MUCM

Installation and Programming Manual

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

Effective: 15 January, 2007

NR&D

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: www.niobrara.com

Contents

Figures

Tables

1 Introduction

The Niobrara MUCM is a user programmable communication module that is in the form factor of a Modicon TSX Momentum base. The user may write write Applications to be loaded into the MUCM to communicate with serial devices. Applications are written as a text file in a "BASIC" like language developed specifically for writing serial protocols. The Applications are compiled and downloaded into FLASH memory in the MUCM. Up to two Applications may be loaded in the MUCM and run at the same time. Each Application has access to both serial ports and the Momentum tophat interface registers. Each Application may have up to 8 parallel tasking threads. Pre-written applications are also offered to cover some of the more common protocols including Modbus RTU and ASCII, SY/MAX, SY/MAX Net-to-Net, POWERLOGIC PNIM, RNIM Master and Slave, and IDEC. The MUCM interfaces to the Momentum tophat as an I/O module with up to 32 Input (3xxxxx) registers and 32 Output (4xxxxx) registers.

The MUCM is available in two variations. MUCM102 with one RS-232 serial port and one RS-485 serial port. MUCM103 with two switch selectable RS-232 serial ports and two switch selectable RS-485 serial ports .

Specifications

Mounting Requirements

Five inches of DIN rail or can be screwed directly to surface of wall or cabinet.

Maximum Tophat Interface Addressing

32 Words In
32 words Out

Power Rating

9-30V AC or DC, 5W max (with tophat)

Operating Temperature

0 to 60 degrees C operating. -40 to 80 degrees C storage.

Humidity Rating

up to 90% noncondensing

Pressure Altitude

-200 to +10,000 feet 1ms

Serial Communication Ports

MUCM102

Two 5-pin socket connectors, one RS-232, one RS485. User selectable baud rates up to 19.2 Kbaud.

MUCM103

Four 5-pin socket connectors, two switch selectable RS-232, two switch selectable RS-485. User selectable baud rates up to 19.2Kbaud.

Memory

2 Application FLASH Areas with 128K bytes of Program size each with application 2 having a possible 256K bytes
32K bytes of Non-volatile Variable RAM memory
8K bytes of Application accessible FLASH
Two 128K bytes of Non-Volatile RAM memory as files.

Indicator lights

12 LEDs:

- Green Power and Ready
- Green User lights 1 and 2
- Red User lights 3 and 4
- Green Application 1 RUN, and Application 2 RUN
- Amber Port 1 Transmit and Receive
- Amber Port 2 Transmit and Receive

Physical Dimensions

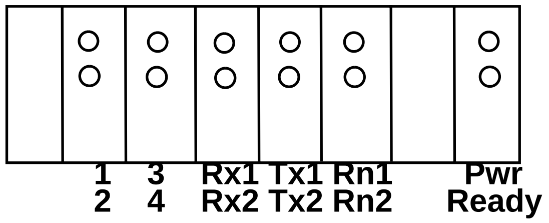
Single width module.
Wt.:.5 lb. max(without tophat)
W: 4.91 in.
H: 5.56 in.
D: 1.84 in.(with tophat)

Real-Time Clock (RTC)

All MUCM's have onboard a real-time clock, or RTC. The RTC's data is displayed in Output registers 70 through 77. All values in registers 71 through 77 should be read as a decimal value. Register 71 displays the seconds, 72 displays the minutes, 73 displays the hours in a 24-hour format, 74 displays day of the month, 75 displays the month of the year, 76 displays the year, and 77 displays the day of the week. The day of the week is displayed as a number from 0 to 6, 0 being Sunday.

Register 70 displays the status of the data contained in these registers. A hex value of E000 indicates unreliable data. A value of C000 indicates reliable data. The data will be unreliable the first time the module is powered up, and each time the voltage supplied to the RTC drops below an acceptable level. The values in registers must then be set to the desired values, and a hex value of C5C5 should be written to register 70. This tells the RTC that the data has been set, and can now be treated as reliable.

LED Indicators and Descriptions



LED	Color	Indication when ON
Pwr	Green	Power to the MUCM is present
Ready	Green	A tophat is communicating with the base
1 - 2	Green	User Lights 1 and 2
3 - 4	Red	User Lights 3 and 4
RN1	Green	Application 1 RUN
TX1	Amber	Port 1 Transmit
RX1	Amber	Port 1 Receive
RN2	Green	Application 2 RUN
TX2	Amber	Port 2 Transmit
RX2	Amber	Port 2 Receive

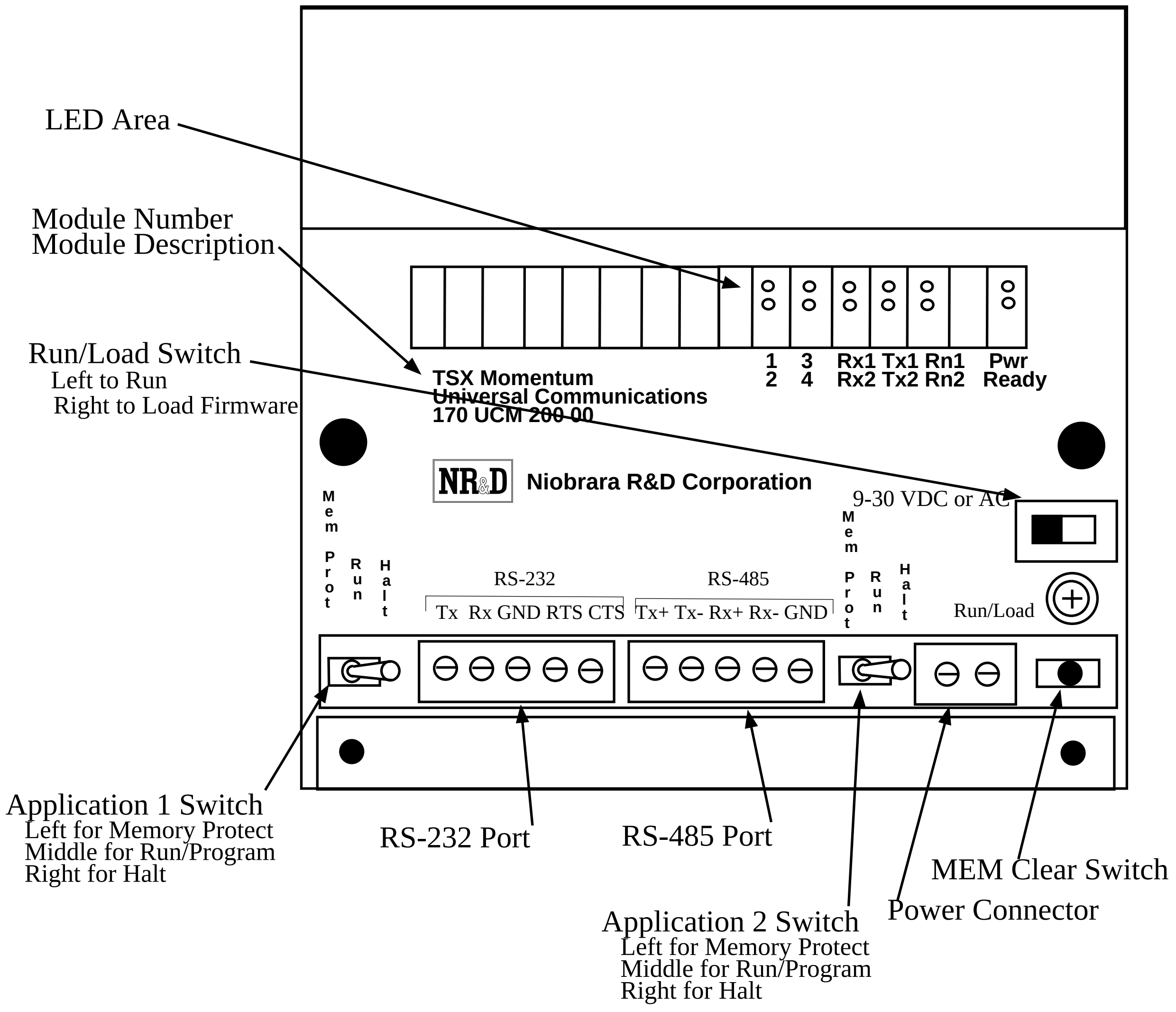


Figure 1-1 MUCM102 Front Panel

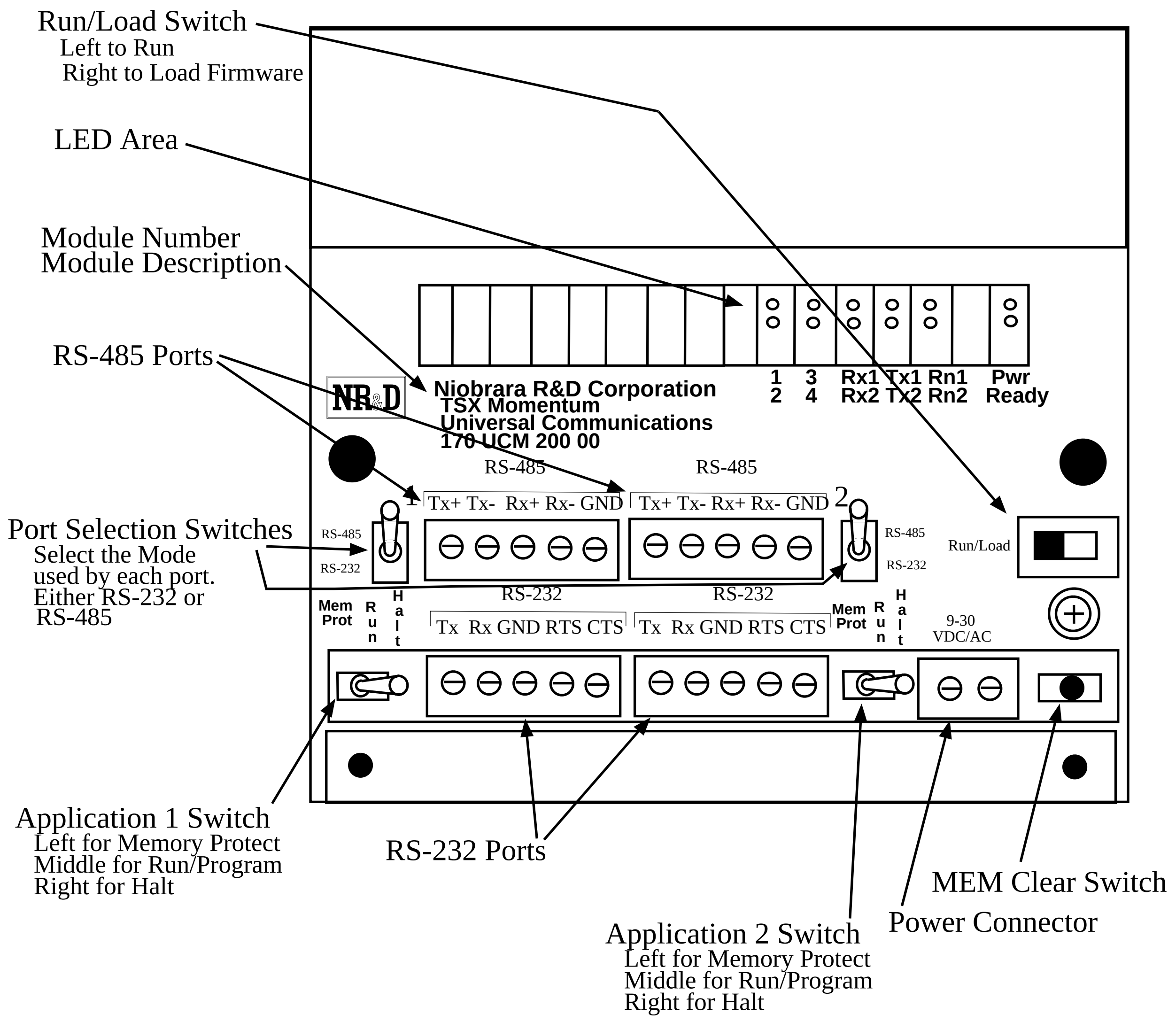


Figure 1-2 MUCM103 Front Panel

Module Installation

1 Mount the MUCM on DIN rail, or directly to a surface using the screw holes provided. The maximum tightening torque for these screws is 2-4 in-lbs.

