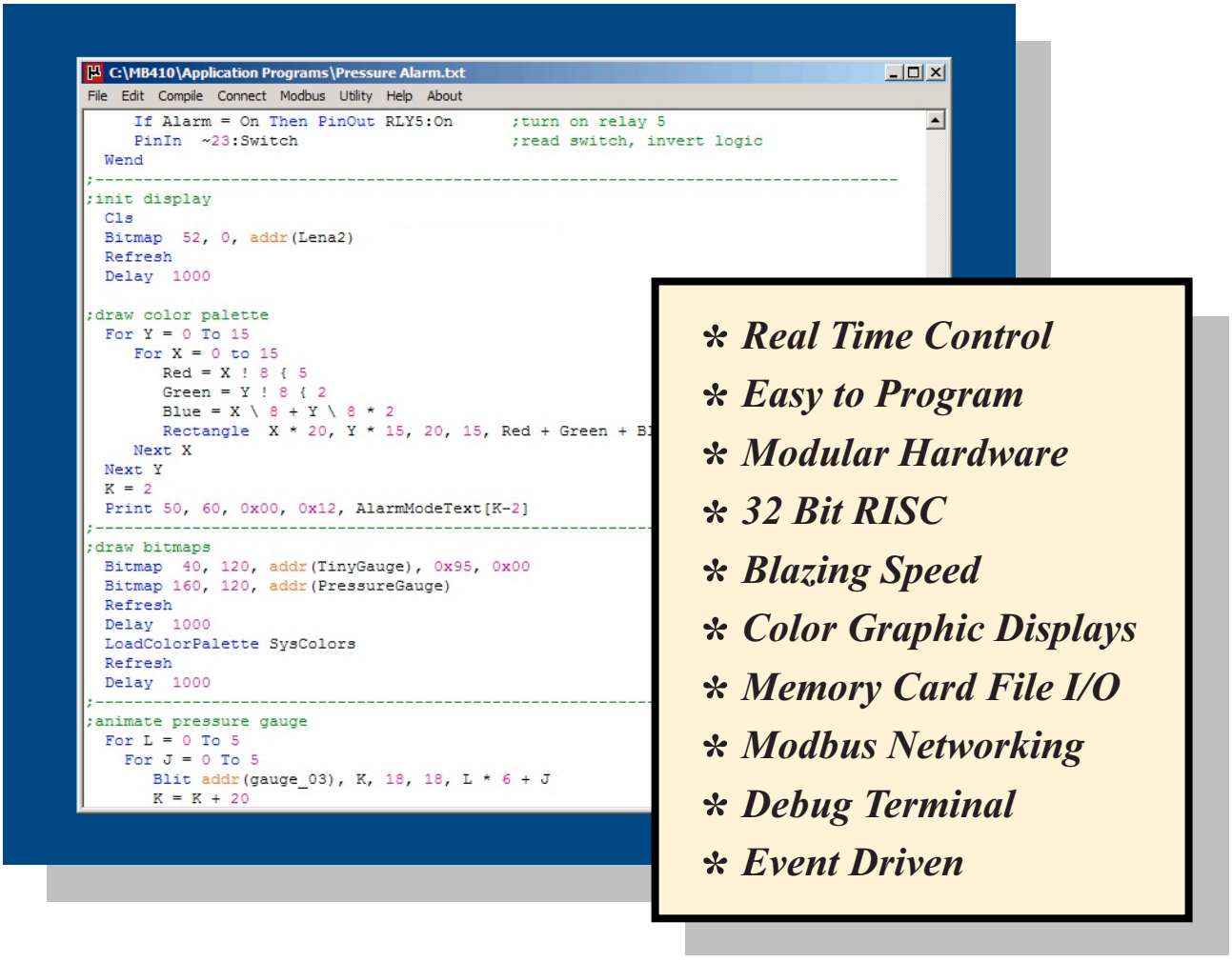
MICROVA
electronics & automation

Microva BASIC Programming

Reference Manual For

Compiler Version 2.10

(Supports Firmware Versions 2.5, 3.4, 4.6 and 5.2)



Microva BASIC Commands

Commands To Open/Close I/O Ports

Open I2CPort	open I2C port
Open SerialPort1	open serial port 1
Open SerialPort2	open serial port 2
Open SpiPort	open SPI port
OpenSDHC	mount SDHC memory card
Close SerialPort1	close serial port 1
Close SerialPort2	close serial port 2
Close SpiPort	close SPI port

Commands To Create Constants and Variables

Constant	define constants
Dim	define integer or float or string variable or array
Float	define float variable or array
Integer	define integer variable or array
String	define string variable
Table	define integer or float or string ROM table

Commands For Program Control

Exit	exit from a For...Next program loop
For...Next	For...Next program loop
Goto	jump to a label in program
If...Then...Else	If...Then...Else program statement
Program	define program name and firmware version
ResetCPU	resets the CPU
Return	returns from the main program or a subroutine
Select Case	Select Case program statement
Sub	define subroutine
While...Wend	While...Wend program statement

Commands For File I/O

FileAppend	append text to a file
FileDelete	delete a file
FileList	creates a list of files on the memory card
FileLoadPgm	loads a new application program
FileNew	create a new file
FileOpen	open a file
FileRead	read data from a file
FileReadLine	read text from a file
FileRename	rename a file
FileResize	change the file size
FileWrite	write data to a file

