3.
8192	1..2047	Pointer to the location of the Status Register Pair for Port 4.

12
Connector Pinouts

RS-422 ports on UCM4-D (DE9S with slide lock posts)

1 TX- transmit data (inverted) from UCM4 to output device

2 TX+ transmit data (noninverted) from UCM4 to output device

3 RX- receive data (inverted) from data source to UCM4

4 RX+ receive data (noninverted) from data source to UCM4

5 CTS- must be more negative than pin 7 to allow UCM4 to transmit

6 RTS- driven low (0V) when UCM4 is ready to receive data

7 CTS+ must be more positive than pin 5 to allow UCM4 to transmit

8 RTS+ driven high (+5V) when UCM4 is ready to receive data

9 Shield ground. AC coupled to the chassis.

RS-232 ports on UCM4-S (DE9P with screw lock posts)

1 CD input no connection in UCM4

2 RXD input serial data from external device

3 TXD output serial data to external device

4 DTR output pulled up to +10 volts internally

5 SG signal ground

6 DSR input no connection in UCM4

7 RTS output normally high (+10), driven low (-10) when internal buffers are filling faster than data can be processed.

8 CTS input must be high for multiplexer to be able to transmit. This pin is usually connected to DTR of a printer. Ensure that the attached device can provide this or

tie it to pin 4 (DTR).

9 RI input no connection in UCM4

RS-485 ports on a UCM4-M (DE9S with slide lock posts)

1 TX- transmit data (inverted) from UCM4 to output device

2 TX+ transmit data (noninverted) from UCM4 to output device

3 RX- receive data (inverted) from data source to UCM4

4 RX+ receive data (noninverted) from data source to UCM4

5 CTS- must be more negative than pin 7 to allow UCM4 to transmit

6 RTS- driven low (0V) when UCM4 is ready to receive data

7 CTS+ must be more positive than pin 5 to allow UCM4 to transmit

8 RTS+ driven high (+5V) when UCM4 is ready to receive data

9 Shield ground. AC coupled to the chassis.

RS-422 port on a UCM1-D (DE9S with slide lock posts)

1 TX- transmit data (inverted) from UCM1 to connected device

2 TX+ transmit data (noninverted) from UCM1 to connected device

3 RX- receive data (inverted) from connected device to UCM1

4 RX+ receive data (noninverted) from connected device to UCM1

5 +5 VDC

6 +5 VDC

7 Logic ground and +5V return

8 Logic ground

9 Shield ground. AC coupled to the chassis

13
Recommended Cabling

Cabling required to configure an UCM

Configuration files are downloaded from an MS-DOS personal computer into the UCM. The factory default configuration for the module is that all ports not running a user program are SY/MAX, 9600 baud, 8 data bits, EVEN parity, 1 stop bit which may be used for downloading user programs or for viewing and modifying UCM registers. The correct cabling needs to be installed to connect the personal computer to an UCM port.

UCM-D to personal computer cabling

Connecting the RS-422 UCM-D is very easy using Niobrara's **SC406** (or SC902) RS-232 to RS-422 converter cable. If the personal computer has a 25-pin RS-232 port then no adapters are needed. If the personal computer has a 9-pin RS-232 port then a Niobrara **SD034** 25-pin to 9-pin adapter is needed for the SC406. The **SC902** cable will plug directly into the 9-pin port of the personal computer so no adapter is needed.. If the UCM is an UCM**1**-D then the SC406 gets power from the RS-422 port of the module. If the UCM is an UCM**4**-D then an AC adapter, which is included with the cable, is needed to power the SC406 and SC902.

UCM4-S to personal computer cabling

If the UCM is a -S then a straight connection to the RS-232 port of the PC may be made to any of the RS-232 ports of the module.

UCM-M to personal computer Cabling

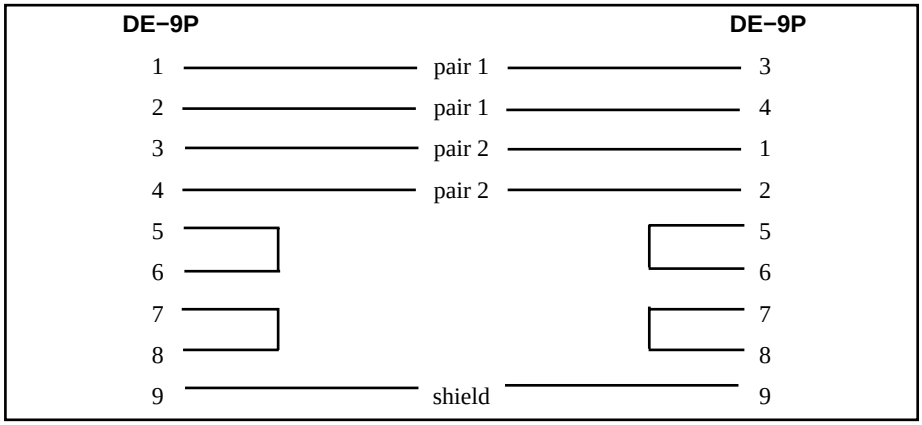
If the UCM is a -M an SC406 (or SC902) can be used to connect the PC comm port to an UCM port.

Note: The included power supply with the SC406 (or SC902) must be used when connecting to ports 1-4 of the UCM4-D or UCM4-M. The power supply is not needed when connecting to the RS-422 port of the UCM1-D or the UCM1-M. The **SC406** (or **SC902**) RS-232 to RS-422 converter cable may be used whenever a single RS-232

port is required on a UCM-D or when a single RS-422 port is needed on an UCM4-S. It should be noted that the SC406 (or SC902) does not support handshaking and that functionality of certain features of the UCM may not be implemented. But in most cases this will not be a concern.

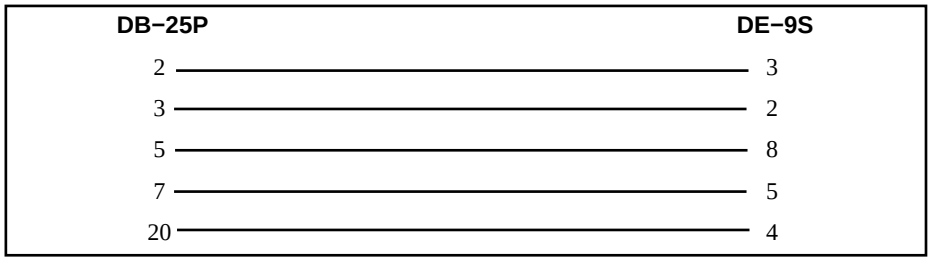
Cabling required to connect a UCM port to an external device

UCM-D RS-422 to SY/MAX RS-422 port

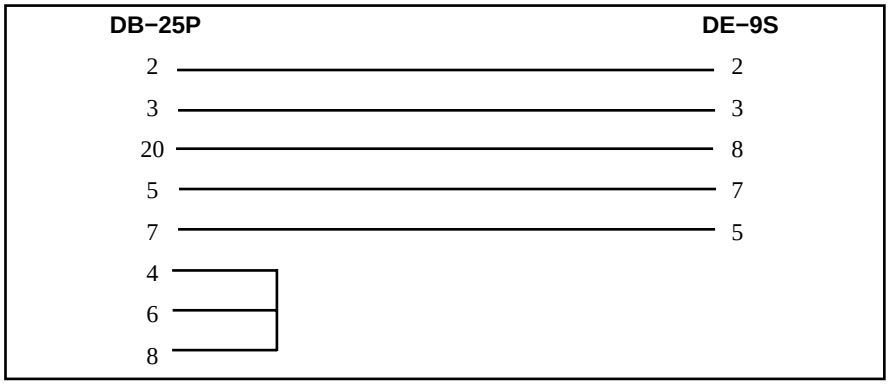


This is a Niobrara **DC1** cable.

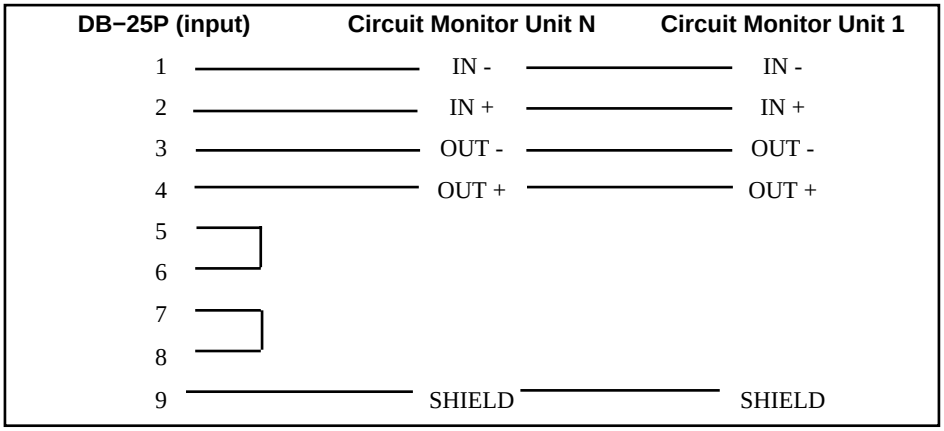
RS-232 DCE (modem) to UCM4-S RS-232 port



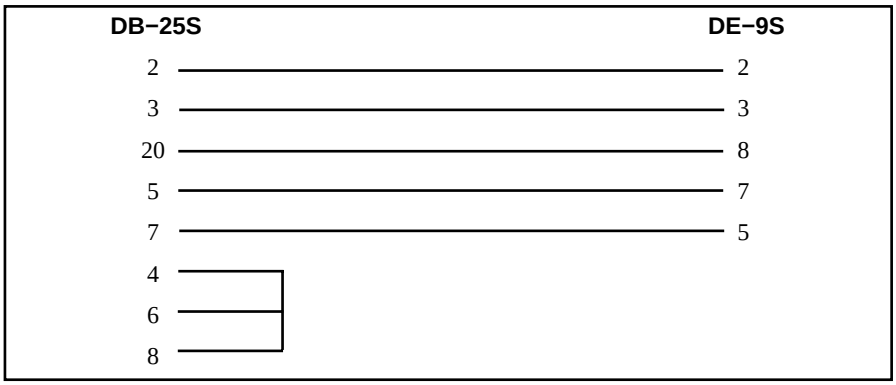
RS-232 DTE (terminal) to UCM4-S RS-232 port



UCM-D RS-422 port to PowerLogic® RS-485



Male RS-232 DTE (personal computer) to UCM4-S RS-232 port



Appendix A
Overview of UCM Demo Programs

An overview of the UCM configuration files DEMO1.UCM through DEMO4.UCM.