2 With power applied to the MUCM, all LEDs should strobe and when finished, the green Power LED should illuminate and remain lit. This indicates that the MUCM has passed its internal self checks and is ready.

3 If the MUCM has a Momentum tophat installed, and the tophat is communicating to the MUCM, then the green Ready LED should illuminate.

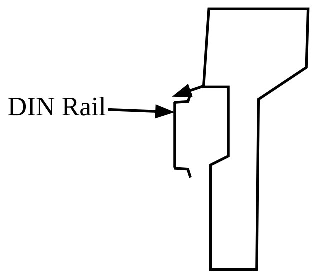


Figure 1-3   Mounting the MUCM on a DIN Rail

Serial Installation

The MUCM connects to external serial devices through Port 1 or Port 2 using the switch selectable modes RS-232 or RS-485.

Momentum Tophat Configuration

A maximum of 32 words of Input and 32 words of Output may be accessed via the Momentum Tophat Interface. These words are accessed like other I/O bases.

2
MUCM Programming Overview

The user programs¤ that run in the MUCM are known as Applications. Applications are written in the¤ MUCM language with a text editor, compiled with the QCOMPILE program, and downloaded into the MUCM to run. The MUCM allows up to two Applications to run at the same time. Each Application has its own separate memory for variables as well as shared access to the PLC Rack I/O interface via the INPUT[x] registers and the OUTPUT[x] registers. These I/O words are the only directly common memory connection between the two Applications although Applications may share data through RAM files that are accessible using the READ FILE and WRITE FILE structures. Applications may be divided into multiple THREADs which multi-task within the Application. Up to eight THREADs may be written into an Application.

Each Application has full access to both serial ports, and as mentioned above the PLC I/O registers. Communication messages are sent from an Application using the TRANSMIT statement and are received with the ON RECEIVE statement. Built-in functions for calculating checksums are provided.

The general outline for an MUCM application is shown below:

```
{Comments}
DECLARE global_variables
FUNCTIONS
{general startup configuration code}
THREAD 1
        DECLARE local_variables
        {thread 1 application code as an endless loop}
THREAD 2
        DECLARE local_variables
        {thread 2 application code as an endless loop}
```

Application code located before thread 1 is processed first as the application starts and then all threads start at the same time. Declares located before thread 1 are global and accessible in any of the threads. Declares within a thread are local only to that thread.

3
MUCM Language Definitions

The MUCM 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 MUCM program then they should be treated as constants. The MUCM supports unsigned decimal integers (16 bits), signed decimal integers, hexadecimal integers, long integers (32 bit), floating point numbers (32 bit), boolean constants, and a few reserved constants. The use of a constant is referred to as <const> in this manual.

Table 3-1 Constant Data Types

Constant Data Type	Range	Prefix Symbol
Decimal	0...65,535	NA
Signed Integer	-32768...32767	NA
Hexadecimal Integer	0...FFFF	x
Long Integers	0...4294967295	NA
Floating Point	8.43 x 10E-37... 3.402 x 10E38	
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
```

Floating Point Numbers

Floating point constants must end with a decimal point and at least one decimal place. The following are valid floating point examples:

```
-1.0
3.14159
2.5E-11
```

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

The MUCM uses alpha-numeric names for variables and each variable must be explicitly declared using the DECLARE statement. The possible variable types supported by the MUCM are listed below:

- BYTE (8 bits signed)
- UNSIGNED BYTE (8 bits unsigned)
- WORD (16 bits signed)
- UNSIGNED WORD (16 bits unsigned)
- LONG (32 bits signed)
- TIMER (32 bits signed)
- FLOAT (32 bits signed)
- STRING (an array of 8 bit bytes)

- SOCKET (IP socket)

If a type is not included in the DECLARE then the type defaults to a SIGNED WORD.

It is also possible to define single dimensional arrays of variables using the form variable[size], and two-dimensional arrays using the form variable[Asize, Bsize]. Valid array indices for array[N] are 0..(N-1).
Multiple variables may be declared on a single statement with commas as separators.
The following statements are valid DECLARE examples:

```
        DECLARE BYTE apple
        DECLARE WORD x, y, zebra
        DECLARE WORD r[100], group[10]
        DECLARE SOCKET s[8], mysock
        DECLARE STRING in[25]
        DECLARE WORD a, b, c FLOAT x, y, z  {the , after the c is optional.  a, b, and c are
words and x, y, and z are floats.}
```

There are two predefined arrays of words that are fixed and reserved: INPUT[x] and OUTPUT[x]. The INPUT[x] array ranges from index 0 through 31 inclusive and refers to the 32 possible PLC input (3x) registers on the backplane. These words are PLC Read-Only and may be modified only by the MUCM applications. The OUTPUT[x] array ranges from index 0 through 2015. Index values 0 through 31 are reserved for the 32 possible PLC OUTPUTs (4x registers) and are Read-Only to the MUCM applications. OUTPUT[32] through OUTPUT[2015] are Read/Write by the MUCM Applications.
The OUTPUT and INPUT variables are global to both Applications and all Threads within the Applications. Variables declared before the first THREAD statement are global to a given Application. Variables declared within a THREAD are local to that Thread.

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 3-2 Numeric Operators

Numeric Operator	Description	Example
+	Addition	x + 5
-	Subtraction	OUTPUT[10] - 5
*	Multiplication	apple * 5
/	Division	z / 5
%	Modulus	OUTPUT[25] % 5
&	Bitwise AND	OUTPUT[25] & x100
|	Bitwise OR	OUTPUT[25] | x100
^	Bitwise Exclusive OR	INPUT[25] ^ x100
>>	Bitwise Shift Right	BYTE >> 4
<<	Bitwise Shift Left	I << 2
-	Unary Negation	-OUTPUT[25]
~	Unary Bitwise Complement	~OUTPUT[25]
()	Parentheses	(OUTPUT[25] + 5) * 3

Precedence of Operators

The order of precedence of supported numeric operators are 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.
5	<<, >>	From left to right within the expression.
6	&, ^, |	From left to right within the expression.

Numeric Functions

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

Table 3-3 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
CRCBOB(<expr>,<expr>,<expr>)	Special CRC16 for BinMaster SmartBob applications
CRCDNP(<expr>,<expr>,<expr>)	Special CRC16 for DNP 3.00 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 3-4 Additional Functions

Function	Description	Example OUTPUT[45]=x1234, OUTPUT[46]=xABCD
MIN(<expr>,<expr>)	Provides a result of the <expr> which evaluates to the smaller of the two expressions.	OUTPUT[100] = MIN(OUTPUT[45],OUTPUT[46]) results in OUTPUT[100] = x1234
MAX(<expr>,<expr>)	Provides a result of the <expr> which evaluates to the larger of the two expression.	OUTPUT[100] = MAX(OUTPUT[45]*x0A,OUTPUT[47]) results in OUTPUT[100] = x65E0
SWAP(<expr>)	Reversed the byte order of the register.	OUTPUT[100] = SWAP(OUTPUT[46]) results in OUTPUT[100] = xCDAB

Labels - <label>

The MUCM 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 an MUCM language reserved word. The label TIMEOUTLoop: is valid while TIMEOUT: is not valid.

Logical Expressions - <logical>

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

Logical Operators

Table 3-5 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 3-6 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

Functions - <function>

Functions are general purpose sections of code that may be accessed from multiple threads and other functions in an application. Functions are similar to a subroutine where the parameters are passed to/from the function during the call.

Memory for variables declared within a function are allocated when the function is called, and the memory is freed when the function exits. Variable names within a function can have the same name as global or thread local variables. When a variable is referenced within a function, the compiler checks first for function local variables, then for thread local variables, then for global variables by that name.

NOTE: At the present time, only "word" variables may be passed as parameters to functions. If the function must process long, byte, string, float, or arrays then they must be declared as global.

```
FUNCTION <function name> <comma separated variable list>
          (function body)
ENDFUNC <returned variable list>
```

```
FUNCTION AVERAGE ( VALUE1, VALUE2 )
    DECLARE WORD RETURNVALUE
    RETURNVALUE = ( VALUE1 + VALUE2) / 2
ENDFUNC( RETURNVALUE )
```

--or--

```
FUNCTION SQUARE ( VALUE)
ENDFUNC ( VALUE * VALUE)
```

Message Descriptions - <message description>

The <message description> refers to the actual serial data that is transmitted from the MUCM 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 \\.

String Variables

String variables may be embedded directly into a message description:

```
DECLARE STRING ALPHA[ 20]
ALPHA = "ABC123"
TRANSMIT PORT 1 "=BEFORE=":ALPHA:"=AFTER="
```

would send the string =BEFORE=ABC123=AFTER= out serial port 1. Similarly, string variables may be embedded directly into ON RECEIVE statements:

```
ON RECEIVE PORT 1 ALPHA:"\0D" GOTO NEXT
```

would place all characters received before the Carriage Return (0x0D) into the string variable ALPHA. Care must be taken to ensure that the data read into the string is not longer than the string declaration. For instance, if the above ON RECEIVE were to attempt to put 21 characters into ALPHA, which was declared with a length of 20 bytes, the program would halt, with runtime stop code 7 (Value out of bounds).

Message Functions

The MUCM can perform a variety of functions on transmitted and received data. When the MUCM is using these functions for transmitting, register data and expressions are turned into strings according to the function's rules. When the MUCM is using these functions for receiving, incoming strings are either matched to the strings that the MUCM 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 42 through 44.

Table 3-7 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(<variable>,<expr>)	Sends/Receives high byte then low byte of a register(s)
RWORD(<expr>)	Sends/Receives low byte of an expression
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 <variable> to have the incoming string translated and placed into the register OUTPUT[] 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(<variable>,<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 <variable> to receive and place in a register or (<expr>) to evaluate and match. For examples of the message functions see Chapter 6 - 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(OUTPUT[10],VARIABLE). The second is VARIABLE followed by a register reference, i.e. TRANSMIT HEX(OUTPUT[11],VARIABLE OUTPUT[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 OUTPUT[11] = 1234 then

```
TRANSMIT PORT 1 "$":DEC(OUTPUT[11], variable OUTPUT[10]):"#"
```

would send out the string $1234# and OUTPUT[10] would have the value 4. If OUTPUT[11] = 89 then the string $89# would be transmitted and OUTPUT[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 OUTPUT[11]=x486F, OUTPUT[12]=x7764, and OUTPUT[13]=x7900 then

```
TRANSMIT PORT 1  "$":RAW(OUTPUT[11], VARIABLE OUTPUT[10]):"#"
```

would send the string $Howdy# and OUTPUT[10] would equal 5. If OUTPUT[12]=x0000 then the string $Ho# would be transmitted and OUTPUT[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  port 1 "$":dec(OUTPUT[100],variable):".":dec(OUTPUT[101],2)
```

In the case of $125.01, register OUTPUT[100] = 125 and OUTPUT[101] = 1. For $3.99, register OUTPUT[100] = 3 and OUTPUT[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 port 1 RAW(OUTPUT[500], variable OUTPUT[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 a variable in the MUCM rather than transmit the string. This is possible by simply using the assignment character = to a string variable.

```
STRINGVARIABLE = <message>
```

The message will be placed in the string variable and the LENGTH of the string will be set to the number of character is <message>. Any valid transmit message may be stored in this manner.