Commands For Serial Port I/O

Modbus1	read or write Modbus packet using serial port 1
Modbus2	read or write Modbus packet using serial port 2
SerialOut1	write to serial port 1
SerialOut2	write to serial port 2

Microva BASIC Commands (continued)

Commands For Pin I/O

PinDir	sets I/O pin direction
PinIn	reads I/O pin input
PinOut	writes I/O pin output
PinType	sets I/O pin type (digital or analog)

Commands To Write To The Display

Blit	block image transfer
Cls	clear the screen
Dot	draw a dot
Ellipse	draw an ellipse
Image	draw an image
Line	draw a line
Print	print to the display
Rectangle	draw a rectangle
Refresh	refresh the display
Triangle	draw a triangle

Commands To Manipulate Strings

Mid	returns a portion of a string
Instr	returns the position of a string within another string
LineOut	converts a string to byte data in an integer array

Commands To Read and Write ROM or RAM

FlashErase	erases a page in Flash ROM
FlashWrite	writes an integer or float to Flash ROM
LoadVarDefs	reloads all variables with their default values
NvVarsLoad	load nonvolatile variables from EERAM
NvVarsSave	save nonvolatile variables to EERAM
SaveComSettings	writes new communication settings to Flash ROM
WriteByte	writes a byte to any CPU address
WriteFlt	writes a float to any CPU address
WriteHalf	writes a "half" integer to any CPU address
WriteInt	writes an integer to any CPU address

Other Commands

Append	appends data files to ROM
Delay	delay in milliseconds
DelayFast	delay in microseconds
Include	includes program files that will be compiled
Sleep	puts CPU into lower power mode

Microva BASIC Functions

Functions That Return Float Data

fabs(float expression)	absolute value of float expression
fisqr(float expression)	inverse square root of float expression
fsqr(float expression)	square root of float expression
cif(integer expression)	convert integer expression to float
cifd(integer expression)	convert integer expression to float directly
csf(string name)	convert string to float
rdf(integer expression)	returns float data from address anywhere in CPU address space

Functions That Return Integer Data

abs(integer expression)	absolute value of integer expression
acos(integer expression)	arccosine of integer expression
addr(name)	address of integer or float or string variable, array or table
ain(integer expression)	read pin analog input value
asc(string name)	integer value of first character in string
asin(integer expression)	arcsine of integer expression
atan(integer expression)	arctangent of integer expression
cfi(float expression)	convert float expression to integer (integer is rounded up)
cfid(float expression)	convert float expression to integer directly (IEEE 754 format)
cfiw(float expression)	convert float expression to integer (integer is not rounded up)
cos(integer expression)	cosine of integer expression
csi(string name)	convert string to integer
i2c(integer expression)	transmit and receive data to/from I2C port
keyin(integer expression)	returns keycode or touch screen value
len(string name)	returns the number of characters in a string
lenx(string name)	returns the width of the text in pixels if the string was printed to display
not(integer expression)	ones complement of integer expression
nva(name)	returns the EERAM address of the nonvolatile variable
rdb(integer expression)	returns byte from address anywhere in CPU address space
rdh(integer expression)	returns "half" integer from address anywhere in CPU address space
rdi(integer expression)	returns integer from address anywhere in CPU address space
rev(integer expression)	bit reversal of integer expression
rnd(integer expression)	returns random integer value
serin(integer expression)	returns a byte from serial port
sin(integer expression)	sine of integer expression
spi(integer expression)	transmit and receive data to/from SPI port
sqr(integer expression)	square root of integer expression
sys(integer expression)	returns system data
tan(integer expression)	tangent of integer expression
timer(integer expression)	returns count from timer 0 or 1
ubound(array name)	upper bound of integer or float or string array or table

Functions That Return String Data

cfs(float expression)	convert float expression to string
cfsf(integer expression)	convert float expression to string and format using decimal format
chr(integer expression)	string character equal to least significant byte of integer expression
cis(integer expression)	convert integer expression to string
cisf(integer expression)	convert integer expression to string and format using decimal format
hex(integer expression)	string equal to hexadecimal value of integer expression
lcase(string name)	convert upper case characters in string to lower case
linein(integer expression)	converts byte data in an integer array to a string
ndf(integer expression)	copy integer expression result to the decimal format system variable
trim(string name)	remove leading/trailing white space
ucase(string name)	convert lower case characters in string to upper case

Introduction

Microva BASIC is a modified version of the standard BASIC programming language with several enhancements which make it better suited for real time control applications. Microva BASIC is a modern, structured, strong typed, event driven, compiled programming language. Microva BASIC is not an interpreted language. There is no byte code, pseudo code, or run time interpreter. Microva BASIC statements are compiled straight to 32 bit RISC machine code. This makes Microva BASIC blazingly fast. For example, a variable can be assigned new data in as little as 2 CPU clock cycles (at 200MHz that's 10 nanoseconds).

In almost all modern operating systems the application program runs in user mode not kernel mode. These operating systems use preemptive multitasking which means the application software is given a time slice window and then preempted as the OS attends to other tasks or allows other applications to run- giving the illusion that tasks are running concurrently. In some cases the OS may need to attend to other tasks for hundreds of milliseconds or even seconds. That's a problem in a real time embedded control environment. For example, an assembly machine might have a safety light curtain. If any part of the operator's body breaks that light field the machine should halt instantly. But if at that exact moment the OS decides to attend to other tasks the operator could be injured. This is why operating systems that run on desktop computers and smart phones are not considered "real time". Microva BASIC runs in kernel mode and uses cooperative multitasking not preemptive multitasking. That means real time control has priority and the application programmer, not the operating system, is in charge of when tasks are switched.