There are 4 files that demonstrate UCM configuration files and several features of the UCM language. They are:

DEMO1.UCM (Brute force method for one port)
DEMO2.UCM (Uses registers as variables to reduce the length of the code)
DEMO3.UCM (Uses DEFINE command to make code more readable)
DEMO4.UCM (A single configuration file that works for multiple ports by using compiler variables)

UCM configuration functional description

There are 2 units connected to one port of the UCM. The PLC writes commands destined for Unit 1 into register 10 and commands for Unit 2 into register 11.

Once a command is received (register 10 or 11 has changed), the UCM puts together a message and sends it to the appropriate unit. The message is constructed as follows:

STX Unit# Command EOT Checksum CR

The unit receives the command and, if the checksum is OK and the command is valid, attempts to execute the command. If the unit performs the command correctly the unit replies as follows:

Unit# Command ACK Checksum CR

If the checksum is bad, an illegal function has been requested, or an error occurs as the unit is attempting a valid command the reply is as follows:

Unit# Command Errorcode NAK Checksum CR

After a response is received from the unit or there is no response for 50 mS after 3 attempts, register 12 is set by the UCM to report the results.

Once the status has been reported by the UCM, the PLC must then clear the control-

ling command register (R[10] or R[11]) and the UCM will respond by clearing the status and error response registers (R[12]..R[14]). The process is ready to start again with a command register written by the PLC.

DEMO1.UCM description:

The file DEMO1.UCM, when compiled and loaded into the UCM, allows port 1 to meet the functional description described above in a brute force method. The following is a line by line description of the file DEMO1.UCM.

The file starts off with register and variable descriptions which are comments. Comments are enclosed in braces {} and can be many lines long. Comments are ignored by the compiler so they can be anywhere in the program including in the middle or end of a line of code.

The first line of code that the UCM would see is the SET command. Here the port is configured for communication with the Units. When the program is not running the port is configured as a SY/MAX compatible port.

The next line of code is the label *Main_loop:*. Labels are always followed by colon :.

The next three lines of code work together. ON CHANGE, ON TIMEOUT and WAIT always work together. The ON CHANGE and ON TIMEOUT set up the UCM for the WAIT statement. As many ON CHANGEs or ON TIMEOUTs can be used as needed for the application. Once the UCM sees the WAIT command it waits until one of the ON CHANGE or ON TIMEOUT conditions is met. In this case the UCM could wait forever since there is no ON TIMEOUT. Once register 10 (R{10]) or register 11 (R[11]) changes the UCM jumps to the label *write_unit_1* or the label *write_unit_2* respectively.

The next line of code is the label *write_unit_1*. If R[10] has changed then the PLC has written a command destined for Unit 1 and the UCM will have jumped to this label. R[2048] is the retry counter and is set to 3 for this application. The line *retry_1*: is a label for retrying when a unit does not respond in the correct amount of time. Notice that the UCM will follow from setting the retry counter through the *retry_1* label to decrimenting the retry counter.

The next line checks to see if the message has been sent three times yet. If so, then R[2048] will be less than 0 and the UCM will jump to the label *no_reply_1*. If not, the UCM will continue with the TRANSMIT line.

The TRANSMIT command is a very powerful method for implementing communication protocols. It allows the programmer to set up a complete message, including data conversions and checksums all in one line of code. In this example, the protocol of command request for the Units is implemented in this single line of code. The "\02" is a literal 2 digit hex number 02 or start of text (STX). Any part of a transmit or receive that is in quotes is a literal value and any literal value preceded by a \ is a hex number. The colons in the transmit and receive commands concatenate the message together. The next "1" is a literal ASCII 1 (or a hex 31 or a decimal 49).

The next part of the TRANSMIT statement is HEX(R[10],1) which takes register 10 and makes a single character out of the least significant byte. If it had been HEX(R[10],2) then two characters would have been made from the least significant byte of R[10]. If it had been HEX(R[10],4) then four characters would have been made from the single register R[10]. There are similar functions for decimal (DEC), unsigned (UNS), octal (OCT), and binary coded decimal (BCD).

The next part of the TRANSMIT "\03" is a literal 2 digit hex number 03 or an end of text character (EOT). The next part, BYTE(LRC(1,4,0)), needs to be broken up into two parts. The LRC(1,4,0) calculates a longitudinal redundancy check of bytes 1 through 4 (the STX through the EOT) with a starting value of zero. The BYTE() makes a single BYTE (a two digit hex number) out of this checksum. If the checksum is greater than hex FF, say hex 13B, then the BYTE() would produce 3B. The "\0D" is a literal 2 digit hex number 0D (decimal 13) or a carriage return character (CR).

The next 4 lines work together as described earlier. This time there are two ON RECEIVEs and one ON TIMEOUT before the WAIT. The UCM either receives the first string, receives the second string or it waits a total of 50 mS before it jumps to another location. If characters other then those described in the strings are received by the UCM they are ignored.

If the first string is received, the UCM jumps to the label *good_reply_1*. If the second string is received, the UCM jumps to the label *nack_1*. If 50 mS expire and neither string has been completely received or there has been a CRC error then the UCM jumps to the label *retry_1*.

The ON RECEIVE command is a very powerful way to implement the receiving end of a communications protocol. It works much the same as the TRANSMIT in that the expected protocol can be written in a very few ON RECEIVE commands. In this case the first ON RECEIVE is the format for a good response from the Unit with no errors. The second ON RECEIVE is the format of an error type response from the Unit with the error encoded into the response.

The first part of the first ON RECEIVE "1" expects an ASCII 1 (hex 31 or decimal 49). The HEX((R[10]),1) expects to match the value that is in register 10. Notice that there is a set of parenthesis around the R[10]. This tells the UCM to match the value in R[10]. If the parenthesis were missing then it would tell the UCM to put the next character into register 10.

The "\06" is a literal 2 digit hex number 06 or an acknowledge (ACK). The BYTE((LRC(1,3,0)) is broken into two parts as above. The LRC(1,3,0) calculates a longitudinal redundancy check of characters 1 through 3 with a start value of zero (from the "1" through the ACK). The parenthesis around the LRC means to match and the BYTE means only match a single byte (throw away any part greater than hex FF as above). Also as before, the "\0D" is a carriage return.

The second ON RECEIVE is similar to the first except it expects one extra character, the errorcode, which it interprets as a hex digit and puts into register 13 in the section HEX(R[13],1). Notice that there are no parenthesis around the R[13]. The ACK is replaced by a NAK (hex 15) and the LRC is calculated on four characters instead of

three in LRC(1,4,0) (from the "1" through the NAK).

The 5 in the ON TIMEOUT is 50 mS. A 1 is 10 mS and a 400 is 4,000 mS or 4 seconds.

The UCM reaches the next label *good_reply_1* by jumping here from the first ON RECEIVE described above. Register 12 is then set to 1 and the UCM jumps to *wait_for_clear*.

The UCM reaches the next label *nack_1* by jumping here from the second ON RECEIVE described above. R[12] is then set to 2 and the UCM jumps to *wait_for_clear*.

The next label, *no_reply_1*, is reached if no reply is received from Unit 1 after three attempts. R[12] is set to 4. The UCM falls through to the next label *wait_for_clear*.

The ON CHANGE and the WAIT work together. No ON TIMEOUT means the UCM will wait until it sees register 10 change. Once R[10] has been changed by the PLC the UCM will jump to the label *clear_status*.

After the UCM reaches the *clear_status* label, register 12, R[13] and R[14] are all set to zero. This lets the PLC know that the UCM is ready to receive another command. The next line, GOTO *Main_loop* jumps back to near the beginning of the program to start the cycle over again.

The rest of the program, from *write_unit_2*: through the WAIT in *no_reply_2*, works the same as described in the sections for *write_unit_1* through *no_reply_1*. The differences are:

- The TRANSMITs and ON RECEIVEs use 2 instead of 1 for the unit number and R[11] instead of R[10] as the command register.
- The first ON RECEIVE jumps to *good_reply_2* instead of *good_reply_1*.
- The second ON RECEIVE uses R[14] in place of R[13] for the communication error register and jumps to *nack_2* instead of *nack_1*.
- The ON TIMEOUT jumps to *retry_2* instead of *retry_1*.
- R[12] is set to 256 instead of 1 (*good_reply_2*), 512 instead of 2 (*nack_2*) and 1024 instead of 4 (*no_reply_2*).
- The PLC needs to change R[11] instead of R[10] to let the UCM know when it is finished with the data (*wait_for_clear_2*).

DEMO2.UCM description:

The DEMO2.UCM configuration file is functionally equivalent to DEMO1.UCM but requires less code and thus less memory inside of the UCM module. Since the *write_unit_1* through *no_reply_1* sections are so similar to the *write_unit_2* through *no_reply_2* sections of DEMO1.UCM, these sections have been combined and 3 new registers have been added for variable storage.

The sections *write_unit_1* and *write_unit_2* now set the variables for the write: section of the program. Register 2047 holds the unit number of interest, register 2046 holds

the number of the command register and register 2045 holds the number of the error register. Both sections call on the *write:* section to accomplish the TRANSMIT and ON RECEIVEs.

If R[2047], R[2046] and R[2045] are replaced with their respective values in the *write:* section, then this section would read just like either the *write_unit_1* or *write_unit_2* section with two exceptions. In the TRANSMIT, where the unit number previously was either a "1" or a "2" it is now HEX(R[2047],1) which will produce either a "1" or a "2" if R[2047] is equal to 1 or 2. In the ON RECEIVE sections, the "1" or "2" has been replaced with HEX((R[2047]),1) which matches the value in R[2047] instead of placing the character received into register 2047 and so is functionally equivalent.

The duplicate *good_reply_1* & *good_reply_2*, *nack_1* & *nack_2*, *no_reply_1* & *no_reply_2* have all been replaced with single sections *good_reply*, *nack* and *no_reply* that make use of variable registers. Comparing the previous sections with the new sections is a straightforward exercise.

DEMO3.UCM Description:

The DEMO3.UCM file takes the previous DEMO2.UCM file and uses the DEFINE command to make the code easier to read. The syntax of the command is DEFINE string_1=string_2. The DEFINE is interpreted by the compiler as a find and replace. Find string_1 and replace it with string_2. It is very much like doing a find and replace inside of a text editor.

Besides being easier to read it is also easier to write configuration files using the DEFINE command. The programmer does not have to remember numbers for each variable but can remember a name. For example it is easier to remember that the command register is called Command then to remember it is R[2046]. Note that the UCM ignores the case of a letter unless it is inside of quotes. So Command is the same as command but "Command" is not the same as "command".

There are 14 DEFINE commands used in DEMO3.UCM. They are right after the SET command. The file DEMO2.UCM could be generated by editing DEMO3.UCM, doing 14 find and replace commands starting with Find: Command_1 and Replace with: R[10] and ending with Find: CR and Replace with: "\0D" and erasing the DEFINE command lines.