For example:

```
STRINGVAR = "Hello!\0d\0a"
```

would result in the string STRINGVAR containing the string "Hello!\0d\0a" (where \0d and \0a are Carriage Return, and Line Feed, respectively).

Something more obviously useful might be:

```
STRINGVAR = byte( Device):"\03":word( Address):word( Count):rword(crc16(1,$-1,0))
```

which would place the reversed word of the checksum in register at the end of the string.

4
MUCM Language Statements

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

The format of these statements includes the definitions from Chapter 3 - MUCM 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 MUCM program is ignored by the MUCM 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 MUCM. The command SET CAPITALIZE can effect the way the MUCM handles ASCII characters on transmitting and receiving.

Program flow within a THREAD 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 MUCM language allows for the assignment of values to variables and bits of variables. These assignments are similar to the BASIC LET statement.

variable[<expr>]=<expr>

This statement sets the variable specified by the first <expr> to the value obtained by the second <expr>. The valid range of variable numbers in the first <expr> is dependent upon the DECLARED range of the variable.

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

This statement sets a single bit of a variable to be one (TRUE) or zero (FALSE). The <expr> can have the values defined by the DECLARE of the variable. The valid values for <const> depend on the type of <Expr> (see Table 5-1, below). <Logical> can have the values TRUE or FALSE.

Table 4-1 Referencing Bits in Different Variable Types

Variable Type	Range of Bits	Bit Significance
OUTPUT[N] and INPUT[N]	1...16	Modicon Bit Numbering: Most Significant Bit (MSB) = Bit 1 ... LSB = Bit 16
BYTE	0...7	IEC Compliant Bit numbering: MSB = Bit 7 ... LSB = Bit 0
WORD	0...15	IEC Compliant Bit numbering: MSB = Bit 15 ... LSB = Bit 0
LONG, TIMER	0...31	IEC Compliant Bit numbering: MSB = Bit 31 ... LSB = Bit 0

<variable>.(<expr>)=<logical>

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

<variable>.<variable>=<logical>

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

<variable>=<message description>

This statement sets the string variable specified by the <expr> to the ASCII values obtained by evaluation of the <message description>. The <message description> may be any valid message used in a TRANSMIT command.

BAUD

See **SET BAUD** on page 32.

CAPITALIZE

See **SET CAPITALIZE** on page 32.

CLEAR

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

The CLEAR statement sets a single bit of a variable to ZERO. The bit number <const> or <expr> must evaluate within the range of 1-16 for OUTPUT registers, 0-7 for bytes, 0-15 for words, and 0-31 for long variables. To clear a single bit of a register to be set to one use the SET statement.

CLOSE

CLOSE SOCKET <socket variable> [TIMEOUT <expr>]

Closes the open IP connection associated with <socket variable>. The optional TIMEOUT specifies how long the MUCM TCP/IP stack will wait for the other device to acknowledge the request to close the connection before aborting (resetting) the connection. If no TIMEOUT is specified, the MUCM will wait indefinitely for the other device to acknowledge the close request. A TIMEOUT value of zero will cause the connection to be immediately closed, without the other devices' acknowledgment.

CONNECT

CONNECT <protocol> SOCKET <socket variable> <IP Address> PORT <port number>

Connect opens an IP connection using the <protocol> to the remote <IP Address> on the <port

number>.

NOTE: Only <protocol>= TCP is presently supported.

The <IP Address> must be a comma separated decimal notation:

```
DECLARE SOCKET S, BYTE HOST[4]
HOST = 206,223,51,161
CONNECT TCP SOCKET S HOST PORT 80
```

would establish a connection to port 80 of the device with IP address 206.223.51.161.

DATA

See **SET DATA** on page 33.

DEBUG

See **SET DEBUG** on page 33.

DECLARE

DECLARE [SIGNED|UNSIGNED] [<variable type>} <variable name>[[array size]]

The DECLARE statement is a compiler instruction which creates a variable named <variable name> of type <variable type>. Variables may be declared anywhere in the program, as long as they are declared before they are referenced. Variables declared before the first THREAD statement will bel global in scope, thus will be accessible to all the threads. Variables declared after a THREAD statement will be accessible only within the thread in which it was declared. Variables declared within a FUNCTION will be accessible only within that function.

If the SIGNED/UNSIGNED specification is omitted, the variable created will be SIGNED. If the <variable type> is omitted, a WORD variable will be created. Thus the statement:

```
DECLARE FOO
```

will create a variable named FOO, which is a signed word variable. Multiple variable types may be declared in one DECLARE statement:

```
DECLARE BAR, STRING A[40], B[30], FLOAT X, Y[10]
```

would create five variables: BAR is a signed word, A is a string with maximum length of 40 bytes, B is a string with a maximum length of 30 bytes, X is a floating point variable, and Y is an array of ten floating point variables (Y[0] ... Y[9]).

When a variable is referenced (i.e. Y[0] = 0.0), the compiler first checks whether the variable is a function local variable (if the statement is inside a function), then checks whether the variable is a thread local variable (if the statement is multi-threaded, and the statement appears after a THREAD statement), then checks whether the variable was defined as a global variable. Thus, the same variable name may be used in different threads, and each thread will access a different variable.

The available <variable types> are:

Table 4-2 Declared Variable Types

Variable Type	Description	Bytes Used	Range
UNSIGNED BYTE	Unsigned Byte (8-bit) variable.	1	0...255
SIGNED BYTE	Signed Byte (8-bit) variable.	1	-128...127
UNSIGNED WORD	Unsigned Word (16-bit) variable.	2	0...65535
SIGNED WORD	Signed Word (16-bit) variable	2	-32768...32767
UNSIGNED LONG, TIMER	Unsigned Long Word (32-bit) variable.	4	0...4294967296
SIGNED LONG	Signed Long Word (32-bit) variable.	4	-2147483648...2147483647
FLOAT	IEEE format 32-bit Floating Point variable. Float variables are always signed.	4	
STRING	String variable. Must be declared as an array: DECLARE STRING A[40]	2 + String Length	Strings in the MUCM are NOT zero-terminated, thus each byte may contain ANY value, including zero.
SOCKET	Socket structures are used for TCP Ethernet connections. Values in the structure are not directly accessible, except through statements (CONNECT, LISTEN, TRANSMIT, ON RECEIVE) and functions (SOCKETSTATE ()).	1538	

DEFINE

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

The DEFINE statement is a compiler instruction for a global find and replace. When the MUCM 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 MUCM to pause in its execution of other instructions until a period of time equal to <expr> times 1ms has expired. Valid range is 0 to xFFFFFFFF.

DUPLEX

See **SET DUPLEX** on page 33.

ERASE

ERASE <variable>

The ERASE command initializes a variable or array to zero.

EXPIRED

ON EXPIRED(<variable>) GOTO <variable>

IF EXPIRED(<variable>) THEN <expression>

The EXPIRED command is used in conjunction with a declared timer to allow the user to perform other functions based on a timeout. A timer is declared, and a value in milliseconds is assigned in one or two commands. The user can then use the EXPIRED command to check if the timer has run out.

FOR...NEXT

The FOR ... NEXT statement provides the ability to execute a set of instructions a specific number of times. The variable <variable> 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 <variable> is incremented by the value equal to the STEP <expr>. If the STEP <expr> is not present a step of 1 is assumed.

```
FOR <variable>=<expr> TO <expr>
one or more statements
NEXT
```

```
FOR <variable>=<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 <variable>=<expr> DOWNTO <expr>
one or more statements
NEXT
```

```
FOR <variable>=<expr> DOWNTO <expr> STEP <expr>
one or more statements
NEXT
```

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

FLUSH

FLUSH PORT x

The FLUSH statement empties the receive buffer for the specified port.

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 MUCM 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 33.

LISTEN

LISTEN <protocol> SOCKET <socket number> PORT <protocol port number>

The listen command instructs the PPP port to use a socket to listen for a particular protocol on a given port number. Presently only the TCP protocol is supported.

Example: LISTEN TCP SOCKET mysock PORT 502

Common TCP port numbers are shown in Table 4-3.

Table 4-3 Well Known TCP Port Numbers

Well Known Port Number	TCP Protocol	Associated RFC[1]
21	FTP	959
23	TELNET	854
25	SMTP	821
80	WEB Server (HTTP)	2616
110	POP3	1939
502	Modbus/TCP	N/A[2]

[1] Internet Protocols are available as Requests For Comment (RFCs). They are available on the Internet via HTTP: http://www.rfc-editor.org
[2] The Modbus/TCP specification is available http://www.modicon.com/openmbus/

MOVE

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

MULTIDROP

See **SET MULTIDROP** on page 34.

ON CHANGE

ON CHANGE <variable> GOTO <label>

ON CHANGE <variable> RETURN

ON CHANGE <variable> & <expr> GOTO <label>

ON CHANGE <variable> & <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 <variable>. When the value in <variable> is modified by another source, the ON CHANGE statement is performed.

ON <expression>

ON <expression> GOTO

ON <expression> RETURN

When the expression evaluates TRUE the wait loop is exited and flow proceeds to the GOTO or RETURN.

ON RECEIVE PORT x

ON RECEIVE SOCKET x

ON RECEIVE port 1 <message description> GOTO <label>

ON RECEIVE socket <socket name> <message description> GOTO <label>

ON RECEIVE port 2 <message description> RETURN

ON RECEIVE SOCKET <socket name> <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 1ms.

PARITY

See **SET PARITY** on page 34.

READ FILE

READ FILE <file number> OFFSET <offset value> <variable,variable,...>

The READ FILE statement allows an MUCM program to read memory from the 6x file areas of the MUCM to the user memory area.