Microva BASIC includes built-in support for Modbus networking. Modbus is one of the most widely used, open source, industrial networking standards (see Microva application note AN130 What is Modbus?). Microva BASIC includes commands for standard serial I/O (RS232 or logic level or RS485) as well as embedded system communication using I2C and SPI protocols. Microva BASIC makes it easy to manipulate bits, bytes, strings, integers and floating point data. There is built in support for nonvolatile variables, i.e. variables that retain their data when the power is turned off. There is built in support for file I/O using SDHC memory cards which is useful for data logging and for loading a new application program. There is built in support for color graphic displays including "widgets" and the full suit of BASIC graphic commands such as Ellipse, Rectangle, Line, Triangle, Image, etc. Microva BASIC makes it easy to format and display real time integer, fixed point or floating point numbers. And there is an extensive library of pre-written software on the Microva website that adds functions such as audio output, motor control, and advanced math.

Microva BASIC is event driven and includes a concept called "change bits" which allows events to occur instantly based on another event occurring such as a sensor input changing or an event timer reaching a specific count. Microva BASIC is optimized for OEMs, including encrypted application files which can be sent to end users in the field without them being able to read, modify or reverse engineer the OEM's application program source code. Microva modular I/O devices make it easy for the OEM to differentiate products with different I/O device options and to replace damaged devices in the field. For example, if an I/O board is damaged, only that one low cost board would need to be replaced rather than a large, expensive board which contains the main CPU and all the I/O in the system.

Microva BASIC Compiler and IDE (Integrated Development Environment)

The compiler and IDE convert the BASIC program statements into 32 bit RISC machine code. The source program file can be any unformatted ANSI, ASCII or UTF-8 (Unicode) text file with file extension "txt", "bas" or "lib". Traditionally, BASIC application programs use the "bas" file extension and pre-written software in a library uses the "lib" file extension. Use the Include command to add library software to the main application program. The ANSI source text file can be used for all Latin based Western languages. ANSI (also called ISO-8859-1) is equivalent to the first 256 characters of unicode (UTF-8). See the ANSI character code table in the appendix. The IDE has a built in text editor with keyword color highlighted text which uses standard Windows text editing commands such as cut, copy, paste, screen refresh, find and replace, etc. However, any text editor could be used to create the source file provided it can save the file as unformatted text. Use screen refresh ("F5") to show the keyword colored text when pasting a paragraph of text from another file into the source program. Function keys "F6" and "F7" can be used to comment out or uncomment a block of selected text in the application program.

To compile the program just open the source file and click on "Compile". The compiler will highlight the first error encountered in the source program. Correct the error and recompile and repeat until all errors are removed. When the source file compiles with no errors there will be a binary data file saved in the same location as the source program with the same file name but with file extension "MVO" (the Microva output file). The MVO file is the compiled output program and is the only file which must be downloaded to the hardware or loaded from the memory card in order to run the application. All file dependencies such as library software, resource files, appended images or data files, etc are included in the MVO output file. The MVO output files are encoded using an SHA type encryption algorithm. Even the exact same source program recompiled will generate a completely different MVO output file. For OEMs that means the application program source code remains proprietary since only the MVO file (not the source text file) is needed to update devices in the field. Devices can be programmed directly over the USB or over the Modbus network or by using an SDHC micro memory card. The Microva BASIC Compiler IDE includes an easy to use terminal interface (the "connect screen") for communication with one or more Microva devices connected to the computer for debugging and for programming devices. The IDE also includes a Modbus RTU network interface for directly reading and writing to Modbus devices on the network regardless if the device is made by Microva or another manufacturer.

How To Enter Program Mode

Microva BASIC devices are always in one of two modes: Program Mode (PGM) or Run Mode (RUN). RUN mode is the normal operating mode when the device is running the application program. In PGM mode the device is "off line" and is ready to accept a new application program, i.e. a new MVO file as described above. In PGM mode all I/O pins will be in the reset state. See the device datasheet for the I/O pin configuration diagram. Devices can be programmed directly from an SDHC memory card if the device has a memory card connector or from the Microva BASIC IDE "connect screen" (shown below). The connect screen is used to load new programs into the device and to debug the application program. In order to download the MVO file into the hardware using the connect screen, first the device must be connected to the computer. There are three ways that devices connect to the computer:

1. Some devices connect directly to the computer over USB. These devices can usually be powered over the USB.
2. Devices with Modbus I/O connect using an RS485-to-USB interface such as Microva model MB100. These devices require a separate power supply which can power one or more nodes on the Modbus network.
3. Devices with logic level serial I/O connect using a serial-to-USB interface such as Microva model MB103. These devices can usually be powered over the USB or with a separate power supply.

The IDE connect screen will not open until the Microva interface or device is recognized.