The UCM requires a configuration file for each port that will be used. Since the registers listed in a configuration file are actual rack addressable registers then the registers used in one configuration would not normally be used in another configuration file inside of the same UCM. If several ports of the UCM were going to be connected to similar equipment then the configuration files would be similar except for the registers used.

DEMO3.UCM could be edited with a text editor and the registers changed so that the configuration could be used in another UCM port. This will work fine but if 4 ports are used then there are 4 configuration files that will need to be created. Careful debugging of the first file could keep changes to a minimum but if the application

changes in the future then there are 4 files to update and maintain. An easier method to create multiple configuration files is described in the next section.

DEMO4.UCM Description:

The UCM configuration file DEMO4.UCM has been changed to allow it to be used in any port. The port and control registers are set by using compiler variables at compile time. This compiler feature reduces the amount of time the programmer spends developing and debugging configuration programs and also reduces the number of files that must be maintained once in production.

The compiler variables are very similar to the DEFINE command described in the DEMO3.UCM DESCRIPTION section but instead of being defined when the configuration file is written they are defined at compile time. The user types the command COMPILE followed by the define option -Dstring1=string2. Whenever string1 is found in the configuration file it is replaced by string2.

When compiling DEMO4 for Port 1 the PLC control registers are 10 through 14 and the variables are stored in registers 2,048 through 2,045. In the program the variable Base is the first control register and the variable Vbase is the largest variable register. The DOS command to create a UCC file for downloading into Port 1 of the UCM is:

```
COMPILE demo4 -Odemop1.ucc -DBase=10 -DVbase=2048
```

The compile program will create a file DEMOP1.UCC which is ready to download into port 1 of the UCM. Also at compile time the variables Base and Vbase are replaced with the numbers 10 and 2,048 respectively. The DOS command to download this program into the UCM would be:

```
UCMLOAD 1 demop1 com1:
```

And likewise the DOS commands for Port 2 are:

```
COMPILE demo4 -Odemop2.ucc -DBase=15 -DVbase=1948
UCMLOAD 2 demop2 com1:
```

The file DEMOP2.UCC is created by the compile program. In this case the control registers for port 2 are 15 through 19 and variables are stored in registers 1948 through 1945.

DOS commands for Port 3:

```
COMPILE demo4 -Odemop3.ucc -DBase=20 -DVbase=1848
UCMLOAD 3 demop3 com1:
```

File created: DEMOP3.UCC Control registers: 20-24 Variable registers: 1848-1845

DOS commands for Port 4:

```
COMPILE demo4 -Odemop4.ucc -DBase=25 -DVbase=1748
UCMLOAD 4 demop4 com1:
```

File created: DEMOP4.UCC Control registers: 25-29 Variable registers: 1748-1745

To install 2 units onto each of the 4 ports of the UCM (8 total units) requires 29 rack address registers. The register map is as follows:

R[1] = UCM Program Control Register
R[2] = Port 1 status register
R[3] = Port 1 line number register
R[4] = Port 2 status register
R[5] = Port 2 line number register
R[6] = Port 3 status register
R[7] = Port 3 line number register
R[8] = Port 4 status register
R[9] = Port 4 line number register
R[10] = Command register for Unit 1 Port 1
R[11] = Command register for Unit 2 Port 1
R[12] = Unit response status Port 1
R[13] = Unit 1 communication error Port 1
R[14] = Unit 2 communication error Port 1
R[15] = Command register for Unit 1 Port 2
R[16] = Command register for Unit 2 Port 2
R[17] = Unit response status Port 2
R[18] = Unit 1 communication error Port 2
R[19] = Unit 2 communication error Port 2
R[20] = Command register for Unit 1 Port 3
R[21] = Command register for Unit 2 Port 3
R[22] = Unit response status Port 3
R[23] = Unit 1 communication error Port 3
R[24] = Unit 2 communication error Port 3
R[25] = Command register for Unit 1 Port 4
R[26] = Command register for Unit 2 Port 4
R[27] = Unit response status Port 4
R[28] = Unit 1 communication error Port 4
R[29] = Unit 2 communication error Port 4

Reducing the rack address space of the UCM

The default location of the UCM status registers is 2 through 9. If, in your application, rack address space is at a premium or the rack scan time needs to be reduced then the UCM status registers can be moved from registers 2 through 9 to other registers that do not have to be rack addressed. The UCM status registers are very useful for debugging your application but may not be required for operation. The first register of the UCM, the program control register, cannot be moved.

Using compiler variables in the UCM configuration file, as in example DEMO4.UCM, allows the programmer to utilize the UCM status registers while debugging and then reduce the rack address space as required by the application. If the application has two units connected to each of 4 ports of the UCM the DOS commands for compiling and loading the configurations into the UCM are as follows:

```
COMPILE demo4 -Odemo4p1.ucc -DBase=2 -DVbase=2048
COMPILE demo4 -Odemo4p2.ucc -DBase=7 -DVbase=1948
COMPILE demo4 -Odemo4p3.ucc -DBase=12 -DVbase=1848
COMPILE demo4 -Odemo4p4.ucc -DBase=17 -DVbase=1748
UCMLOAD 1 demop1 com1: -s22
UCMLOAD 2 demop2 com1: -s24
UCMLOAD 3 demop3 com1: -s26
UCMLOAD 4 demop4 com1: -s28
```

When using the commands listed the programmer can either rack address the UCM for 21 registers for minimum rack address space or 29 registers in order to see the UCM status registers for debugging purposes. A register map follows:

R[1] = UCM Program Control Register
R[2] = Command register for Unit 1 Port 1
R[3] = Command register for Unit 2 Port 1
R[4] = Unit response status Port 1
R[5] = Unit 1 communication error Port 1
R[6] = Unit 2 communication error Port 1
R[7] = Command register for Unit 1 Port 2
R[8] = Command register for Unit 2 Port 2
R[9] = Unit response status Port 2
R[10] = Unit 1 communication error Port 2
R[12] = Command register for Unit 1 Port 3
R[13] = Command register for Unit 2 Port 3
R[14] = Unit response status Port 3
R[15] = Unit 1 communication error Port 3
R[16] = Unit 2 communication error Port 3
R[17] = Command register for Unit 1 Port 4
R[18] = Command register for Unit 2 Port 4
R[19] = Unit response status Port 4
R[20] = Unit 1 communication error Port 4

R[11] = Unit 2 communication error Port 2 R[21] = Unit 2 communication error Port 4

The following registers can be read by the PLC if the rack address space of the UCM is large enough, 29 registers.

R[22] = Port 1 status register
R[23] = Port 1 line number register
R[24] = Port 2 status register
R[25] = Port 2 line number register
R[26] = Port 3 status register
R[27] = Port 3 line number register
R[28] = Port 4 status register
R[29] = Port 4 line number register

Appendix B
DEMO1.UCM

```
{*****************************************************************************

REGISTER      DESCRIPTION                WHO WRITES
    R[1]   program control register          PLC
    R[2]..R[9] program status registers      UCM
    R[10] command register for unit 1        PLC
    R[11] command register for unit 2        PLC
    R[12] Unit response status               UCM
          Bit 1 Good reply from unit 1
          Bit 2 Error reply from unit 1
          Bit 3 No reply from unit 1
          Bit 9 Good reply from unit 2
          Bit 10 Error reply from unit 2
          Bit 11 No reply from unit 2
    R[13] Unit 1 communication error         UCM
    R[14] Unit 2 communication error         UCM
Rack address 14 registers for the UCM

VARIABLES:

R[2048] retry counter
*****************************************************************************}

SET BAUD 9600 PARITY EVEN DATA 8 STOP 1 CAPITALIZE FALSE

Main_loop:
    ON CHANGE R[10] GOTO write_unit_1
    ON CHANGE R[11] GOTO write_unit_2
WAIT

write_unit_1:
R[2048] = 3 { R[2048] is retry counter }
retry_1:
    R[2048] = R[2048] - 1
    IF R[2048] < 0 THEN GOTO no_reply_1
    TRANSMIT "\02":"1":HEX(R[10],1):"\03":BYTE(LRC(1,4,0)):"\0D"
```

```
            ON RECEIVE "1":HEX((R[10]),1):"\06":BYTE((LRC(1,3,0))):"\0D" GOTO good_reply_1
            ON RECEIVE "1":HEX((R[10]),1):HEX(R[13],1):"\15":BYTE((LRC(1,4,0))):"\0D" GOTO
nack_1
    ON TIMEOUT 5 GOTO retry_1
WAIT

good_reply_1:
R[12] = 1
GOTO wait_for_clear

nack_1:
R[12] = 2
GOTO wait_for_clear

no_reply_1:
R[12] = 4
wait_for_clear:
    ON CHANGE R[10] GOTO clear_status {Wait for PLC to clear R[10]}
WAIT

clear_status:
R[12] = 0 {clear status register}
R[13] = 0 {clear Unit 1 error}
R[14] = 0 {clear Unit 2 error}
GOTO Main_loop

write_unit_2:
R[2048] = 3
retry_2:
    R[2048] = R[2048] - 1
    IF R[2048] < 0 THEN GOTO no_reply_2
    TRANSMIT "\02":"2":HEX(R[11],1):"\03":BYTE(LRC(1,4,0)):"\0D"
            ON RECEIVE "2":HEX((R[11]),1):"\06":BYTE((LRC(1,3,0))):"\0D" GOTO good_reply_2
            ON RECEIVE "2":HEX((R[11]),1):HEX(R[14],1):"\15":BYTE((LRC(1,4,0))):"\0D" GOTO
nack_2
            ON TIMEOUT 5 GOTO retry_2
    WAIT

good_reply_2:
R[12] = 256
GOTO wait_for_clear_2

nack_2:
R[12] = 512
GOTO wait_for_clear_2

no_reply_2:
R[12] = 1024
wait_for_clear_2:
    ON CHANGE R[11] GOTO clear_status {Wait for PLC to clear R[11]}
WAIT
```