The <file number> is an expression which evaluates a number in Table 4-4.

Table 4-4 MUCM Internal File List

File Number (dec)	File Number (hex)	Memory Description	Memory Size
256	100	Application 1 Program	128K bytes
384	180	Application 1 Variables	32K bytes
512	200	Application 2 Program	128K bytes
640	280	Application 2 Variables	32K bytes
768	300	General Use Ram Block 1*	128K bytes
1024	400	General Use Ram Block 2*	128K bytes
1792	700	Application 2 Program Extension	128K bytes
2560	A00	Flash Block 1	8K bytes
2816	B00	Flash Block 2	8K bytes

The <offset> is an expression which evaluates to the byte location for the start of the read.

*NOTE: MUCM-SE RAM block 1 at file 768 is only 16K bytes and block 2 does not exist. Attempting to access these RAM areas will result in a runtime error.

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 FALSE. When the <logical> is TRUE, 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 only execute while the <logical> is TRUE, and will not execute the program statements within their boundaries if the <logical> is FALSE.

RETURN

See **GOSUB...RETURN** on page 29.

SET

The SET statement allows the initialization of the MUCM for the following parameters: Baud rate, Capitalization of incoming characters, Data bits, Parity, Stop bits, and Debug mode. SET must be followed by the serial port number for the action to take place.

SET PORT x 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. Example: SET PORT 1 BAUD 9600

SET PORT x CAPITALIZE <const>
SET SOCKET x CAPITALIZE <const>

The SET CAPITALIZE statement performs a translation on incoming ASCII alphabet characters from the lower case to the upper case. Example: SET PORT 2 CAPITALIZE TRUE or SET PORT 1 CAPITALIZE FALSE.

SET PORT x CTS <const>

The SET CTS statement sets the operation of the CTS pin on the RS-232 port. Possible values are
CTS ON - The is the normal mode of CTS where CTS must be asserted to allow the serial port to transmit.
CTS OFF - Allows the use of the CTS pin to be independently monitored for its state while the serial port is allowed to transmit regardless of the state of CTS. This operation is very useful in modem applications where CTS is wired to DCD on the modem so the MUCM can tell if the modem has carrier.

SET PORT x 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 PORT 1 DATA 8

SET DEBUG <const>

The SET DEBUG statement determines the operation of the MUCM port in the event of a run time error. If SET DEBUG TRUE is used, the MUCM 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 MUCM program will halt upon a run time error and immediately restart the program from the beginning.

SET PORT x DUPLEX <const>

The SET DUPLEX statement determines the operation of the port's receiver. With DUPLEX HALF, the receiver is only turned on when the port is not transmitting. With DUPLEX FULL, the receiver is always on. DUPLEX HALF should is used in 2-wire applications.

SET LIGHT <exp> <const>

The SET LIGHT statement is used to determine the state of the 10 indicator lights for the MUCM. SET LIGHT 1 ON turns on the light while SET LIGHT 1 OFF turns off the light. See also TOGGLE LIGHT on page 36.

SET MODE <const>

The SET MODE statement determines the operating mode of the port. Valid entries are UCM, RTU, SYMAX, RNIM, and PPP.

UCM mode allows the use of raw TRANSMIT and RECEIVE statements to communicate with the external device. Example: TRANSMIT PORT 1 "Example string"

RTU mode gives the MUCM more automatic control of the TRANSMIT and RECEIVE statements. This mode lets the MUCM assume that the communication will be Modbus RTU. The programmer will create a Modbus packet in a byte array, then hand the MUCM a length and the name of the array. During TRANSMIT, the MUCM will calculate and append the checksum to the end of the packet. During RECEIVE, the MUCM will watch for the 3.5 character timeout, then verify the checksum. The MUCM will then replace the data in the array with the new data from the reply.

```
DECLARE  UNSIGNED BYTE CMD[100]
DECLARE WORD CMDLEN
...
TRANSMIT PORT 1 WORD(CMDLEN):RAW(CMD,CMLEN)
ON RECEIVE PORT 1 WORD(CMDLEN):RAW(CMD,CMLEN) GOTO <variable>
```

SYMAX mode works on the same principle as RTU mode. The MUCM will assume that the following communication is SY/MAX, and will handle checksums, ACK's, DLE escapes, etc. , involved in SY/MAX communications. During TRANSMIT, the programmer will hand the MUCM the length of the SY/MAX packet data, the SY/MAX route escaped by xFF, and the SY/MAX packet data. During RECEIVE, the MUCM will hand the programmer, the length of the reply, the route escaped by xFF, and the SY/MAX reply data.

```
DECLARE STRING ROUTE[16], REPLYDATA[200]
DECLARE WORD REMOTE, COUNT, REPLYLEN
...
TRANSMIT PORT 1 WORD(6):RAW(ROUTE,LENGTH(ROUTE)):"\FF":"\00\03":
WORD(REMOTE):WORD(COUNT)
ON RECEIVE PORT 1  WORD(REPLYLEN):"\11":RAW(ROUTE,4):"\FF":"\86\03":
WORD((REMOTE)):RAW(REPLYDATA,REPLYLEN-4) GOTO <variable>
```

RNIM mode is nearly identical to SYMAX mode. The only differences are the addition of a Network ID, transaction number, and a drop before the route.

```
DECLARE STRING ROUTE[16], REPLYDATA[200]
DECLARE WORD DROP, TRANSNUM, REMOTE, COUNT, REPLYLEN
...
TRANSMIT PORT 1 WORD(6):BYTE(DROP):BYTE(TRANSNUM):"\00":
RAW(ROUTE,LENGTH(ROUTE)):"\FF":"\00\03":WORD(REMOTE):WORD(COUNT)
ON RECEIVE PORT 1  WORD(REPLYLEN):"\11":BYTE(TRANSNUM):RAW(ROUTE,4):
"\FF":"\86\03":WORD((REMOTE)):RAW(REPLYDATA,REPLYLEN-4) GOTO <variable>
```

PPP Mode allows the MUCM or MUCM to use the serial port for TCP/IP communication using the PPP protocol.

SET SOCKET <socket> NAGLE <const>

The Set Socket Nagle statement controls how data is sent out a TCP/IP connection. In a socket with NAGLE OFF, every TRANSMIT SOCKET command will create its own Ethernet packet. In a socket with NAGLE ON (The default state), data sent out the socket is buffered as necessary by the MUCM, which results in larger packets and better throughput, especially for applications such as a Telnet server or a WWW server.

SET PORT x MULTIDROP <const>

The SET MULTIDROP statement controls the operation of the port's transmitter. With MULTIDROP TRUE, the transmitter is only on while transmitting. With MULTIDROP FALSE, the transmitter is always on.

SET PORT x RTS <const>

The SET RTS statement sets the operation of the RTS pin on the RS-232 port. Possible values are
RTS ON - Forces RTS on continuously
RTS OFF - Forces RTS off continuously
RTS AUTO - Allows RTS to behave in normal Push-to-Talk operation

SET PORT x 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 PORT 1 DATA 8

SET PORT x PARITY <const>

The SET PARITY statement determines the parity of the port. Valid entries are EVEN, ODD, or NONE. Example: SET PORT 1 PARITY EVEN

SET PORT x PPPUSERNAME <string const|string variable>

The SET PPPUSERNAME statement determines username for the PPP connection between the MUCM and the PPP client or server.

SET PORT x PPPPASSWORD <string const|string variable>

The SET PORT x PPPPASSWORD statement determines password for the PPP connection between the MUCM and the PPP client or server.

SET PORT x PPPHANGUP

The SET PORT x PPPHANGUP statement causes a graceful disconnect between the PPP connection of the MUCM and the client/server.

SET PORT x STOP <const>

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

SET (bit)

SET <variable>.<const> or SET <variable>.(<expr>)

The SET statement sets a single bit of a variable to ONE. The bit number <const> or <expr> must evaluate within the range off 1-16 for OUTPUT registers, 0-7 for bytes, 0-15 for words, and 0-31 for long variables. To clear a single bit of a register to be set to one use the CLEAR statement.

SOCKETSTATE

ON SOCKETSTATE (<socket>).<const> GOTO <label>

ON SOCKETSTATE (<socket>).<const> RETURN

IF SOCKETSTATE (<socket>).<const> THEN <expression>

The SOCKETSTATE statement allows the Application to make decisions based on the status of a socket that was initiated by a CONNECT statement. Status bits are set for the SOCKETSTATE of each declared socket. Bit 15 indicates when a socket is open. Bit 14 indicates that the socket is listening. These are the most useful bits.

STOP

The STOP statement causes the MUCM 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 35.

SWITCH...CASE...ENDSWITCH

SWITCH CASE<expr><statement(s)> [CASE <expr> <statement(s) ...] ENDSWITCH

The SWITCH...CASE...ENDSWITCH construct allows many mutually exclusive conditional statements or routines to be written without nesting a lot of IF...ELSE...ENDIF statements. Only one of the CASEs contained within the SWITCH...ENDSWITCH construct will be executed. For Example:

```
SWITCH
    CASE X=2
        Y = 2 * Y  {Will execute only if X = 2}
    CASE X < 5
        Y = X * 5  {Will execute only if X < 5, but not if X = 2}
    CASE Y > 10  {Logical expressions can operate on different variables}
        Y = 0
    CASE TRUE  {Comparable to default: in C}
        Y = 99
        X = 0  {These will execute only if all other CASEs fail to match}
ENDSWITCH
```

Program execution will continue with the instruction immediately after the ENDSWITCH statement, whether any CASE matches or not.

TOGGLE

TOGGLE <variable>.<const> or TOGGLE <variable>.(<expr>)

The TOGGLE statement changes the state of a single bit of a variable. The bit number <const> or <expr> must evaluate within the range off 1-16 for OUTPUT registers, 0-7 for bytes, 0-15 for words, and 0-31 for long variables.

TOGGLE LIGHT

TOGGLE LIGHT <expr>

The TOGGLE LIGHT statement is used to change the state of the 10 indicator lights for the MUCM. See also the SET LIGHT command on page 33.

TRANSMIT

TRANSMIT PORT x <message description>

TRANSMIT SOCKET s <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, ON <expression>, 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, <expression>, 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 FILE

WRITE FILE <file number> OFFSET <offset value> <variable,variable,...>

The WRITE FILE statement allows an MUCM program to write memory from the user memory area to memory in the 6x file areas of the MUCM

The <file number> is an expression which evaluates a number in Table 4-5.

Table 4-5 MUCM Internal File List

File Number (dec)	File Number (hex)	Memory Description	Memory Size
256	100	Application 1 Program	128K bytes
384	180	Application 1 Variables	32K bytes
512	200	Application 2 Program	128K bytes
640	280	Application 2 Variables	32K bytes
768	300	General Use Ram Block 1	128K bytes
1024	400	General Use Ram Block 2	128K bytes
1792	700	Application 2 Program Extension	128K bytes
2560	A00	Flash Block 1	8K bytes
2816	B00	Flash Block 2	8K bytes

The <offset> is an expression which evaluates to the byte location for the start of the read.