For devices with no display, to enter PGM mode: With the power off, press and hold the PGM switch on the PCB, then turn on the power and release the switch. The red LED on the PCB will be lit to indicate that the unit is in PGM mode. Note: If the PGM switch continues to be held on for more than a few seconds the device will return to RUN mode and execute the internal application program. That allows the node number to be changed by holding the PGM switch and counting the number of LED blinks- see the device data sheet for details.

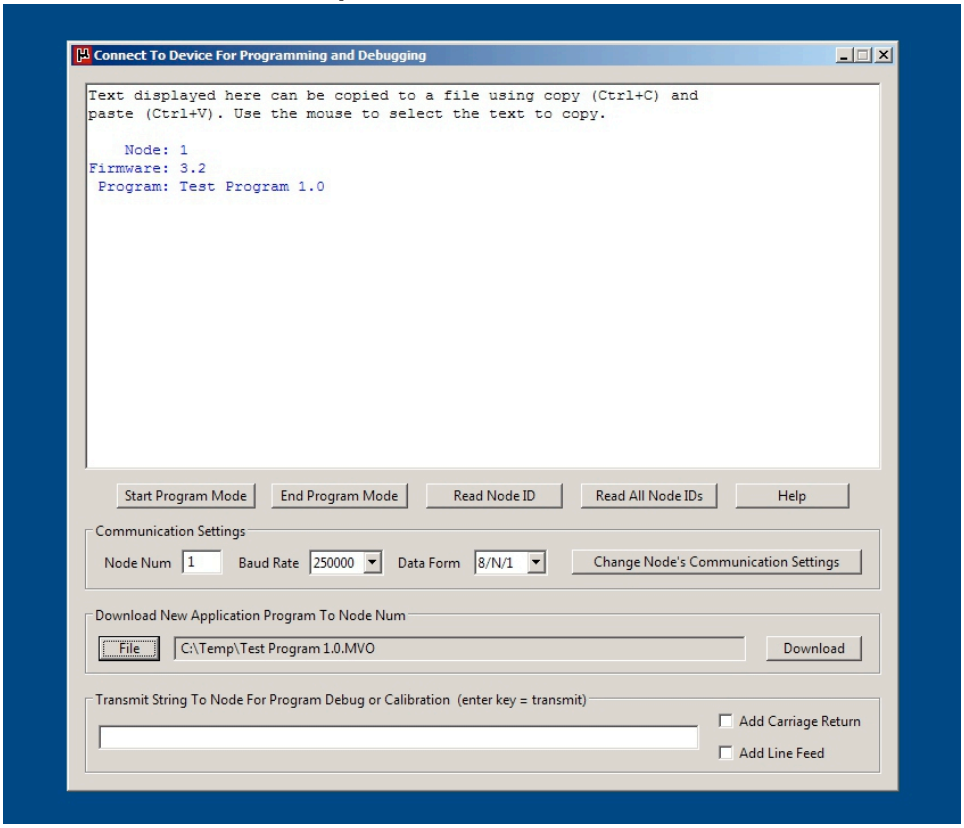
For devices with a display and keypad, to enter PGM mode: Hold the DOWN arrow key on the keypad and turn on the power (or press/release the reset switch if the device has a reset switch).

For devices with a display and touch screen, to enter PGM mode: Press anywhere on the touch screen and turn on the power (or press/release the reset switch if the device has a reset switch).

How to enter PGM mode over the Modbus: All Microva Modbus slave nodes on the network can be forced into PGM mode at the same time. Click on "Start Program Mode" on the connect screen (shown below). This command is sent to Modbus address 0, the global node address, so all slave nodes on the network will respond. However, the Modbus master node will not change to PGM mode using this method. Therefore, the master node must be disconnected or forced into PGM mode manually as described above. Otherwise the master will continue to issue commands to the slaves which will cause communication errors when the slave nodes are programmed. For devices with no display, when in PGM mode the red LED will be on. If the device has a display both the firmware version and the application program name will be displayed while in PGM mode. Normally, in RUN mode, when the device is configured as a Modbus slave node, the red LED will be off and the green LED will blink each time a Modbus packet is received from the Modbus master. However, the red/green LEDs are under control of the application program in RUN mode so it depends on the application.

Use ResetCPU Command: The ResetCPU command can be used in the application program to force the device into PGM mode as described on the command page.

Microva BASIC Compiler IDE Connect Screen



How To Load A New Program From The Computer

In order to load a new program file (MVO file) to the device, first make sure the device is in PGM mode as explained above, then select the file to download on the connect screen, then click on "Download". The communication settings in the device and the communication settings shown on the connect screen must be the same. To check if the device is communicating properly, click on "Read Node ID". If the communication settings are correct the node identification string, which consists of the node number, firmware version and application program name in the device, will be displayed (as shown on the previous page). When connected to multiple devices on a Modbus network, all the node identifications can be read automatically one-by-one using "Read All Node IDs" (all devices must have unique node numbers). The node IDs can be read in both PGM mode and RUN mode. When the device is forced into PGM mode by holding the switch at power on or if the device is powered on but does not have a valid application program (which can happen for example by loading a program with the wrong firmware version) then the device's internal communication settings are ignored and the communication settings are temporarily set to the default configuration which is Node Number = 1, Data Format = 8/N/1 and Baud Rate = 250K or 1M (see Table 1). However, when the device enters PGM mode by clicking on "Start Program Mode" on the connect screen then the Node Number, Baud Rate and Data Format retain the same values they had while in RUN mode (set by the application program) and therefore the communication settings on the connect screen must be changed to match these RUN mode settings.