Appendix C
DEMO2.UCM

```
{*****************************************************************************
REGISTER      DESCRIPTION                    WHO WRITES
R[1]     program control register                PLC
R[2]..R[9] program status registers              UCM
R[10] command register for unit 1                PLC
R[11] command register for unit 2 PLC
R[12] Unit response status                       UCM
    Bit 1 Good reply from unit 1
    Bit 2 Error reply from unit 1
    Bit 3 No reply from unit 1
    Bit 9 Good reply from unit 2
    Bit 10 Error reply from unit 2
    Bit 11 No reply from unit 2
R[13] Unit 1 communication error                 UCM
R[14] Unit 2 communication error                 UCM
Rack address 14 registers for the UCM

VARIABLES:
R[2048] retry counter
R[2047] Unit number
R[2046] Command register
R[2045] Error register
Note that the variables are not rack addressed.
*****************************************************************************}

SET BAUD 9600 PARITY EVEN DATA 8 STOP 1 CAPITALIZE FALSE

Main_loop:
ON CHANGE R[10] GOTO write_unit_1
ON CHANGE R[11] GOTO write_unit_2
WAIT

write_unit_1:
R[2047] = 1
R[2046] = 10
R[2045] = 13
```

```
GOTO write

write_unit_2:
R[2047] = 2
R[2046] = 11
R[2045] = 14

write:
R[2048] = 3 { R[2048] is retry counter }
retry_loop:
R[2048] = R[2048] - 1
IF R[2048] < 0 THEN GOTO no_reply
TRANSMIT "\02":HEX(R[2047],1):HEX(R[R[2046]],1):"\03":BYTE(LRC(1,4,0)):"\0D"
    ON RECEIVE HEX((R[2047]),1):HEX((R[R[2046]]),1):"\06":BYTE((LRC(1,3,0))):"\0D" GOTO
good_reply
    ON RECEIVE
HEX((R[2047]),1):HEX((R[R[2046]]),1):HEX(R[R[2045]],1):"\15":BYTE((LRC(1,4,0))):"\0D"
        GOTO nack
    ON TIMEOUT 5 GOTO retry_loop
WAIT

good_reply:
R[12] = 1
IF R[2047] = 2 then R[12] = 256
GOTO wait_for_clear

nack:
R[12] = 2
IF R[2047] = 2 then R[12] = 512
GOTO wait_for_clear

no_reply:
R[12] = 4
IF R[2047] = 2 then R[12] = 1024
wait_for_clear:
ON CHANGE R[R[2046]] GOTO clear_status {Wait for PLC to clear command register}
WAIT

clear_status:
R[12] = 0 {clear status register}
R[13] = 0 {clear Unit 1 error}
R[14] = 0 {clear Unit 2 error}
GOTO Main_loop
```

Appendix D
DEMO3.UCM

```
{*****************************************************************************
REGISTER      DESCRIPTION                    WHO WRITES
   R[1]    program control register             PLC
   R[2]..R[9] program status registers          UCM
   R[10] command register for unit 1            PLC
R[11] command register for unit 2               PLC
R[12] Unit response status                      UCM
   Bit 1 Good reply from unit 1
   Bit 2 Error reply from unit 1
   Bit 3 No reply from unit 1
   Bit 9 Good reply from unit 2
   Bit 10 Error reply from unit 2
   Bit 11 No reply from unit 2
R[13] Unit 1 communication error                UCM
R[14] Unit 2 communication error                UCM
Rack address 14 registers for the UCM

VARIABLES:
R[2048] retry counter
R[2047] Unit number
R[2046] Command register
R[2045] Error register
Note that the variables are not rack addressed.
*****************************************************************************}

SET BAUD 9600 PARITY EVEN DATA 8 STOP 1 CAPITALIZE FALSE

DEFINE Command_1 = R[10]
DEFINE Command_2 = R[11]
DEFINE Status = R[12]
DEFINE Error_1 = R[13]
DEFINE Error_2 = R[14]
DEFINE Retry = R[2048]
DEFINE Unit = R[2047]
DEFINE Command = R[2046]
DEFINE Error = R[2045]
```

```
DEFINE STX = "\02"
DEFINE EOT = "\03"
DEFINE ACK = "\06"
DEFINE NAK = "\15"
DEFINE CR = "\0D"

Main_loop:
ON CHANGE Command_1 GOTO write_unit_1
ON CHANGE Command_2 GOTO write_unit_2
WAIT

write_unit_1:
Unit = 1
Command = 10
Error = 13
GOTO write

write_unit_2:
Unit = 2
Command = 11
Error = 14

write:
Retry = 3
retry_loop:
Retry = Retry - 1
IF Retry < 0 THEN GOTO no_reply
TRANSMIT STX:HEX(Unit,1):HEX(R[Command],1):EOT:BYTE(LRC(1,4,0)):CR
   ON RECEIVE HEX((Unit),1):HEX((R[Command]),1):ACK:BYTE((LRC(1,3,0))):CR GOTO
good_reply
   ON RECEIVE
HEX((Unit),1):HEX((R[Command]),1):HEX(R[Error],1):NAK:BYTE((LRC(1,4,0))):CR
         GOTO nack
   ON TIMEOUT 5 GOTO retry_loop
WAIT

good_reply:
Status = 1
IF Unit = 2 THEN Status = 256
GOTO wait_for_clear

nack:
Status = 2
IF Unit = 2 THEN Status = 512
GOTO wait_for_clear

no_reply:
Status = 4
IF Unit = 2 THEN Status = 1024
wait_for_clear:
ON CHANGE R[Command] GOTO clear_status {Wait for PLC to clear command register}
WAIT

clear_status:
Status = 0 {clear status register}
Error_1 = 0 {clear Unit 1 error}
Error_2 = 0 {clear Unit 2 error}
```

```
GOTO Main_loop
```

Appendix E
DEMO4.UCM

```
{*****************************************************************************
REGISTER        DESCRIPTION                     WHO WRITES
R[1]        program control register                PLC
R[2]..R[9] program status registers                  UCM
R[10] command register for unit 1                    PLC
R[11] command register for unit 2                    PLC
R[12] Unit response status                           UCM
   Bit 1 Good reply from unit 1
   Bit 2 Error reply from unit 1
   Bit 3 No reply from unit 1
   Bit 9 Good reply from unit 2
   Bit 10 Error reply from unit 2
   Bit 11 No reply from unit 2
R[13] Unit 1 communication error                     UCM
R[14] Unit 2 communication error                     UCM
Rack address 14 registers for the UCM

VARIABLES:
R[2048] retry counter
R[2047] Unit number
R[2046] Command register
R[2045] Error register
Note that the variables are not rack addressed.
*****************************************************************************
*************}

SET BAUD 9600 PARITY EVEN DATA 8 STOP 1 CAPITALIZE FALSE

DEFINE Command_1 = R[Base]
DEFINE Command_2 = R[Base+1]
DEFINE Status = R[Base+2]
DEFINE Error_1 = R[Base+3]
DEFINE Error_2 = R[Base+4]
DEFINE Retry = R[Vbase]
DEFINE Unit = R[Vbase-1]
DEFINE Command = R[Vbase-2]
```

```
DEFINE Error = R[Vbase-3]
DEFINE STX = "\02"
DEFINE EOT = "\03"
DEFINE ACK = "\06"
DEFINE NAK = "\15"
DEFINE CR = "\0D"

Main_loop:
ON CHANGE Command_1 GOTO write_unit_1
ON CHANGE Command_2 GOTO write_unit_2
WAIT

write_unit_1:
Unit = 1
Command = Base
Error = Base+3
GOTO write

write_unit_2:
Unit = 2
Command = Base+1
Error = Base+4

write:
Retry = 3
retry_loop:
Retry = Retry - 1
IF Retry < 0 THEN GOTO no_reply
TRANSMIT STX:HEX(Unit,1):HEX(R[Command],1):EOT:BYTE(LRC(1,4,0)):CR
   ON RECEIVE HEX((Unit),1):HEX((R[Command]),1):ACK:BYTE((LRC(1,3,0))):CR GOTO
good_reply
   ON RECEIVE
HEX((Unit),1):HEX((R[Command]),1):HEX(R[Error],1):NAK:BYTE((LRC(1,4,0))):CR
        GOTO nack
   ON TIMEOUT 5 GOTO retry_loop
WAIT

good_reply:
Status = 1
IF Unit = 2 THEN Status = 256
GOTO wait_for_clear

nack:
Status = 2
IF Unit = 2 THEN Status = 512
GOTO wait_for_clear

no_reply:
Status = 4
IF Unit = 2 THEN Status = 1024
wait_for_clear:
ON CHANGE R[Command] GOTO clear_status {Wait for PLC to clear command register}
WAIT

clear_status:
Status = 0 {clear status register}
Error_1 = 0 {clear Unit 1 error}
```

```
Error_2 = 0 {clear Unit 2 error}
GOTO Main_loop
```

Appendix F
Serial Communication Overview

This Appendix is meant to provide a general background for common types of asynchronous serial communication.

Hardware Overview

The need for information to be exchanged between independent devices has brought about the development of several serial communication standards. The most commonly encountered are RS-232, RS-422, RS-485, and 20mA current loop.

RS-232

RS-232 is intended for connecting two devices together for serial communication for short distances (50 feet or less) and low baud rates (19200 baud or less).

RS-232-C has two main classes of devices: DTE (Data Terminal Equipment) such as terminals and personal computers, and DCE (Data Communication Equipment) such as modems.

The original implementation of the RS232 connection was for connecting terminals to modems as shown in Figure F-1 The standard connectors were mounted on the equipment were DB25 females. Straight through cables with 25 pin male connectors were used to connect the DTE to DCE.

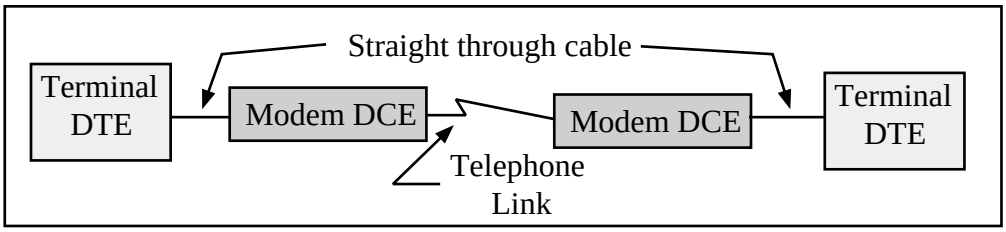


Figure F-1 DTE to Modem connection

Each pin on the DTE was connected to the same pin on the DCE. The most common pins and their definitions are listed below.

Pin 2: **TD** Transmit Data

This circuit is the path that serial data is sent from the DTE to the DCE.

Pin 3: **RD** Receive Data

This circuit is the path that serial data is sent from the DCE to the DTE.

Pin 4: **RTS** Request to Send

This circuit is the signal that indicates that the DTE wishes to send data to the DCE. In normal operation the RTS line will be OFF (MARK). Once the DTE has data to send it asserts RTS (SPACE) and waits for the DCE to assert CTS. RTS will remain asserted until the data is completely sent. In a full duplex channel, RTS may be asserted at initialization and left in that state.

Pin 5: **CTS** Clear to Send

This circuit is the signal that indicates that the DCE is ready to receive data from the DTE. In normal operation the CTS is not asserted. When the DTE asserts RTS, the DCE will do whatever is necessary to allow data to be sent. (This may mean raising the carrier and waiting until it is stabilized.) When the DCE is ready, it asserts CTS which allows the DTE to send data. When the DTE is finished sending data it will reset the RTS and the DCE will in turn reset its CTS.