*NOTE: MUCM-SE RAM block 1 at file 768 is only 16K bytes and block 2 does not exist. Attempting to access these RAM areas will result in a runtime error.

5
MUCM Language Functions

The MUCM 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 the 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.

CRCBOB

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

The CRCBOB function calculates the CRC16 Check upon a message while leaving out the $-2 character. The first <expr> is the starting index. This value is the 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 -1.

The CRCAB is a variation of the CRC16 customized for use with BinMaster Smartbob II's.

CRCDNP

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

The CRCDNP function calculates the CRC16 Check upon a message while leaving out the $-1 character. The first <expr> is the starting index. This value is the 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 DNP 3.00 protocol.

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 MUCM register.

ON RECEIVE formats: BCD(<variable>,<expr>) or BCD((<expr>),<expr>)

Note: The MUCM 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 MUCM registers and zeros the upper byte of these registers.

ON RECEIVE formats: BYTE(<variable>) or BYTE((<expr>))

Note: If the MUCM 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 MUCM 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 MUCM register.

ON RECEIVE formats: DEC(<variable>,<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 MUCM registers.

ON RECEIVE formats: HEX(<variable>,<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 MUCM registers. Transmits the hex alpha characters as lower case a through f.

ON RECEIVE formats: HEXLC(<variable>,<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 MUCM registers.

ON RECEIVE formats: IDEC(<variable>,<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.

LONG

Usual Format: LONG(Variable name)

Valid characters: hexadecimal 00 through FF

Transmitting: Converts an expression into its 32-bit hexadecimal representation, translates the 32-bit number into four 8-bit hexadecimal numbers and transmits the bytes in order of descending significance. If the variable VAR of type long contains 0x12345678, the four bytes would be transmitted: x12, x34, x56, x78.

TRANSMIT format: LONG(<expr>)

Receiving: Interprets four hexadecimal characters as four 8-bit hexadecimal numbers, assembles the four 8-bit numbers into a 32-bit number, first number the high byte, the second number in the second most significant byte, and the fourth number the low byte, and then compares the number to an expression or places the number into an MUCM variable.

ON RECEIVE formats: LONG(<variable>) or LONG((<expr>))

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(<variable>,<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(<variable>,<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 MUCM registers.

ON RECEIVE formats: RAW(<variable>,<expr>) or RAW((<expr>),<expr>)

Note: If the MUCM 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 MUCM register.

ON RECEIVE formats: RWORD(<variable>) 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 MUCM 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 MUCM 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(STRINGVAR,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 MUCM register.

ON RECEIVE formats: UNS(<variable>,<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 MUCM register.

ON RECEIVE formats: WORD(<variable>) or WORD((<expr>))

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

Other Functions

APPLICATION

The APPLICATION internal variable returns a value of 1 or 2, indicating which application area the program is running in. A program which is loaded into application area 2 of an MUCM will read this variable as 2.

CHANGED

Format: CHANGED(<variable>) or CHANGED(<variable> & <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(OUTPUT[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.

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 OUTPUT[4] = xABCD then SWAP(OUTPUT[4]) would bring the result xCDAB.

THREAD

The THREAD variable returns a value for the thread number where the variable is called. Valid results are 1-8 inclusive.

RTS

RTS is a variable which may be used to control the state of the Request to Send line for an MUCM port. SET PORT 1 RTS ON will assert the RTS line. SET PORT 2 RTS OFF will negate the RTS line. SET PORT 1 RTS AUTO will force RTS to be in "push-to-talk" mode.

CTSx

CTSx is a variable which gives the current state of Clear to Send on the MUCM port. CTS1 provides the state for port 1 while CTS2 is for port 2. IF CTSx = TRUE then CTS is asserted by the external device. If CTSx = FALSE then CTS is negated.

6

Examples

TRANSMIT message function with register references

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

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

TRANSMIT HEX

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

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

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

Command: TRANSMIT HEX(OUTPUT[23],VARIABLE OUTPUT[600])
ASCII Characters transmitted: A1B2
Decimal values: 65 49 66 50
Hex values: 41 31 42 32

OUTPUT[600] would then equal 4.

TRANSMIT DEC

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

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

TRANSMIT UNS

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

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

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

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

TRANSMIT BCD

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

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

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

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

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

Several of the examples have remaining characters. The remaining characters will be received by the MUCM 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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[23],4) may be better.

Remaining characters: "43F"

Command: ON RECEIVE DEC(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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(OUTPUT[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 a hex 44 and the "8" is a hex 38 so register 23 is now 4438
Remaining characters: "76543F"

Command: ON RECEIVE RAW(OUTPUT[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(OUTPUT[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(OUTPUT[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 OUTPUT[23] is changed.
Characters remaining: "876543F"

ON RECEIVE WORD

Command: ON RECEIVE WORD(OUTPUT[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 OUTPUT[23] is changed.

Characters remaining: "76543F"

ON RECEIVE RWORD

Command: ON RECEIVE RWORD(OUTPUT[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 OUTPUT[23] is changed.

Characters remaining: "76543F"

7 Compiling

QCOMPILE.EXE

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

The QCOMPILE command syntax is as follows:

QCOMPILE filename[*.ext*] [-O*file2*] [-D*macro*=*string*] [-L*file3*] [-S] [-W]

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

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

Options can appear in any order. Additional options may be displayed by using -? as an option.

-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 MUCM configuration file for more than 1 port of the same MUCM module. The *macro* portion of the -D option is the string inside of the MUCM 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.MUCM, 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:

```
QCOMPILE 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 MUCM then you can use the -L option. Most users will not have a use for this option.

-S option

The -S option is for generating a list of the location of each declared variable. The variables are located in the 6x file areas of the MUCM. Application 1 variables are located in file 384. Application 2 variables are located in file 640.

The variable list is displayed as a table, sorted by the order that the variables were declared. The table has columns that show the varible’s byte address, register address, type, number of elements (if applicable), number of bytes, and what thread they were assigned to.

-W option

The -W option disables warnings that indicate possible trouble but do not prohibit the program from successfully compiling. Mostly used for disabling the warning "Program is too large" warnings on large applications that use the optional large flash for application 2.

Compiler Errors

When the MUCM 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 .MUCM 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 MUCM 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 MUCM.

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 <Compiler error chapter> - <Compiler error chapter>.

8
Downloading Compiled Code

QLOAD.EXE

The program QLOAD.EXE is a WIN32 console program that will download compiled applications into an MUCM via Modbus serial.

```
Niobrara MUCM Downloader 28Sep99  Copyright (c) 1998 NR&D.
Usage is:
  QLOAD <channel> <file>[.QCC] <port> [<drop>] switches
  where <channel> is the UCM application (1 or 2), <port>
  is a PC COM port name (for example COM1:)