How To Load A New Program From A Memory Card

If the device includes a memory card connector then the MVO file can be loaded directly from the memory card- the computer is not needed. The memory card must be an SDHC micro memory card formatted with FAT32 (which is standard for SDHC cards). SDHC memory cards range from 4GB to 32GB and have "SDHC" printed on them. See the device data sheet for the exact procedure.

How To Debug Programs

The connect screen is also used for debugging the application program. In the application program, when debugging, set the device's serial port to mode "SerialIO". It is then easy to transmit human readable data to the terminal window using the SerialOut command. Code can be added to the application to pause the program until a key is pressed on the computer. The program could then proceed to the next pause and wait for another input. Use the "Transmit String" on the bottom part of the connect screen to send ANSI character data to the device. See the SerIn function description for example code that pauses the program until a character is received then transmits the value of a variable to the connect screen terminal window for debugging the application program.

How To Change The Device's Communication (COM) Settings

The communication settings shown on the connect screen are what the computer uses to communicate to the device. New devices will have the COM settings set at the factory to: Node Num = 1, Data Form = 8/N/1 and Baud Rate = 250K or 1M (see table 1) . Click on "Change Node's Communication Settings" in order to change the internal communication settings in the device. The device must be in PGM mode in order to change these settings. The communication settings are stored in the Flash ROM in the device and are retained even when a new application program is loaded. See the SaveComSettings command for a detailed description of the device's internal COM settings. Typically, the device's internal COM settings are used to open the serial port in the Reset subroutine in the application program. As mentioned above, these internal COM settings are overridden when the device is forced into PGM by holding the switch at power on. For Microva devices without a display, the COM settings can also be changed just using the PGM switch on the PCB and counting the number of blinks of the green LED (see the device data sheet for a detailed explanation). This allows the installer to change the COM settings in the field even when there is no computer available.

How To Change The COM Settings On Several Devices At The Same Time

Sometimes it's necessary to change all the COM settings for multiple devices all at once rather than change the settings for each device one-by-one. For example, an OEM may need 40 units of a relay output board all with the same node numbers, baud rates and data forms. Rather than connecting these 40 boards one-by-one to the computer to change the COM settings, it is possible to connect them all and then change all the COM settings all at once instantly. To do that first make sure all the devices have the same COM settings (new devices from the factory will have the same settings as described above). Connect all the units on the network to the computer using the RS485-to-USB interface such as Microva model MB100. Then connect the power supply to the network and turn on the power. All the units will be in RUN mode so the red/green LEDs should all be off. Click "Start Program Mode" on the Microva BASIC IDE connect screen to put all units into PGM mode. All the red LEDs should now be on. Next click on "Change Node's Communication Settings" and set the COM settings to the new values desired. Click "Change Settings" to write the new COM settings to all 40 devices at the same time. The response will be "Error: No response from node, etc..." that's because when a command is sent to a device the device is expected to respond with an acknowledgment. However, all the devices have the same node number so they will all acknowledge at the same time causing a response error. But the new COM settings were correctly written into all the devices, it is only the acknowledge response from the nodes that causes the apparent error. Once the COM settings in the device are changed, to communicate with the device, the connect screen COM settings must also be changed.

Microva BASIC Variables

Microva BASIC supports three variable types: Integer, Float and String. An Integer variable holds a 32 bit signed binary value with a range from -2,147,483,648 to +2,147,483,647. A Float variable holds a 32 bit IEEE 754 format "single" floating point value with a range of approximately +/-1.0e-38 to +/-1.0e+38. A String variable holds text characters. All Microva BASIC variables are assigned to fixed locations in RAM memory when the program is compiled. A variable's location in memory never changes while the program is running. All Microva BASIC variables are "public" which means any variable can be read or written within any subroutine. Microva BASIC does not allow "private" variables, i.e. variables which are assigned within a subroutine and which can only be used in that subroutine. Therefore, all variables must be defined at the start of the program or between subroutines. The ADDR function can be used to get the memory address of a variable (called a pointer) if needed.

When working with numbers with an absolute value between 0 and a few million it is generally best to work with integer variables. When working with numbers that are extremely small or extremely large it is better to use float variables. That's because integer math is both more accurate and faster than floating point math. The reason is because both integer and float values must be converted into their boolean equivalents. A 32 bit integer variable reserves 31 bits to store the number plus 1 bit for the sign whereas a 32 bit float variable reserves 24 bits for the number plus 1 bit for the sign and 8 bits for the exponent (that appears to exceed 32 bits but doesn't really as explained in the IEEE 754 specification). Also, every decimal integer value has an exact equivalent binary value. However, not every floating point decimal value has an exact translation to binary in the same way that not every fraction can be represented in decimal (like 1/3). For example, the binary float equivalent value of decimal 0.1 is 0.09999999. Pretty close but not exact, which is why float variables should be compared using greater than or less than ranges rather than equals. For example, the float expression 30.0 x 0.1 may not exactly equal 3.0 but the integer expression 30 / 10 is exactly 3. On the other hand the integer expression 3 /10 will return 0 but the float expression 3.0 / 10.0 will return 0.3 (or very close). But on the other hand Microva BASIC makes it easy to display integer values using fixed point decimal notation (see the NDF function). So the integer 30 could be displayed as "3.0" in fixed point even though the variable holds 30. Then the result of the expression 30 / 10 would display as "0.3" so we're back to the case that integer math is both more accurate and faster than float math (since most integer math executes in a single CPU clock cycle whereas float math requires many CPU clock cycles).