Note: Most DTE must have CTS asserted before it will transmit.

Pin 6: **DSR** Data Set Ready

This circuit is the signal that informs the DTE that the DCE is active. It is normally asserted by the DCE at power-up and left that way.

Note: Most DTE must have DSR asserted to operate properly.

Pin 7: **SG** Signal Ground

This circuit is the ground to which all signals are referenced.

Pin 8: **DCD** Data Carrier Detect

This circuit is the signal that the DCE informs the DTE that it has an incoming carrier.

Note: Some DTE must have DCD asserted to operate properly. Also, some personal computer modems always assert DCD.

Pin 20: **DTR** Data Terminal Ready

This circuit provides the signal that informs the DCE that the DTE is alive and well. It is normally asserted by the DTE at power-up and left in that state.

Note: Most DCE must have DTE asserted to operate properly.

Pin 22: **RI** Ring Indicator

This circuit provides the signal from the DCE to indicate that the modem is ringing. The line is asserted by the DCE during each ring cycle.

The full pinout for the standard 25 pin connector is shown in Figure F-1.

With the down-sizing of computers it became necessary to move to a 9 pin port to save room. Only the most commonly used functions were kept for the 9 pin configu-

ration. The TYPE A and TYPE B configurations are shown in Figures F-2 and F-3. The only difference is pins 2 and 3.

Table F-1 25 pin RS-232 port

Pin	Name	DTE/DCE	Function
1	CG	<--->	Frame Ground
2	TD	--->	Transmitted Data
3	RD	<---	Received Data
4	RTS	--->	Request to Send
5	CTS	<---	Clear to Send
6	DSR	<---	Data Set Ready
7	SG	<--->	Signal Ground
8	DCD	<---	Data Carrier Detect
9*		<---	Positive DC Test Voltage
10*		<---	Negative DC test Voltage
11*	QM	<---	Equalizer Mode
12+	SDCD	<---	Secondary Data Carrier Detect
13+	SCTS	<---	Secondary Clear to Send
14+	STD	--->	Secondary Transmitted Data
15#	TC	<---	Transmit Clock
16+	SRD	<---	Secondary Receive Data
17#	RC	<---	Receive Clock
18	DCR	<---	Divided Clock Receiver
19+	SRTS	--->	Secondary Request to Send
20	DTR	--->	Data Terminal Ready
21*	SQ	<---	Signal Quality Detect
22	RI	<---	Ring Indicator
23*		<---	Data Rate Selector
24*	SCTE	--->	Data Rate Selector
25*		--->	Busy

In the above table, the character following the pin number means:

* rarely used
+ used only if secondary channel implemented
# used only in synchronous interfaces

Although originally all DB25 RS-232 ports were female, most personal computers which have a DB25 RS-232 connector use a male connector. The female DB25 connector on a personal computer is most likely the parallel printer port and should never be connected to any RS-232 device.

As is indicated above, the 25 RS-232 standard has the option of 2 data channels, each with their own handshake lines, and the option of synchronous link. These functions are rarely used and have been left off of the newer 9 pin ports.

Table F-2 Type A 9 pin RS-232 port

Pin	Name	Function
1	DCD	Data Carrier Detect
2	TD	Transmitted Data
3	RD	Received Data
4	DTR	Data Terminal Ready
5	SG	Signal Ground
6	DSR	Data Set Ready
7	RTS	Request to Send
8	CTS	Clear to Send
9	RI	Ring Indicator

Table F-3 Type B 9 pin RS-232 port

Pin	Name	Function
1	DCD	Data Carrier Detect
2	RD	Received Data
3	TD	Transmitted Data
4	DTR	Data Terminal Ready
5	SG	Signal Ground
6	DSR	Data Set Ready
7	RTS	Request to Send
8	CTS	Clear to Send
9	RI	Ring Indicator

The TYPE B connection is the most common on 9 pin personal computer ports. These ports are usually male connectors.

Determining the type of RS-232 port with a voltmeter.

It is possible to determine the type of port with the use of a voltmeter using the following procedure:

1 Set the voltmeter for DC volts, 30 volt range. The voltage being read likely be negative and be within the range of +3VDC to -15VDC.

2 Power up the equipment.

3 Place the negative probe (black) of the voltmeter on the SG pin of the port. (Pin 7 of a DB25 port or pin 5 of a DE9 port)

4 Place the positive probe (red) of the voltmeter on pin 2.

5 Write down that voltage.

6 Place the positive probe (red) of the voltmeter on pin 3.

7 Write down that voltage.

The TX voltage is within the range of -15V to -5V.
The RX voltage is within the range of -3V to +3V.

Therefore, if pin 2 is the more negative voltage of the two, the serial port is **TYPE A**. If pin 3 is the more negative voltage, the serial port is **TYPE B**.

Since almost every device that is not a modem is a DTE, it is quite common to connect DTE to DTE without a modem pair. A "null modem" connection has been established to simplify this situation. It simply reverses the transmit and receive connections from one side of the connector to another and jumpers some of the hardware handshake lines.

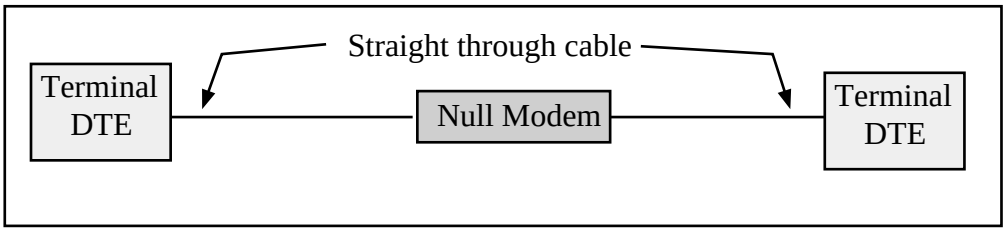


Figure F-2 Null Modem connection

The null modem is frequently a small enclosure with one male connector and one female connector. Sometimes the null modem is built into a single cable to connect the DTE device to another DTE device. In addition to the crossing of the transmit and receive pins, some additional connections to the hardware handshake pins is usually made the pinouts of null modems for 25 pin and 9 pin connections are shown below.

Table F-4 DB25 Null Modem

Male Connector	Female Connector
1	1
2	3
3	2
4	5
5	4
6	20
8	6
20	8
7	7

Table F-5 DE9 Null Modem

Male Connector		Female Connector
2	———	3
3	———	2
5	———	5
7	———	8
8	———	7
1	———	4
6	———	1
4	———	6

Electrical characteristics of RS-232

The RS-232 interface is an Single-Ended driver with an open ended receiver. The driver asserts a voltage between -5V and -15V relative to the Signal Ground to represent the MARK state (Logic TRUE). The driver asserts a voltage between +5V and +15V relative to the Signal Ground to represent a SPACE state (Logic FALSE). The fact that there may be a 30 volt swing between MARK and SPACE conditions may lead to problems with the slew rate of the signal due to the capacitance of the cable. If the cable run is long and communication problems are occurring, try lowering the baud rate.

RS-422

The RS-422 interface uses a closed ended driver and a closed ended receiver. The RS-232 interface is ground referenced. This can cause trouble in situations where a common mode induced noise may severely affect the signal by changing the reference. A better solution for noise immunity is to convert the ground referenced data at the transmission end into a differential signal and transmit this down a balanced, twisted-pair line. At the receiving end any induced noise voltage will appear equally on each line. If the receiver only looks at the differential signal, any induced common mode voltage will be rejected. This is the idea behind the RS-422 and RS-485 standards.

The RS-422 interface typically used in a point to point connection using a pair of wires from the transmitter of Unit 1 to the receiver of the Unit 2. Another pair of wires is from the transmitter of Unit 2 to the receiver of Unit 1. This connection allows for full duplex operation, i.e. the units can transmit messages while they are receiving messages. The Square D SY/MAX family of PLCs us RS-422 communication. Their standard pinout is as follows:

Table F-6 SY/MAX DE9S RS-422 Pinout

Pin	Function	Description
1	TX -	Transmit Data from device. (Data OUT) (inverted)
2	TX+	Transmit Data from device. (Data OUT) (non-inverted)
3	RX-	Receive Data (Data IN) (inverted)
4	RX+	Receive Data (Data IN) (non-inverted)
5	CTS-	Clear to Send (inverted)
6	RTS-	Request to Send (inverted)
7	CTS+	Clear to Send (non-inverted)
8	RTS+	Request to Send (non-inverted)
9	Shield	Shield Ground. AC coupled to chassis.

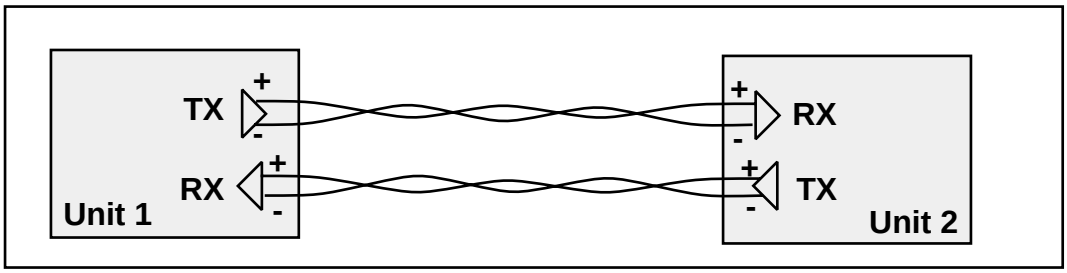


Figure F-3 RS-422 Setup

Like RS-485, the RS-422 protocol may be used in a multidrop configuration as shown in Figure F-4. The RS-485 standard requires a minimum capability of 32 receivers on the network. The RS-422 standard places no minimum requirement and therefore is typically used in point to point or as the host of a RS-485 network.

Always connect the + terminals on the TX to the + terminals on the RX. Similarly, connect the - terminals on the TX to the - terminals on the RX.

Note: Occasionally the manufacture incorrectly labels the polarity of the connections. If the system is not working try exchanging the polarity of the TX pair and the RX pair on the host.

With the high noise immunity and low voltage swings, the RS-422 interface may have long runs of up to 10,000 feet.

Electrical Characteristics of RS-422.

The driver asserts a negative voltage across the receiver to represent a MARK state, a positive voltage to represent the SPACE state. The receiver triggers off of the transition through the zero voltage point.

RS-485 (four wire)

The RS-485 interface is like the RS-422 interface with the exception that the transmitters are able to tri-state, i.e. float. This allows up to 32 transmitters to be connected to a host receiver and multiple receivers to be connected to a host transmitter. This tech-

nique is called multidropping and is shown in Figure F-4. Square D PowerLogic Circuit monitors use this type of RS-485 for communication.