Switches:
  -A        Enable autostart for download program.
  -B        Use Modbus ASCII (RTU is the default).
  -C<count> Allow serial retries (0 is default)
  -E        Erase flash only.  No download.
  -M<baud>,[E|O|N],[7|8],[1|2]   Set port mode (COM).
  -N        Don't erase flash before loading.
  -R<reg>   Use output register reg for run control.
  -S<reg>   Move status registers to reg and reg+1
  -T<time>  Reply timeout in seconds.
  -?        Display this help message.
```

Examples:

The file MRPC.QCC is included with the MUCM files. This precompiled application provides routing and protocol translations for the MUCM to make it act like an NR&D SPE4 with only two ports (refer to www.niobrara.com, and find the QRPC manual for more information). This application is to be loaded into Application 1 and it also prohibits any Applications from being loaded into Application 2.

To load the file from COM1 of a PC with a MU1 serial cable to one of the serial ports on the MUCM-LE simply do the following:

1 Move the switches for Applications 1 and 2 to HALT.

2 Connect the MU1 to the PC's com1 port and one of the serial ports on the MUCM.

3 In Windows, go to:
    >Start,Programs,Niobrara,RPC,RPCload

4 Click Browse, and find C:\Niobrara\firmware\mrpc.qrc

5 Use Modbus Serial, and set the defaults. Then click Start Download.

This will start loading the application and set it to automatically start on power cycles. MRPC is a large application and takes several minutes to download via Modbus RTU serial at 9600 baud.

9 Connector Pinouts

RS-232 port (Screw Terminal)

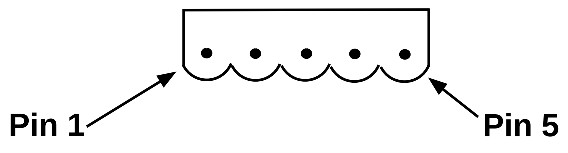


Figure 9-1 Port 1 and Post 2 Screw Terminal

Table 9-1 RS-232 Pinout

Pin	Function	Notes
1	TX	Transmit
2	RX	Receive
3	SG	Signal Ground
4	RTS	Push to Talk Request To Send
5	CTS	CTS must be high to transmit

RS-485 port (Screw Terminal)

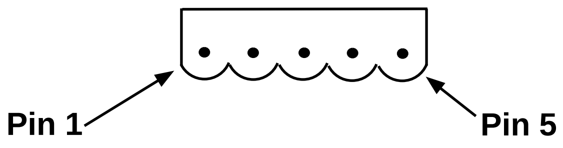


Figure 9-2 Port 1 and Port 2 Screw Terminal

Table 9-2 RS-485 Pinout

Pin	Function	Notes
1	Tx+	Output from MUCM
2	Tx-	Output from MUCM
3	Rx+	Input to MUCM
4	Rx-	Input to MUCM
5	SG	Signal Ground

For 2-wire RS-485 applications, Rx+ and Tx+ must be tied together outside the MUCM, and Rx- and Tx- must also be tied together outside the MUCM.

10 Recommended Cabling

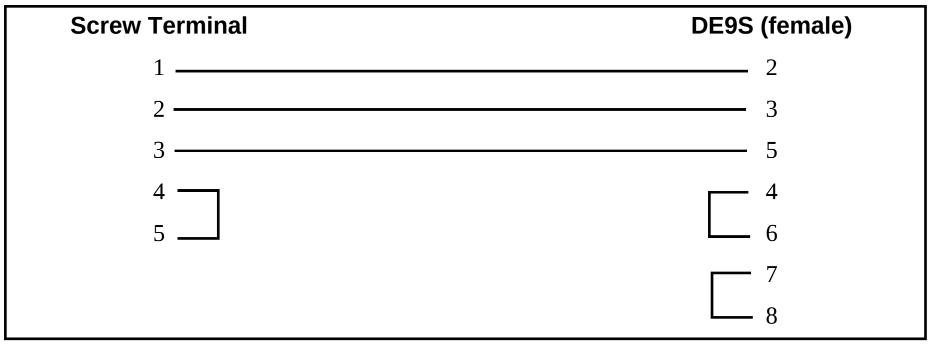
Cabling required to configure an MUCM

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

MUCM RS-232 to personal computer cabling

A connection to the RS-232 port of the PC may be made to the RS-232 port of the module.

MUCM RS-232 to RS-232 PC DCE Port (9-pin) (MU1 Cable)

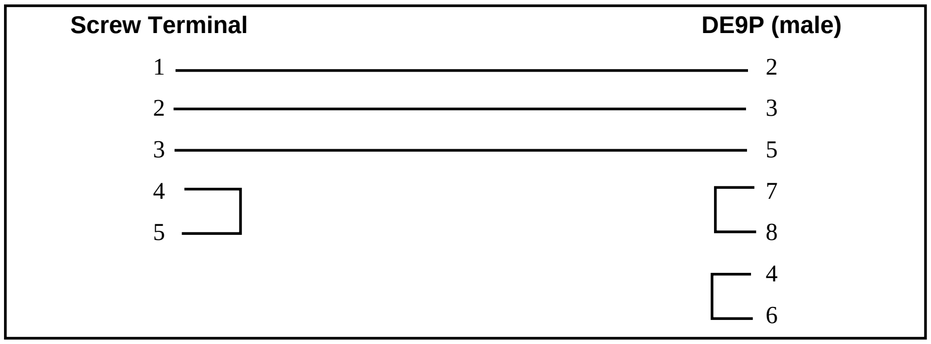


The Niobrara MU1 cable may be used for connecting the MUCM to a personal computer.

MUCM RS-232 to Modicon Quantum PLC port(9-pin) (MU2)

The Niobrara MU2 cable may be used to connect the MUCM RS-232 port to a Modicon Quantum PLC port.

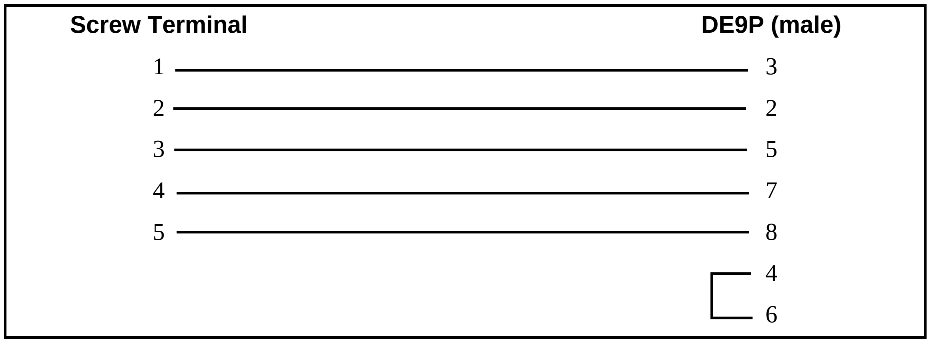
MUCM RS-232 to Quantum PLC (9-pin) (MU2 Cable)



MUCM RS-232 to 9-pin DTE

The Niobrara MU3 cable may be used to connect the MUCM RS-232 port to a 9-pin DCE device. This cable gives the MUCM a standard PC type 9-pin male connector. The MU3 may be used in conjunction with the MU1 to connect two MUCM type screw terminal serial ports together. The MU3 may be used with a Niobrara SC902 cable to connect an MUCM type screw terminal RS-232 serial port to a SY/MAX type RS-422 port.

MUCM RS-232 to RS-232 DTE Port (9-pin) (MU3 Cable)

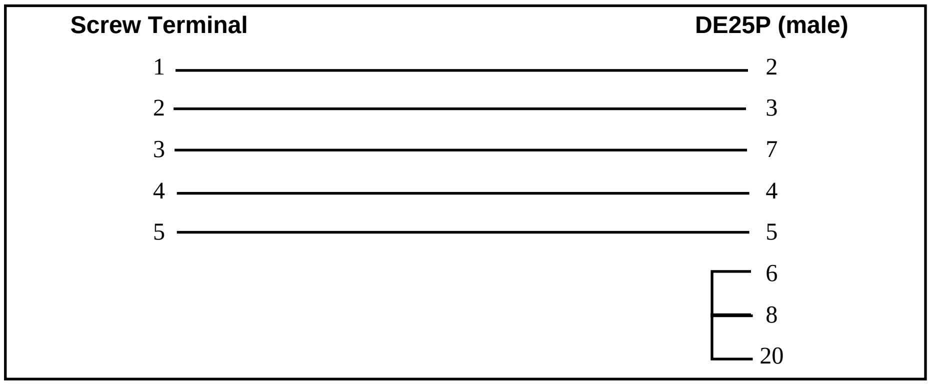


The Niobrara MU3 cable may be used for providing the MUCM with a 9-pin port that acts like a personal computer's serial port.

MUCM RS-232 to 25-pin DTE

The Niobrara MU4 cable may be used to connect the MUCM RS-232 port to a 25-pin DCE device such as a modem or a Cutler-Hammer MINT II.

MUCM RS-232 to RS-232 DTE Port (25-pin) (MU4 Cable)

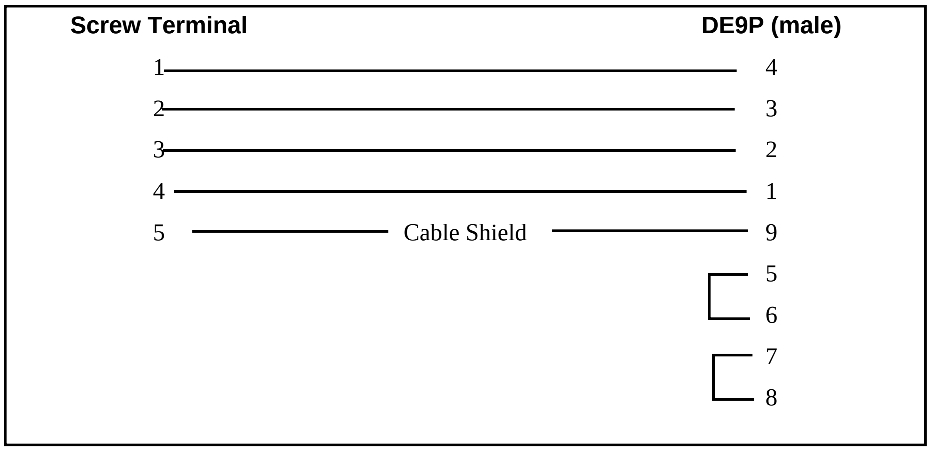


The Niobrara MU4 cable may be used to connect the MUCM to a modem.

MUCM RS-485/422 to SY/MAX 9-pin Port

When a non-isolated connection can be made between the MUCM and a SY/MAX pinout port, the following cable may be used.

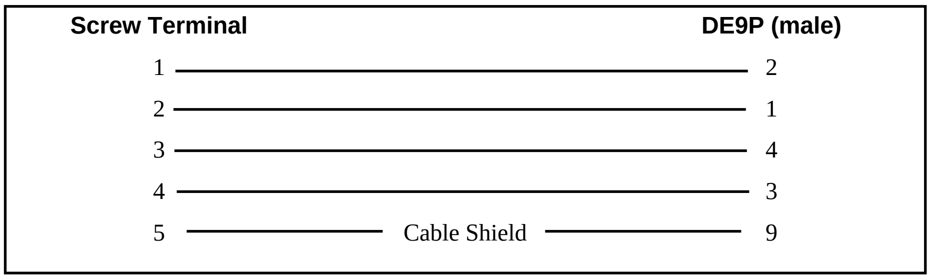
MUCM RS-422/485 to SY/MAX pinout 9-pin port (MU7 Cable)



MUCM RS-485/422 as SY/MAX 9-pin Port

When it is desirable to connect an MUCM as a SY/MAX device, a Niobrara MU8 cable may be used.

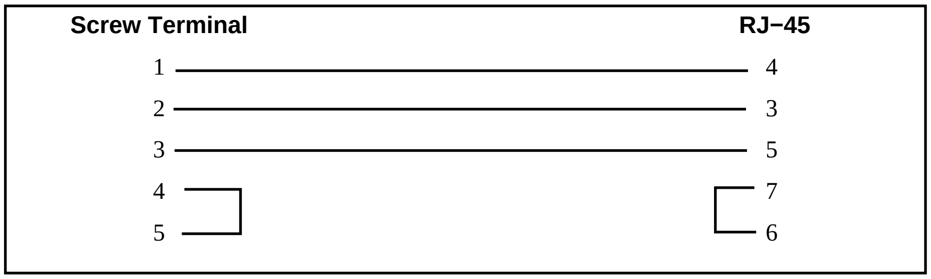
MUCM RS-422/485 as SY/MAX pinout 9-pin port (MU8 Cable)



MUCM RS-232 to Modicon RJ-45 RS-232 (MU9)