String variables hold ANSI text characters. See the table of ANSI characters in the appendix. Use double quotes to define text such as: MyText = "Hello World". Each individual string character is stored in memory as a single byte. Each string variable can hold from 0 to 255 characters. Variables are created using the Integer, Float, String or Dim commands. The Dim statement is the traditional way to create variables in BASIC, however, the other methods allow the variable to be assigned additional properties when the variable is created such as a Modbus address. Variables can also be created as arrays. An array is just a group of variables with the same name in which the individual "elements" in the array are referenced by index rather than by individual name. For example, to save 20 temperature data values it would be easier to create an array called "Temp" with 20 elements like: Dim Temp(20) As Integer rather than create 20 separate variables (Temp1, Temp2, Temp3, etc). The array elements are referenced by index such as Temp[3] which has the advantage that the index can itself be a variable or expression such as Temp[J + 5]. Arrays are indexed starting with 0, i.e. the first element in the array has index 0 and the last element in the array has index = array size - 1. When defining the array use parenthesis to indicate the array size as shown above. Use square brackets to refer to an individual element in the array (ex. Temp[J+1]). In this manual a stand alone variable like Temp1 is called a direct variable. A variable which is part of an array like Temp[3] or Temp[J + 5] is called an indirect variable.

Some Microva BASIC commands require parameters that are simple direct or indirect variables. Other commands can accept parameters that are integer or float expressions. An expression is considered to be any statement that is not a simple direct or indirect variable name. For example, Temp1 is not an expression but Temp1 + 5 is. MyArray[J] is not considered an expression but MyArray[J + 5] is. Microva BASIC is a "strong typed" programming language meaning the compiler does not allow mixing variable types. For example, the statement MyInteger = 5.0 will generate a compiler error because 5.0 is considered a float value and the variable is an integer. Likewise, MyString = 5 will generate an error because 5 is a number (an integer) not a string. Microva BASIC includes many functions for converting between variable types. For example, function CFI is used to **C**onvert a **F**loat to **I**nteger and CIS is used to **C**onvert an **I**nteger to **S**tring. Therefore, MyInteger = cfi(5.0) and MyString = cis(5) are valid statements. Of course both statements are pointless since they could have been more easily written: MyInteger = 5 and MyString = "5".

ROM, RAM, EERAM, Tables and Constants

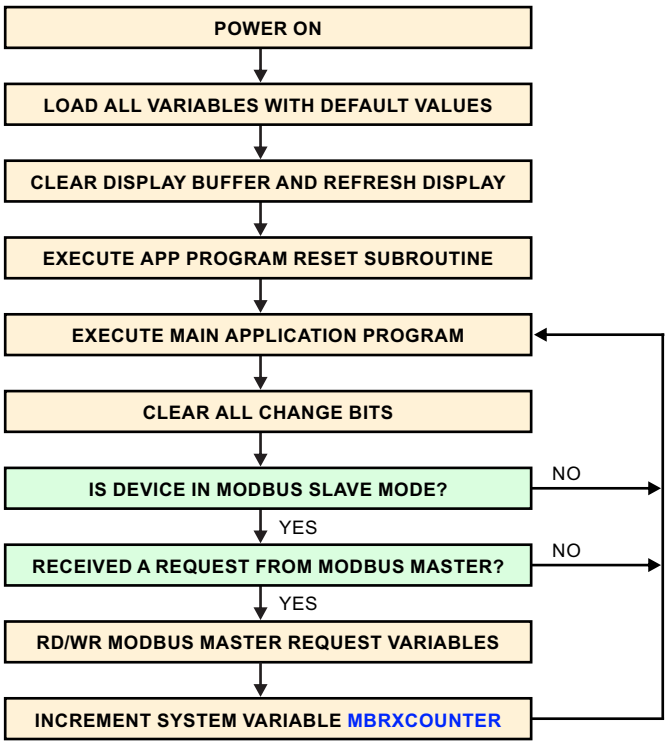
All variables exist in RAM. RAM is the fastest type of memory for both reading and writing and with no limitations on the number of times it can be read/written. The application program resides in Flash ROM (called ROM in this manual). ROM can be read nearly as fast as RAM but is written hundreds of times slower than RAM. ROM can be read an unlimited number of times but can "only" be written approximately 20,000 times. Writing to the same ROM location more than this number of times can cause ROM damage. When the power is turned off the data stored in RAM is lost, however, ROM data is retained. All data is read and written from or to memory in "little endian" data format. Microva BASIC supports nonvolatile variables. These are variables that exist in RAM but retain their data when the power is turned off. This is accomplished with the help of a third type of memory known as EERAM. EERAM is a cross between RAM and ROM. Microva BASIC devices do not always include EERAM due to the relatively high cost of this type of memory. Check the device data sheet. See the NvVarsLoad and NvVarsSave commands for more information.