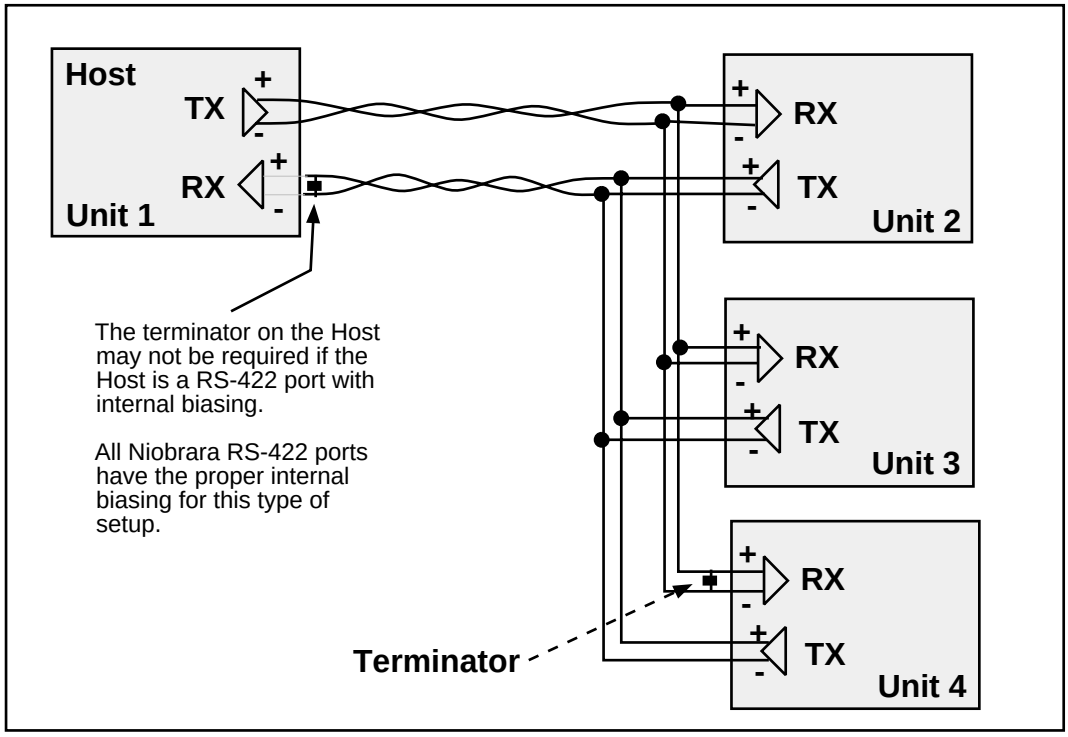


Figure F-4 RS-485 Four Wire Setup

As shown in Figure F-4, it is possible to use a 4 wire RS-422 port to drive a 4 wire RS-485 multidrop network. If the RS-422 port does not have internal biasing on the RX pair it may be necessary to add a terminator to that end of the network. The terminator is a resistor correctly matched to the line which reduces reflections on the network.

RS-485 (two wire)

Another version of the differential communication system is the RS-485 two wire network. The two wire system is a half-duplex connection where each unit transmits and receives on the same pair of wires. Only one transaction may occur on the network at one time as opposed to the four wire system where the units may transmit while they are receiving. The two wire system is inexpensive to install because only one twisted pair cable is needed. All the + terminals are connected together, all the - terminals are connected together. A terminating resistor is usually required on each end.
Since each unit can listen to the transmissions of every other unit on the network, peer to peer communication is available. The trade off is that the half-duplex connection has half of the throughput of the full duplex four wire system. A typical installation is shown in Figure F-5.

The Allen-Bradley® Data Highway is an example of a two wire RS-485 multidrop network.

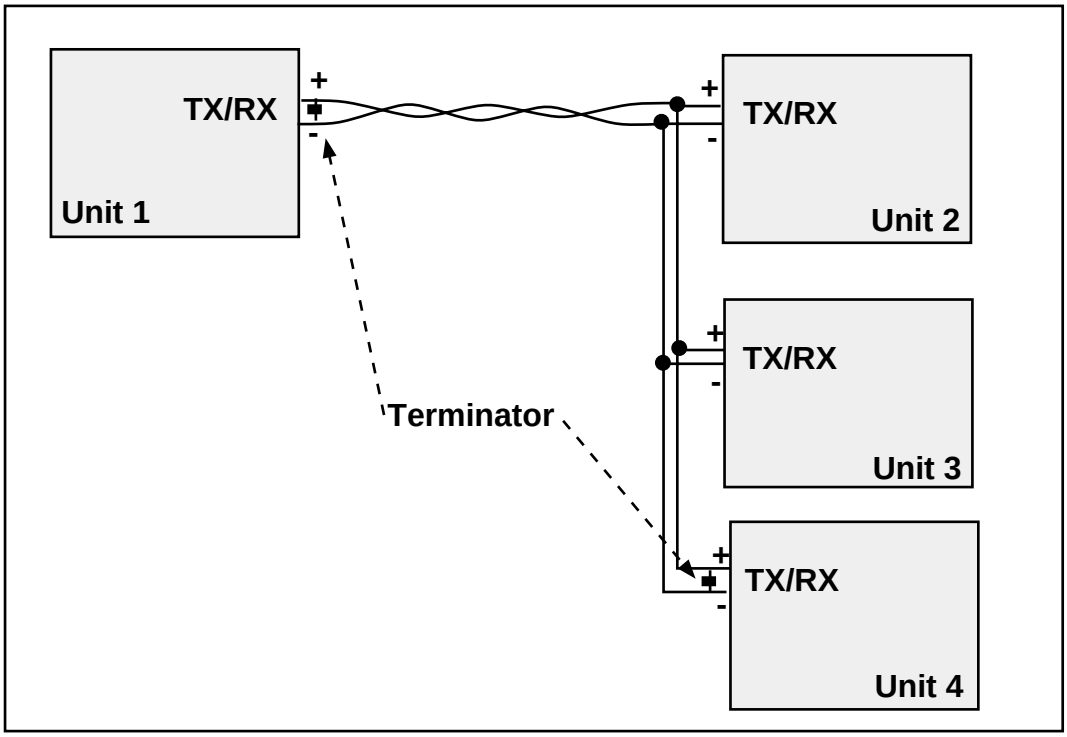


Figure F-5 RS-485 Two wire Multidrop Setup

Note: Occasionally the manufacture incorrectly labels the polarity of the connections. If the system is not working try exchanging the polarity of the TX/RX pair on the unit.

20mA Current Loop

The 20mA Current Loop is another multidrop configuration. The transmitting mechanism may be explained as simply opening a normally closed switch for the data bit transmission. The receiver is usually a optical isolator (LED/phototransistor) unit. Each loop is powered by a Constant Current Source. The source may be part of one of the units or may be a separate device as shown in Figure F-6.

As in other multidrop schemes, each unit watches the RX line for its messages with its address. The number of units on the network is dependent upon the addressing scheme as well as the voltage supply of the constant current source. Each RX receiver on the circuit will cause a voltage drop in the supply current. The series sum of these drops must be a value less than the available voltage from the current supply or none of the receivers will work.

To check, add up the voltage drops around the loop and make sure that the sum is less than the compliance voltage of the current source.

Some Red Lion Controls equipment such as the Apollo message centers use a 20mA current loop for serial communication.

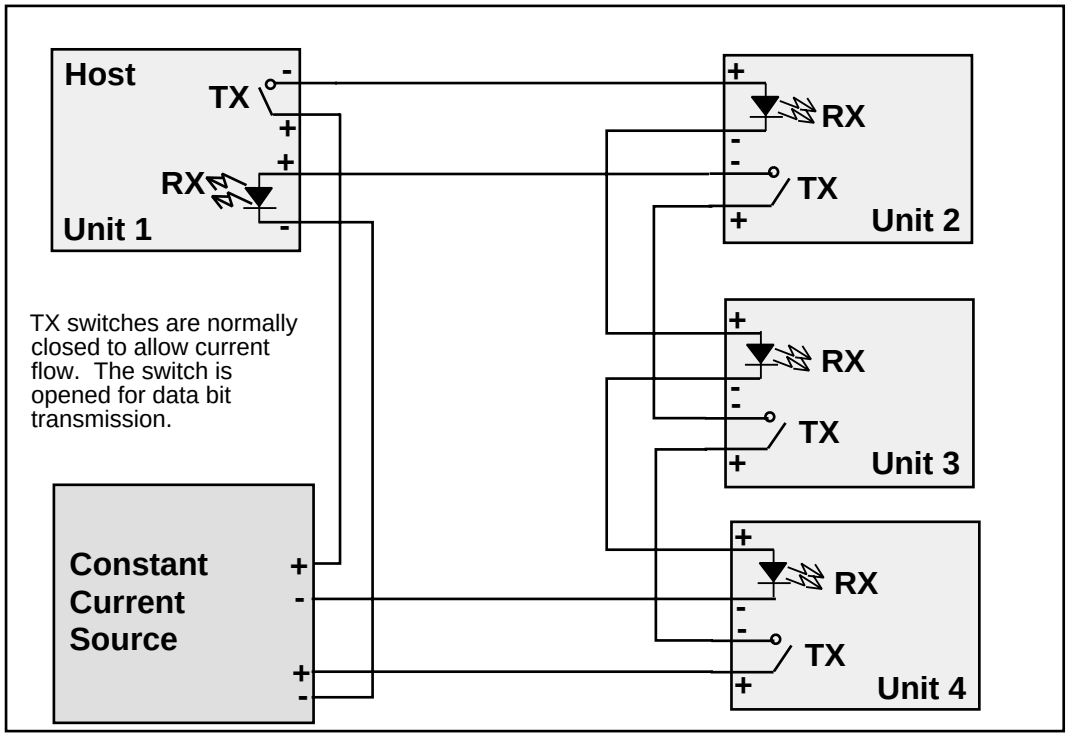


Figure F-6 20mA Current Loop (Full Duplex)

A half-duplex 20mA current loop may be formed by connecting the a single current supply through all RX and TX connections serially.

Hardware Handshaking

Sometimes it is necessary for the transmitting and receiving devices to signal each other to control the flow of data. This may be done with special characters in the software (see Software Handshaking on page 137.) or with physical connections to the hardware. This physical signaling system is known as hardware handshaking. The electrical characteristics of the handshaking lines are the same as the TX/RX lines of the system. It may be a single-ended, ground referenced signal in RS-232, or a differential pair in RS-422 or RS-485. The two most common types of hardware handshaking are: Request to Send (RTS) and Clear to Send (CTS).

Request to Send (RTS) is employed in two different modes: Request to Send (also known as Push to Talk) and Restraint.