The Niobrara MU9 cable may be used to connect the MUCM RS-232 port to a Modicon-style RJ-45 RS-232 port, such as on the MUCM.

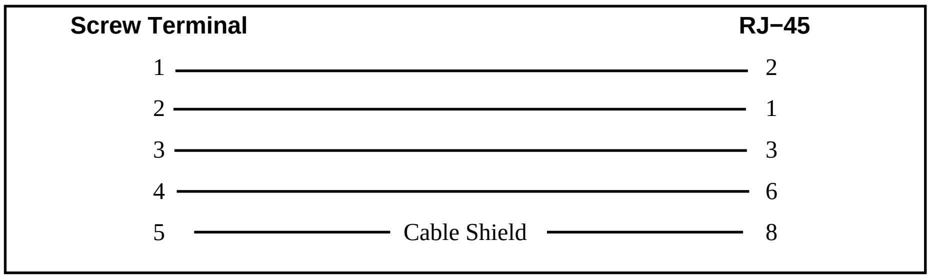
MUCM RS-232 to Modicon RJ-45 (MU9 Cable)



MUCM RS-485/422 to Modicon RJ-45 RS-485 Port (MU10)

A Niobrara MU10 cable may be used to connect an MUCM to a Modicon-style RJ-45 RS-485 port.

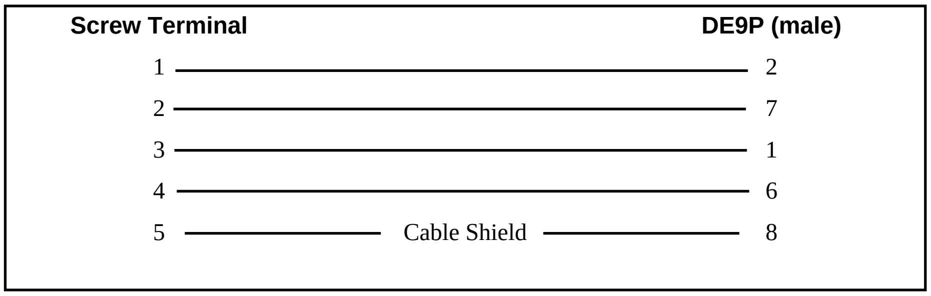
MUCM RS-422/485 to Modicon RJ-45 RS-485 Port (MU10 Cable)



MUCM RS-485/422 to Modicon 9-Pin RS-485 Port (MU11)

A Niobrara MU11 cable may be used to connect an MUCM to a Modicon-style 9-pin RS-485 port.

MUCM RS-422/485 to Modicon 9-pin RS-485 Port (MU11 Cable)



Isolated Cabling to SY/MAX Port

The Niobrara DDC2I Isolated RS-232<>RS-422/485 converter provides an optically isolated connection from the MUCM to a SY/MAX pinout device. A Niobrara MU10 cable is used to connect the RS-232 port on the MUCM to the RJ45 port on the DDC2I. A Niobrara DC1 cable is used to connect the 9-pin RS-422 port on the DDC2I to the SY/MAX device. The DIP switches on the DDC2I should be set for 4-Wire, Bias, and Termination. The DDC2I must be powered by a power supply.

The Niobrara MU3 and SC902 cables may also be used together to provide a non-isolated connection between the MUCM and a SY/MAX port.

MUCM RS-232 to DDC2I RJ-45 (MU10 Cable)

Screw Terminal	RJ−45
1	3
2	4
3	5
4	6
5	7

Appendix A
Downloading New Firmware

As new features and fixes are added to the MUCM, it may become necessary for the user to upgrade their firmware to take advantage of these changes. The MUCM’s operating firmware is stored in a FLASH memory and may be downloaded through serial Port 1 using an RS-232 cable and a special program called FWLOAD.EXE

1 Locate the slide switch on the front of the module and move the switch to the LOAD (right) position.

2 Connect the personal computer to Port 1 of the MUCM with an MU1 cable.

3 On the MUCM103 set the Port 1 mode selection switch to RS-232.

4 Run the program FWLOAD.EXE in the following manner:
  In Windows, Click Start,Programs,Niobrara,MUCM,FWLOAD MUCM firmware
  Choose the appropriate COM: port, then click Start Download.

5 After the completion of the download, the program will end.

6 Move the slide switch back to run, and it should be ready for service. It may be necessary to download the Applications before returning the unit to service.

Note: If at any point during this procedure the download fails, it will be necessary to reboot the MUCM before trying again.

It is now possible to load firmware through any serial port. This will eliminate the need for the user to move the firmware Load/Run switch. A special version of the new firmware will be available that can be installed as an Application. To accomplish this, simply QLOAD the firmware into the module. This will replace any current Application. The firmware Application will then write the new firmware to flash, and then self-terminate. Reload the original Application into the MUCM, and return the unit to service.

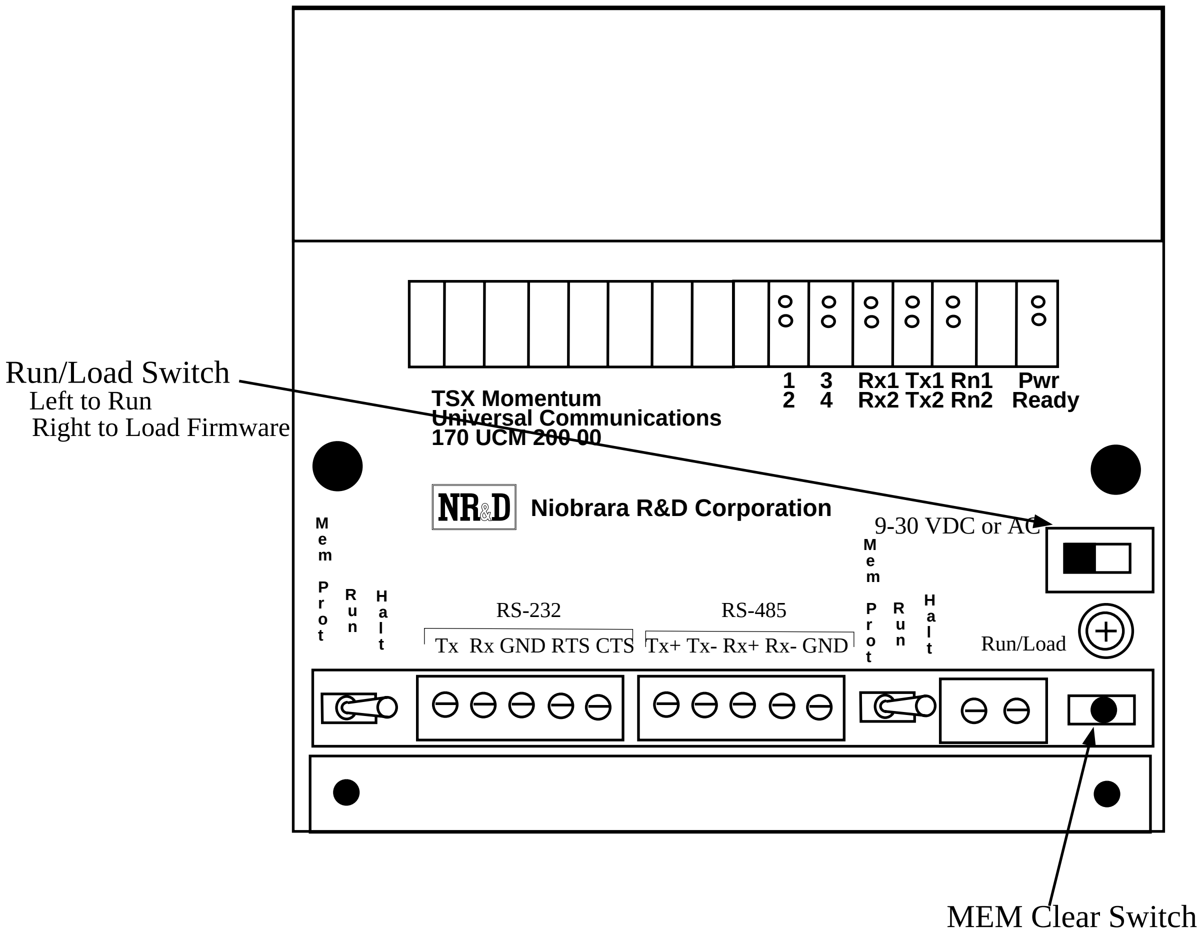


Figure A-1 MUCM102 Front Panel

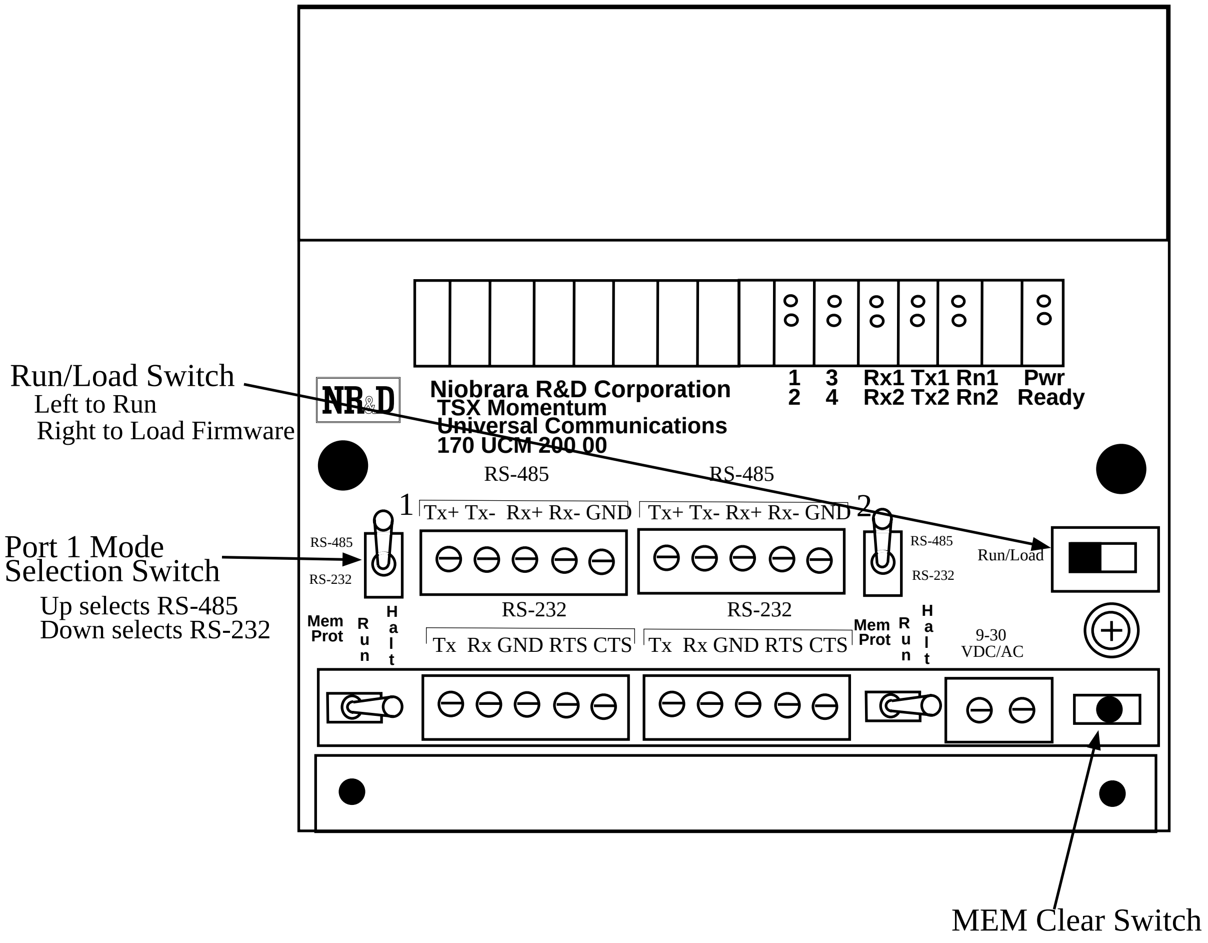


Figure A-2 MUCM103 Front Panel

Appendix B
ASCII Table

Table B-1 lists the common ASCII Characters and their decimal and hex values.

Table B-1 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 C
MUCM Language Syntax

STATEMENTS

ON RECEIVE PORT <port number> <message description> GOTO <label>

ON RECEIVE PORT <port number> <message description> RETURN

ON CHANGE <variable> GOTO <label>

ON CHANGE <variable> RETURN

ON CHANGE <variable> & <expr> GOTO <label>

ON CHANGE <variable> & <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 <variable> = <expr> TO <expr>
one or more statements
NEXT

FOR <variable> = <expr> TO <expr> STEP <expr>
one or more statements
NEXT

FOR <variable> = <expr> DOWNTO <expr>
one or more statements
NEXT

FOR <variable> = <expr> DOWNTO <expr> STEP <expr>
one or more statements
NEXT

<variable> = <expr>

<variable>.<const> = <logical>

DELAY <expr>

STOP

TRANSMIT PORT <port number> <message description>

SET <variable>.<const>

READ FILE <file address> <variable>, <variable>, ...
WRITE FILE <file address> <variable>, <variable>, ...

CLEAR <variable>.<const>

TOGGLE <variable>.<const>

SET PORT <port number> BAUD <const>
SET PORT <port number> CAPITALIZE <const>
SET PORT <port number> DATA <const>
SET DEBUG <const>
SET PORT <port number> MODE <const>
SET PORT <port number> PARITY <const>

SET PORT <port number> 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
```