A constant is just a data value that is given a name to make it more convenient to use. Constants are declared using the Constant command. For example, C = Pi * D is better than C = 3.1415927 * D. A constant can be used anywhere a regular number can be used. There are several predefined constants (such as Pi). See the Constant command page for a list of predefined constants. A table is an array of integer or float or string values that exist in ROM. Table data is referenced in exactly the same way as normal RAM variable arrays (ex. Data = MyTable[J]). A string table is actually an array of strings. String tables are useful for holding data such as engineering unit names (inch, foot, meter, gram, pound, etc). See the Table command for more information.

The Microva BASIC Kernel

The Microva BASIC Kernel firmware is responsible for handling low level I/O tasks associated with the hardware. For example, the BASIC statement SerialOut1 "Hello" writes a string to serial port 1. The compiler does not write all the machine code necessary to write the string to the serial port each time the "SerialOut" command is used. For some commands that would require hundreds of bytes of ROM code each time the command is used in the application program. A better way is for the compiler to just write the string ("Hello") into a buffer and then call a subroutine in the kernel firmware which actually writes the characters one-by-one to the serial port hardware. Likewise, the BASIC command Print writes to the display regardless of the display resolution or specifications. The kernel firmware handles the task of writing to the display hardware. The firmware is also responsible for handling tasks which run in the background while the normal application program is running. These background tasks, called "interrupts", are enabled and disabled in the application program. Interrupts are described in more detail below.

At power on the firmware loads all variables with their default values specified when the variable was created. If the device has a display the display is cleared. If the application program includes a Reset subroutine, that subroutine is executed. Then the main application program is executed. Then all "change bits" are cleared (change bits are described below). Then, if the device is configured as a Modbus slave, the Modbus master request is parsed and executed. If the request was received and executed with no errors then the internal counter MbRxCounter is incremented. MbRxCounter can be used in the application program to blink the green LED each time a valid Modbus request is received from the master (see the Open SerialPort command). Then the firmware loops back and begins a new program loop by executing the main application program again, then clearing the change bits, etc.- as shown in the flow chart to the right. This program loop continues until the power is turned off. Therefore, the application program is executed continuously, over-and-over, thousands or even tens or hundreds of thousands of times every second.



Change Bits

Microva BASIC allows statements like this:

```
If Timer50ms Changes Then PinIn ~PB5:InputSwitch
If InputSwitch ChangesTo On Then Count = Count + 1
```

This example increments a counter each time InputSwitch (connected to pin PB5 using inverted logic) is activated. The qualifier Changes means the input switch will only be read when variable Timer50ms changes, i.e. 20 times per second. That's an easy way to "debounce" a mechanical switch. When pressed, a mechanical switch does not cleanly toggle from off to on, it bounces on-off-on-off-on-off, etc for several milliseconds before settling to on. But in the statement above the switch is only read once every 50ms so the switch bounce does not matter. And 20 times per second is way faster than any person could press and release a switch. The second line in the example above increments the counter only when the switch is activated by using the qualifier ChangesTo. If the second line had been:

```
If InputSwitch = On Then Count = Count + 1
```

Then Count would increment many times instead of just the one time expected because the program is executed continuously in a loop as described above. The qualifiers Changes, ChangesTo and NoChanges can be used on any integer or floating point variable

that is not part of an array. The firmware maintains a "change bit" for each variable that uses one of the "changes" qualifiers. Whenever a variable is assigned new data, that variable's change bit is set or cleared based on whether the new value is the same or different than the original value. As shown in the flow chart above, each program loop, after the application program is completed, all the change bits are cleared so they are ready for the next program loop. For the example above, when the input switch is first activated variable InputSwitch is set to 1 and the change bit is also set to 1 so Count is incremented. On the next program loop InputSwitch is not read (because Timer50ms did not change) and therefore remains 1, however, the change bit has been cleared so Count does not increment. Eventually, the input switch will change to off and the change bit is again set to 1 but Count only increments when InputSwitch both changes and is equal to 1. Also, notice that when the device is in Modbus slave mode, the requests from the Modbus master are processed after the change bits have already been cleared. Therefore, when the master writes to a variable in the slave device, if the variable data changes, the variable's change bit will be set. That makes it easy to write an application program in the Modbus slave device that causes some action to occur only after the master writes new data to a specific variable.

Interrupts

Interrupts are short subroutines which run in the background when an event occurs, while the normal application program is running. For example, when the device is running in Modbus slave mode the network packet data can arrive at any time during the normal program execution. This data is temporarily stored to a buffer using an interrupt subroutine which executes each time a new Modbus data byte is received. Interrupt subroutines typically execute in a few microseconds or less and, therefore, do not slow down the application program. For example, a motor encoder interrupt subroutine might execute in 1 microsecond. If the encoder interrupt occurred every 100 microseconds (a rate of 10kHz), then the CPU would spend just 1% of its time servicing the interrupt. For some applications there will be several interrupts running at the same time. All interrupts have the same priority, meaning if an interrupt is already executing and another interrupt occurs, that second interrupt must wait for the first interrupt to finish before it can be serviced. Therefore, all interrupt service routines must be kept as short as possible. See Microva application note *AN160 Precompiled Assembly Code and Interrupts in Microva BASIC* for more information.