Request to Send (Push to Talk) mode is used by the transmitting device to signal the receiving device that it has data to send. This is typically used in modem and radio modem setups. When the transmitting device has data to send, it asserts its RTS. When the modem is ready, it asserts the CTS of the transmitting device to allow it to transmit. The transmitting device will keep RTS asserted until it has sent all of its data.

Push to talk mode is the original mode of the RS-232 standard for connecting DTE to DCE. This mode of RTS usage is not commonly used in most serial communication situations and is usually restricted to modems and radios.

Restraint mode is used by the receiving device to signal the transmitting device

that it is sending data too quickly for to be processed. When this situation occurs, the receiving device negates its RTS to signal the transmitting device to stop. When the receiving device has "caught up" it asserts its RTS line to signal the transmitting device to continue.

Restraint mode is the most common handshaking mode.

Sometimes if two RS-232 DTE devices are connected together, the DTR pin may be used as the restraint handshake line instead of RTS.

Clear to Send (CTS) is used to signal the transmitting device that the receiving device is ready to accept data. The transmitting device will not send data until its CTS is asserted.

In a typical hardware handshaking application RTS on the one device is usually connected to CTS on the other.

The vast majority of cases do not require active hardware handshaking. In this case it is common to jumper the RTS of each unit to its own CTS. This will allow each port to transmit at will.

Software Overview

Binary Representation of Data

With the implementation of an appropriate hardware system, a uniform system must be implemented to allow the data to be transferred.

In a synchronous serial system, each bit of data is transferred with a clock signal. This ensures that the receiving device "knows" when each bit is transferred so it can reconstruct the data. The throughput is determined by the number of data bits and the clock frequency. The data synchronization is accomplished by the transmission of a sync character which is an out-of-band character.

The more common type of serial communication is asynchronous. In an asynchronous system the character timing is determined by a local clock at each end. This clock usually runs at 16 times the baud rate. Each bit is to stay in its MARK or SPACE state for the time determined by the baud rate. Data is transmitted in packets of the following form: Start bit; Data Bits, Parity Bit, Stop Bits.

Start Bit

The Start bit is always a SPACE bit and signifies the beginning of the data packet.

Data Bits

Typically 7 or 8 bits. On rare occasions may be 5 or 6 bits. The if the data is logic true it is in the MARK state. If the data is false it is in the SPACE state. The data is transmitted LSB first.

Parity Bit

The parity bit is a bit that follows the last data bit. Its determination is based upon the type of parity selected. The most common types are ODD, EVEN, NONE, MARK, and SPACE. These parity selections are calculated as follows.

Odd: The parity bit is set to the MARK state if the number of data bits in the MARK state are even. Therefore the total number of data bits in the MARK state plus the parity bit is an **ODD** number.

Even: The parity bit is set to the MARK state if the number of data bits in the MARK state are odd. Therefore the total number of data bits in the MARK state plus the parity bit is an **EVEN** number.

None: No parity bit. The stop bit comes immediately after the last data bit.

Mark: Always logic high. Like an additional stop bit. The mark parity bit is usually used in conjunction with 7 bit ASCII data.

Space: Always logic low. The space parity bit is usually used in conjunction with 7 bit ASCII data.

Stop Bit

The Stop bit is always a MARK bit. There is always one stop bit, sometimes two stop bits, and on rare occasions 1.5 stop bits. The receiving device starts timing for the next start bit half way through the last stop bit in the message. A message with two stop bits is the same as a message with one stop bit plus one extra character time worth of dead space between packets. Therefore a device set for 1 stop bit can also receive a message with 2 stop bits.

It is worth noting that the total number of bits in a packet is sometimes used to reference the type of communication. For instance, Square D SY/MAX packets have:

1 start bit + 8 data bits + 1 parity bit + 1 stop bit = 11 bits.

Most Hayes modems can only handle a 10 bit protocol. Now this may be:

1 start bit + 8 data + 0 parity + 1 stop = 10 bits
or 1 start bit + 7 data + 1 parity + 1 stop = 10 bits
or 1 start bit + 7 data + 0 parity + 2 stop = 10 bits

Since the Hayes modem is designed to handle only 10 bits per character, this explains why it is not possible to send an 11 bit protocol like SY/MAX across these modems. The bit count gets off when the 10th bit arrives at the modem and the modem expects a start bit instead of the stop bit. An 11 bit modem is required for this type of communication.

Message Determination

With the above descriptions of an asynchronous serial packet, binary data may be sent from one device to another.

Hexadecimal numbers

Binary representation is somewhat cumbersome to deal with so hexadecimal numbers are often used. Hexadecimal (hex) numbers are base sixteen numbers. There are 16 digits in hex, the ten decimal digits 0 through 9 plus the six letters A through F which represent the decimal numbers 10 through 15. The following is a table of the decimal numbers 0 through 31, their hexadecimal equivalent and their binary equivalent.

Table F-7 Decimal, Hex, Binary

Dec	Hex	Binary		Dec	Hex	Binary
0	0	0000		16	10	0001 0000
1	1	0001		17	11	0001 0001
2	2	0010		18	12	0001 0010
3	3	0011		19	13	0001 0011
4	4	0100		20	14	0001 0100
5	5	0101		21	15	0001 0101
6	6	0110		22	16	0001 0110
7	7	0111		23	17	0001 0111
8	8	1000		24	18	0001 1000
9	9	1001		25	19	0001 1001
10	A	1010		26	1A	0001 1010
11	B	1011		27	1B	0001 1011
12	C	1100		28	1C	0001 1100
13	D	1101		29	1D	0001 1101
14	E	1110		30	1E	0001 1110
15	F	1111		31	1F	0001 1111

Notice that it takes four binary digits makes one hex digit. Conversion from binary to hex is straight forward and explains why hex is so popular in PLCs. Since the most PLC devices use 16-bit registers it takes 4 hex digits to represent one register i.e. 1A2F. The first two characters "1A" make up the most significant byte (8 bits) and the last two characters make up the least significant byte.

ASCII characters

ASCII is a set of 7-bit characters used in communication, computers and programmable logic controllers. The word ASCII is a acronym for American Standard Code for Information Interchange. ASCII is a way to interpret 7 bits (or 8 bits with a leading 0) as alphanumeric characters. There is an ASCII table following this section for your reference.

In ASCII. the small letter "p" is represented by the binary number 0111 0000. This is the hexadecimal number 70 or the decimal number 112.

Similarly, the capital letter "E" is represented by the binary number 0100 0101 which is the hexadecimal number 45 or the decimal number 69.

Software Handshaking

It is common for communicating devices to need to exert control over each other. Typically the receiving device will need to force the transmitting device to stop and wait for it to catch up. This may be done at the hardware level (See Hardware Handshaking on page 134.) or at the software level using special characters. A common way for devices communicating with ASCII is to use the X-ON/X-OFF characters.

X-ON

The X-ON character is the ASCII character DC1, which has the decimal value 17, hex value 11, and may be generated by pressing Ctrl+Q.

When a device receives an X-ON it starts sending data from the position at which it received an X-OFF character. The X-ON is only acted upon if the device is first halted with an X-OFF.

X-OFF

The X-OFF character is the ASCII character DC3, which has the decimal value 19, the hex value 13, and the may be generated by pressing Ctrl+S.

When a device receives an X-OFF it stops sending data. It will remain in this state until an X-ON character is received. At this time, it will resume sending the data.

Obviously software handshaking is only useful when devices are sending characters for the handshake which are out of the normal range of data. Sometimes certain protocols get around this by specifying special escape sequences that the receiving device will recognize. This is done by methods such as sending that special character once if is to be recognized as a control character and twice if it is to be included as data.

Table F-8 ASCII Table

Hex	Dec	Character	Description	Abrv		Hex	Dec	Char.		Hex	Dec	Char.		Hex	Dec	Char.
00	0	[CTRL]@	Null	NUL		20	32	SP		40	64	@		60	96	‘
01	1	[CTRL]a	Start Heading	SOH		21	33	!		41	65	A		61	97	a
02	2	[CTRL]b	Start of Text	STX		22	34	"		42	66	B		62	98	b
03	3	[CTRL]c	End Text	ETX		23	35	#		43	67	C		63	99	c
04	4	[CTRL]d	End Transmit	EOT		24	36	$		44	68	D		64	100	d
05	5	[CTRL]e	Enquiry	ENQ		25	37	%		45	69	E		65	101	e
06	6	[CTRL]f	Acknowledge	ACK		26	38	&		46	70	F		66	102	f
07	7	[CTRL]g	Beep	BEL		27	39	’		47	71	G		67	103	g
08	8	[CTRL]h	Back space	BS		28	40	(		48	72	H		68	104	h
09	9	[CTRL]i	Horizontal Tab	HT		29	41	)		49	73	I		69	105	i
0A	10	[CTRL]j	Line Feed	LF		2A	42	*		4A	74	J		6A	106	j
0B	11	[CTRL]k	Vertical Tab	VT		2B	43	+		4B	75	K		6B	107	k
0C	12	[CTRL]l	Form Feed	FF		2C	44	,		4C	76	L		6C	108	l
0D	13	[CTRL]m	Carriage Return	CR		2D	45	-		4D	77	M		6D	109	m
0E	14	[CTRL]n	Shift Out	SO		2E	46	.		4E	78	N		6E	110	n
0F	15	[CTRL]o	Shift In	SI		2F	47	/		4F	79	O		6F	111	o
10	16	[CTRL]p	Device Link Esc	DLE		30	48	0		50	80	P		70	112	p
11	17	[CTRL]q	Dev Cont 1 X-ON	DC1		31	49	1		51	81	Q		71	113	q
12	18	[CTRL]r	Device Control 2	DC2		32	50	2		52	82	R		72	114	r
13	19	[CTRL]s	Dev Cont 3 X-OFF	DC3		33	51	3		53	83	S		73	115	s
14	20	[CTRL]t	Device Control 4	DC4		34	52	4		54	84	T		74	116	t
15	21	[CTRL]u	Negative Ack	NAK		35	53	5		55	85	U		75	117	u
16	22	[CTRL]v	Synchronous Idle	SYN		36	54	6		56	86	V		76	118	v
17	23	[CTRL]w	End Trans Block	ETB		37	55	7		57	87	W		77	119	w
18	24	[CTRL]x	Cancel	CAN		38	56	8		58	88	X		78	120	x
19	25	[CTRL]y	End Medium	EM		39	57	9		59	89	Y		79	121	y
1A	26	[CTRL]z	Substitute	SUB		3A	58	:		5A	90	Z		7A	122	z
1B	27	[CTRL][	Escape	ESC		3B	59	;		5B	91	[		7B	123	{
1C	28	[CTRL]\	Cursor Right	FS		3C	60	<		5C	92	\		7C	124	|
1D	29	[CTRL]]	Cursor Left	GS		3D	61	=		5D	93	]		7D	125	}
1E	30	[CTRL]^	Cursor Up	RS		3E	62	>		5E	94	^		7E	126	~
1F	31	[CTRL]_	Cursor Down	US		3F	63	?		5F	95	_		7F	127	DEL