MIN(<expr>,<expr>)

MAX(<expr>,<expr>)

SWAP(<expr>) {reverses byte order of a word}

LOGICAL EXPRESSIONS <logical> above

Logical Operators:

AND
OR
XOR
NOT (unary)

Logical Functions:

CHANGED(<variable>)
CHANGED(<variable> & <expr>)
<variable>].<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

$ the current index in a message description. Used in check sum calculations.

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 MUCM compiler accepts the above sequences in upper or lower case. These are in addition to the original MUCM 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 MUCM 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
             <variable> in ON RECEIVE to evaluate and place result in RXVARIABLE
             (<expr>) in ON RECEIVE to generate and match string

RAW(<variable>,<expr>)
        |       |
        |       +- width in characters
        +-------- starting register number
BYTE(<expr>)
WORD(<expr>)
RWORD(<expr>)
           |
           +----  expression in transmit
                  <variable> in ON RECEIVE to evaluate and place result in RXVARIABLE
                  (<expr>) in ON RECEIVE to generate and match string
TON( <expr>)
TOFF(<expr>)
        |
        +------ translation number 1..8
```

MUCM 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

MUCM Reserved Word List

The following lists of words are reserved by the MUCM language. These words may not be used for define macro names or labels.

AND	FALSE	OFFSET	TCP
BCD	FILE	ON	THEN
BAUD	FLASH	OR	THREAD
BYTE	FLOAT	OUTPUT	TIMEOUT
CAPITALIZE	FOR	PARITY	TIMER
CASE	FULL	PORT	TO
CHANGE	FUNCTION	PPP	TOGGLE
CHANGED	GOSUB	PPPUSERNAME	TRANSMIT
CLEAR	GOTO	PPPPASSWORD	TRUE
CLOSE	HALF	PPPHANGUP	TRUNC
CONNECT	HEX	RAW	UCM
CRC	IDEC	READ	UDP
CRC16	IF	REALTIME	UNS
CRCAB	INPUT	RECEIVE	UNSIGNED
CRCBOB	LENGTH	RETURN	UNTIL
CRCDNP	LIGHT	REPEAT	VARIABLE
CTS	LISTEN	RNIM	WAIT
DATA	LONG	RTS	WEND
DEBUG	LRC	RTU	WHILE
DEC	LRCW	RWORD	WORD
DECLARE	MAX	SET	WRITE
DEFINE	MIN	SIGNED	XOR
DELAY	MODE	SOCKET	UNTIL
DOWNTO	MOVE	SOCKETSTATE	VARIABLE
DUPLEX	MULTIDROP	STEP	WAIT
ELSE	NAGLE	STOP	WEND
ENDIF	NEXT	STRING	WHILE
ENDFUNC	NOT	SUM	WORD
ENDSWITCH	NONE	SUMW	WRITE
ERASE	OCT	SWAP	XOR
EVEN	ODD	SWITCH	
EXPIRED	OFF	SYMAX	

Appendix D
Modsoft Traffic Cop Configuration

MUCM

The register configuration of the MUCM is governed by its entry in the Traffic Cop description and characteristic file: M1TCOP.SYS. This file is typically located in the \MODSOFT\RUNTIME directory.

NOTE: Pay special notice to the warning about editing this file with editors that do not support line width greater than 255 characters.
DO NOT use MS-DOS EDIT on this file!

The entry for the MUCM follows the form of a Momentum I/O module with 64 bytes of input and 64 bytes of output. The standard entry used in Modsoft V2.4 is shown below:

```
MUCM      ,214,0,64,64,NR&D Universal Comm,0,M00C5,0000,0000,00
12345678901234567890123456789012345678901234567890123
```

The fields are comma separated.

The first 10 characters are the Name of the module.

The 214 is the next entry in the list of available devices. If you are adding this line, choose the next free number in the list that is between 104 and 300.

The 0 in character position 16 indicates that other modules may be inserted in this drop.

The 64 in character positions 18 and 19 set the number of INPUT bytes. A value of 64 provides 32 WORDS of input (3x registers)

The 64 in character positions 21 and 22 set the number of OUTPUT bytes. A value of 64 provides 32 WORDS of output (4x registers)

The next field is the text description (19 characters max.).

The 0 in position 40 determines that the module is a discrete module and may take 0x, 1x, 3x, 4x references.

The module ID is M00C5.

The rest of the entry follows the typical I/O module entry with no additional parameters.

NOTE: If the entry in the Traffic Cop configuration file is altered, all MUCMs in the PLC system will use this entry. Also, special care will be needed during future updates of Modsoft to carry the altered setting to the new revision.

Appendix E
Concept 2.1 (or later) Configuration

The register configuration of each module is controlled by the ModConnect tool in Concept. The ModConnect tool adds devices to Concept that were not originally available when Concept was developed. To accomplish this, the user must copy a .mdc file to the \Concept directory. The .mdc file for the MUCM is available on the website. Download the cncept21.zip file, and extract either the Nrd_w95.mdc, or the Nrd_wnt.mdc file to the \Concept directory.

Next open the ModConnect tool from the Start menu. From the file menu, choose Open Installation File. The appropriate .mdc file should appear in the list of files from which to choose. Click the .mdc file, then click OK. The Select Module dialogue box will appear, allowing the user to choose the MUCM. Next, click Add Module. Concept now has all the information needed to configure a Momentum PLC for the MUCM.

Appendix F
NR&D on the Internet

Niobrara offers product information, firmware and software upgrades, user manuals, and technical support via the Internet at:

http://www.niobrara.com

For technical support questions e-mail: **techsupport@niobrara.com**

For marketing questions e-mail: **marketing@niobrara.com**

For direct anonymous ftp connect to **ftp.niobrara.com**

Appendix G
Memory Map

The memory of the MUCM is divided into separate areas including PLC Rack Input (3x) 16-bit register, PLC Rack Output (4x) 16-bit registers, and four files (6x).

PLC INPUTS (3x)

The MUCM has 60 implemented 3x registers. The first 32 are available to the Momentum PLC through the backplane and are read-only to the PLC. Inputs 1 through 4 default to the status and line number displays for Applications 1 and 2. The INPUT registers are arranged in Modicon bit order with the bits 1-16 in MSB-LSB.

Table G-1 INPUT Registers (3x)

MUCM Register	Description	Bit Description
1	App. 1 Status	This is the default location for these status registers. The location of the Status and Line number registers for Application 1 is controlled by OUTPUT[42]
2	App. 1 Line Number	
3	App. 2 Status	This is the default location for these status registers. The location of the Status and Line number registers for Application 2 is controlled by OUTPUT[43]
4	App. 2 Line Number	
5-32	Rack Inputs	PLC Read only.
33	Reserved	
34	Reserved	
35	Reserved	
36	Read Switch State	bit 16 = Halt 1 (0=off, 1=on) bit 15 = Prot 1 bit 14 = Halt 2 bit 13 = Prot 2 bit 11 = RS-485/RS-232 1 (0=232, 1=485) bit 10 = RS-485/RS-232 2 (0=232, 1=485)
37	Read CTS State	bit 16 = CTS Port 1 (0=off, 1=on) bit 15 = CTS Port 2
38	Ethernet Port MAC Address	MSW
39		
40		LSW
41	App 1, Thread 1	Last line number entered by each Application thread. This is a convenient debugging tool which allows a view of what each thread is doing (thus, what it is waiting for)
42	App 1, Thread 2	
43	App 1, Thread 3	
44	App 1, Thread 4	
45	App 1, Thread 5	
46	App 1, Thread 6	
47	App 1, Thread 7	
48	App 1, Thread 8	
51	App 2, Thread 1	
52	App 2, Thread 2	
53	App 2, Thread 3	
54	App 2, Thread 4	
55	App 2, Thread 5	
56	App 2, Thread 6	
57	App 2, Thread 7	
58	App 2, Thread 8	

PLC OUTPUTS (4x)

The MUCM has 2048 implemented 4x registers. The first 32 are available to the Quantum PLC through the backplane and are read/write from the PLC. The INPUT and OUTPUT registers are arranged in Modicon bit order with the bits 1-16 in MSB-LSB.

Table G-2 OUTPUT Registers (4x)

MUCM Register	Description	Bit Description
1-32	Rack Outputs	MUCM Read only.
33	Default Run Mask	
34	Pointer to Run Mask	
35	Auto-Start Mask	Copied to Run Mask at boot
36	Read Switch State	bit 16 = Halt 1 (0=off, 1=on) bit 15 = Prot 1 bit 14 = Halt 2 bit 13 = Prot 2 bit 11 = Port 1 in RS-485 bit 10 = Port 2 in RS-485
37	Read CTS State	bit 16 = CTS Port 1 (0=off, 1=on) bit 15 = CTS Port 2
38	High Byte = Port 1 control Low Byte = Modbus Drop of Port 1	bit 1 = parity (0=even, 1=none) bits 3-8 = baud rate: 0 = 9600 15 = 19200 16 = 38400
39	High Byte = Port 2 control Low Byte = Modbus Drop of Port 2	bit 1 = parity (0=even, 1=none) bits 3-8 = baud rate: 0 = 9600 15 = 19200 16 = 38400
40	Rack Comms	Bit 16 = Rack Comms Active
41	Modbus Drop of E-net Port	
42	Pointer to Application 1 Status Register	
43	Pointer to Application 2 Status Register	
44	Reserved	
45	Reserved	
46	IP Address (one byte/reg.)	
47	IP Address	
48	IP Address	
49	IP Address	
50	Subnet Mask (one byte/reg)	
51	Subnet Mask	
52	Subnet Mask	
53	Subnet Mask	
54	Default Gate (one byte/reg)	
55	Default Gate	
56	Default Gate	
57	Default Gate	
58 through 61	Reserved	
62	TCP Backstep	
63	Modbus/TCP Server Port Number	
64	Web Server Port Number	
65	Quiet Timeout in seconds	

MUCM Register	Description	Bit Description
66 through 68	Reserved	
69	LED bit map	bit 16 = Light 3 (0=off, 1=on) bit 15 = Light 4 bit 14 = Light 5 (for MUCM, Ready LED) bit 13 = Light 6 bit 12 = Light 7 (fpr MUCM, App. 1 Run) bit 11 = Light 8 (for MUCM, App. 2 Run) bit 10 = Light 9 (for MUCM, Light 1) bit 9 = Light 10 (for MUCM, Light 2) bit 8 = Fault LED bit 7 = Active LED bit 6 = Ready LED bit 5 = Run LED bit 4 = App. 1 Run bit 3 = App. 2 Run bit 2 = Light 1 bit 1 = Light 2
70	Real-Time Clock	bit 1 = RTC support (1=support of RTC) bit 2 = RTC chip present (1=chip present) bit 3 = Status (0=reliable, 1=unreliable)
71	RTC Seconds	
72	RTC Minutes	
73	RTC Hours	
74	RTC Day of Month	
75	RTC Month	
76	RTC Year	
77	RTC Day of Week	
78	Register 1 of UNIX time	UNIX time represents the number of seconds since the beginning of 1970.
79	Register 2 of UNIX time	

Index