Appendix G
UCM Language Syntax

STATEMENTS

ON RECEIVE <message description> GOTO <label>

ON RECEIVE <message description> RETURN

ON CHANGE R[<expr>] GOTO <label>

ON CHANGE R[<expr>] RETURN

ON CHANGE R[<expr>] & <expr> GOTO <label>

ON CHANGE R[<expr>] & <expr> RETURN

ON TIMEOUT <expr> GOTO <label>

ON TIMEOUT <expr> RETURN

WAIT

GOTO <label>

GOSUB <label>

RETURN

IF <logical> THEN one or more statements followed by a *newline*

IF <logical> THEN one or more statements ELSE one or more statements, *newline*

IF <logical> THEN *newline*
one or more statements
ENDIF

IF <logical> THEN *newline*
one or more statements
ELSE
one or more statements
ENDIF

WHILE <logical> one or more statements WEND

REPEAT one or more statements UNTIL <logical>

FOR R[<expr>] = <expr> TO <expr>
one or more statements
NEXT

FOR R[<expr>] = <expr> TO <expr> STEP <expr>
one or more statements
NEXT

FOR R[<expr>] = <expr> DOWNTO <expr>
one or more statements
NEXT

FOR R[<expr>] = <expr> DOWNTO <expr> STEP <expr>
one or more statements
NEXT

R[<expr>] = <expr>

R[<expr>].<const> = <logical>

DELAY <expr>

STOP

TRANSLATE <const> : <string> = <string>

TRANSMIT <message description>

SET R[<expr>].<const>

READ <port> (<route>,...) <local>, <remote>, <count>
WRITE <port> (<route>,...) <local>, <remote>, <count>
PRINT <port> (<route>,...) <message description>

READ <port> R[<expr>] <local>, <remote>, <count>
WRITE <port> R[<expr>] <local>, <remote>, <count>
PRINT <port> R[<expr>] <message description>

CLEAR R[<expr>].<const>

TOGGLE R[<expr>].<const>

SET BAUD <const>
SET CAPITALIZE <const>
SET DATA <const>
SET DEBUG <const>
SET DUPLEX <const>
SET MODE <const>
SET MULTIDROP <const>
SET PARITY <const>
SET STOP <const>

DEFINE <macro>=<replacement string> *newline*

CONSTANTS <const> in descriptions above

decimal numbers 12345
signed numbers -123
hexadecimal constant x12ab
reserved constants:
    EVEN
    ODD
    NONE
boolean constants:
    TRUE
    FALSE

EXPRESSIONS <NUMERIC expr> above

Operators:

- unary negation
~ unary bitwise complement
* multiplication
/ division
% modulus
+ addition
- subtraction
<< left shift
>> right shift
& bitwise AND
| bitwise OR
^ bitwise XOR
() parenthesis

Precedence:

First, operands or sub expressions in parenthesis
Then unary negation - or complement ~
Then *, /, % left to right
Then +, - left to right
Then <<, >> left to right
Then & left to right

Then |, ^ left to right

Functions:

CRC(<expr>,<expr>,<expr>) {only used in message descriptions}
SUM(<expr>,<expr>,<expr>)
SUMW(<expr>,<expr>,<expr>)
LRC(<expr>,<expr>,<expr>)
LRCW(<expr>,<expr>,<expr>)
CRC16(<expr>,<expr>,<expr>)

```
          |       |       |
          |       |       +---- initial value usually 0 or -1
          |       +------------ ending index
          +------------------- starting index
```

CHANGED(<expr)

CTS

EXPIRED(<expr>) {Timer Function}

FLOAT(<expr>)

TRUNC(<expr>)

MIN(<expr>,<expr>)

MAX(<expr>,<expr>)

PORT

SWAP(<expr>) {reverses byte order of a word}

LOGICAL EXPRESSIONS <logical> above

Logical Operators:

AND
OR
XOR
NOT (unary)

Logical Functions:

CHANGED(R[<expr>])
CHANGED(R[<expr>] & <expr>)
R[<expr>].<const> {constant bit number 1..16}

Relational Operators:

< less than
> greater than
<= less than or equal
>= greater than or equal

= equal
<> not equal

ARITHMETIC VARIABLES

F[<expr>] 32 bit floating point register pair
R[<expr>] 16 bit register
$ the current index in a message description

MESSAGE DESCRIPTIONS

Operator:

: concatenation

Literal string:

Enclosed in quotes.
\xx where xx is two digit hex number can be used for non-printables
\" can be used to embed a quotation mark
\\ can be used to embed a \
\a - Bell, same as "\07", makes printers and terminals beep
\b - Backspace, "\08", nondestructive backspace
\f - Form feed, "\0c", top of form, clears some terminal screens
\n - New line, "\0a"
\r - Return, "\0d"
\t - Tab, "\09", advances to tab stop
\v - Vertical tab, "\0b", used by some printers with VFU

Unlike 'C', the UCM compiler accepts the above sequences in upper or lower case. These are in addition to the original UCM escape sequence:

\xx - where each x is 0..9, A..F
and last but not least: \? - where ? is any character encodes that character
which is commonly used for: \\ - literal slash or \" - literal quote
The UCM compiler does not recognize the BASIC style """" to represent "\"".

Functions:

HEX(<expr>,<expr>)
DEC(<expr>,<expr>)
UNS(<expr>,<expr>)
OCT(<expr>,<expr>)
BCD(<expr>,<expr>)
| |
| +-width in characters
+------expression in TRANSMIT
R[<expr>] in ON RECEIVE to evaluate and place result in R[]
(<expr>) in ON RECEIVE to generate and match string

RAW(R[<expr>],<expr>)
| |
| +- width in characters
+-------- starting register number

```
BYTE(<expr>)
WORD(<expr>)
RWORD(<expr>)
              |
              +----  expression in transmit
                     R[<expr>] in ON RECEIVE to evaluate and place result in R[]
                     (<expr>) in ON RECEIVE to generate and match string
TON( <expr>)
TOFF(<expr>)
           |
           +------ translation number 1..8
```

UCM RUN TIME ERROR CODES

0 - Halted by clearing RUN bit
1 - Halted by STOP or RETURN statement
2 - Execution of invalid instruction (program corrupted, compiler bug)
3 - Division by zero
4 - No memory for ON CHANGE
5 - No memory for ON RECEIVE
6 - Illegal run time call (module firmware version doesn't support compiler)
7 - Value out of bounds (register < 1 or > 2048, buffer index out of range,
SET parameter bad, output/input too long (> 256),
width specification < 0 or > 64)
8 - Checksum error in downloaded code

COMPILE.EXE Command line parameters

The COMPILE program is called using the following syntax:

COMPILE <filename> switches

<filename> refers to the text file containing the source code for the UCM. If no extension is included in the filename then .UCM is assumed.

Various switches may follow the filename including: -D, -L, -O, -U, -V, -X, and -?.

-D<macro>=<def> option is used for defining compile time macro substitution. For instance if the string "Time" is used throughout the program TEST.UCM then the command line >COMPILE TEST -Dtime=125 would set the variable time to 125.

-L<file> option is used to specify a listing file name. The default is no listing.

-O<file> option followed forces the compiler to output the compiled code into the new filename. If the -O option is not present, the compiler places the compiled data into the source filename with the extension .UCC.

-U option is used for forcing integers to be treated as unsigned numbers for purposes of multiplication, division, and modulo.

-V provides a verbose compilation with information message.

-X<file> provides Object code dump file. Default is no dump.

-? provides a help display describing the usage information.

UCMLOAD.EXE Command line parameters

The UCMLOAD program is called using the following syntax:

UCMLOAD <channel> <file>[.UCC] <port> [<route>] switches

<channel> refers to the target UCM port for the download.

<file> refers to the file containing the compiled source code for the UCM. If no extension is included in the filename then .UCC is assumed.

<port> refers to the serial device on the personal computer for the connection to the UCM. COM1:, COM2:, COM3:, or COM4: would be used to access the serial port of the computer. If an SFI-510 SY/LINK card is used, the base address in hex is used. SFI610 is used for the Ethernet board.

[<route>] is the optional SY/NET route needed to reach the target UCM.

Various switches may follow the route including: -A, -S, -C, and -?.

-A enables the autostart for the download program.

-S<reg> Moves the status register pair to reg and reg+1.

-C<count> Load long program in <count> contigious channels.

-? provides a help display describing the usage information.

UCM Reserved Word List

The following lists of words are reserved by the UCM language. These words may not be used for define macro names or labels.

AND	FALSE	ON	TIMEOUT
BCD	FLOAT	OR	TO
BAUD	FOR	PARITY	TOFF
BYTE	FULL	PORT	TOGGLE
CAPITALIZE	GOSUB	PRINT	TON
CHANGE	GOTO	PROGRAM	TRANSLATE
CHANGED	HALF	RAW	TRANSMIT
CLEAR	HEX	READ	TRUE
CRC	IDEC	READ	TRUNC
CRC16	IF	RECEIVE	UCM
CTS	LRC	RETURN	UNS
DATA	LRCW	REPEAT	UNTIL
DEBUG	MAX	RTS	VARIABLE
DEC	MIN	RWORD	WAIT
DEFINE	MODE	SET	WEND
DELAY	MOVE	STEP	WHILE
DOWNTO	MULTIDROP	SUM	WORD
DUPLEX	NEXT	SUMW	WRITE
ELSE	NOT	SWAP	WRITE
ENDIF	NONE	SYMAX	XOR
EVEN	OCT	THEN	
EXPIRED	ODD		

Appendix H
NR&D/Online BBS

Niobrara Research & Development is currently offering a Bulletin Board Service for its customers. This valuable customer service tool makes it easy to bring the user up to date on software revisions, firmware changes, product support news, and more.

This BBS operates on a 24 hour a day basis and is accessible from any personal computer equipped with a Hayes compatible modem. **NR&D/Online** will support communications from 300 to 9600 baud, at 8 data bits, NO parity, and 1 stop bit. Set your communications software for the baud rate of your modem, 8 data bits, NO parity, and 1 stop bit, then dial (417) 624-2028 to connect to **NR&D/Online**.

Once connected, you will find a message center, product bulletins, downloadable files and software, plus other news from the NR&D product support team.

Access and online time for **NR&D/Online** is free! You simply pay for your phone call.

For more information about **NR&D/Online** call Tom Fahrig at (800) 235-6723. He will take your information to allow you to log on to **NR&D/Online**.

Niobrara is now on the Internet!

http://niobrara.com

Check it out!

Index

O

P

R

S

T

U

V

W

X

<