Microva BASIC Firmware Versions

Microva BASIC supports different firmware versions for different hardware configurations. The table below shows the firmware versions supported. The firmware version is listed in the device datasheet. All versions use 32 bit RISC CPUs rated for the industrial temperature range (-40 to 85C). The main difference between firmware versions is the CPU clock speed (SysClk in the table below) and the amount of RAM and ROM supported. The Microva BASIC kernel firmware supports devices with color graphic displays (or no display). The Program statement tells the compiler software which firmware version is being used. The compiler makes sure that the application program will fit in the available ROM and that there is enough RAM for all the variables used in the program. The kernel firmware reserves the RAM amount shown in the table. Therefore, the amount of RAM usable in the application program is Total RAM - Kernel RAM. If the device has a display part of the RAM is allocated for the display buffer which is explained in the section on color graphics display.

At power on, when the device is reset, and in program mode, the firmware sets all I/O pins to the reset or "idle" condition. Many pins have alternate functions. For example, a pin might be labeled "PB14/U2TX". That means when UART2 is enabled (using the Open SerialPort command) the pin will function as the UART2 transmit pin. If UART2 is not enabled the pin can be used as a generic I/O pin which usually means it can function as an analog input, a digital input or a digital output and can be configured with a pullup resistor or can be configured as an open drain output. See the PinIn and PinOut command page. The application program Reset subroutine must configure the pins to match the hardware. Uncommitted I/O pins are configured as inputs by the firmware. These pins should not be allowed to float since the pin voltage can drift and cause the pin input buffer to draw high current. Therefore, unused pins should be re-configured as digital outputs in the Reset subroutine. See the device datasheet for the I/O pin configuration.

Table 1: Microva BASIC Kernel Firmware Versions

F/W Ver	CPU	ROM(1) Main	Boot	User	RAM(2) Total	= Kernel	+ Display	+ User	SysClk	File I/O	Display W x H	Default Baud(3)
2.5	PIC32MX130F064	64K	3K	40K	16K	= 2K	+ 0K	+ 14K	40MHz	no	none	250K
3.4	PIC32MX350F128	128K	12K	88K	32K	= 4K	+ 20K	+ 8K	100MHz	yes	160x128	1M
4.6	PIC32MX370F512	512K	12K	472K	128K	= 4K	+ 75K	+ 49K	100MHz	yes	320x240	1M
5.2	PIC32MZ1064DAK	1M	160K	1M	640K	= 8K	+ 618K	+ 14K	200MHz	yes	800x600	1M

1. User ROM is the amount of ROM available for the application program.
2. Display RAM is the default at reset which can be changed in the application program.
3. Port 1 UART baud rate set by kernel firmware at reset (node num = 1, data form = 8/N/1).

Arithmetic And Boolean Operators

Integer data values can be entered in decimal or hexadecimal (use prefix "0x") or binary (use prefix "0b"). For example, the decimal number 37 can be written 0x25 in hexadecimal or 0b100101 in binary. Float values can be entered in decimal (ex. 3.2537) or scientific notation (ex. 325.37e-02). Float data must include the decimal point and if the exponent is used the exponent sign must be included (see the Float command for further details). The tables below show the arithmetic and Boolean (logical) operators supported by Microva BASIC as well as the operator precedence. All operators are permitted with integer variables. Only add, subtract, multiply and divide are permitted with float variables. And only concatenation (add) is permitted with string data. Integer division can be done with a "rounded up" result or not. The modulo operator returns the remainder of the integer division. The sign of the modulo result is always equal to the sign of the dividend, i.e. -5 ! 3 = -2 but 5 ! -3 = 2. When there is no parenthesis or brackets in an expression, the expression will be evaluated left-to-right in order of precedence as shown in the table. For example, 5 + 2 * 3 = 11 but (5 + 2) * 3 = 21. Bit manipulation can be done using the predefined bit constants and the logical operators below. The Constant command shows examples for setting, clearing, toggling, and testing individual bits within an integer variable.

Arithmetic And Boolean Operators

Operation	Var Type			Comments and Examples
+ Addition	Int	Flt	Str	concatenation for string, "A"+"B"+"C" = "ABC"
- Subtraction	Int	Flt		arithmetic subtraction
* Multiplication	Int	Flt		arithmetic multiplication
/ Division	Int	Flt		integers are rounded up, 49/100 = 0, 50/100 = 1
\ Division	Int			integers are not rounded, 99/100 = 0, 199/100 = 1
! Modulo	Int			remainder from division, 33!6 = 3, 24!4 = 0
& Logical AND	Int			15 & 2 = 2, 0b10111 & 0b1101 = 0b101
| Logical OR	Int			12 | 2 = 14, 0b10110 | 0b1000 = 0b11110
^ Logical XOR	Int			15 ^ 2 = 13, 0b11011 ^ 0b1100 = 0b10111
{ Logic Shift Left	Int			15 { 2 = 60, 0b01101 { 4 = 0b11010000
} Logic Shift Right	Int			15 } 2 = 3, 0b10111 } 2 = 0b101

Operator Precedence

1. Parenthesis and brackets	( ) and [ ]
2. Unary functions	abs(J)
3. Multiply, divide and modulo	* / \ !
4. Addition and subtraction	+ -
5. Logical operations	& | ^ { }

Microva BASIC Program Statements

Only the Program and End Program statements and at least one other statement are required to create a valid application program. All other statements are optional. The "Hello World" application program is:

```
Program 3.3, "My Hello World Program Version 1.0"
   Print "Hello World!"                    ;this is a comment (ignored by the compiler)
End Program
```

Comments are indicated with a semi-colon as shown above. All "white space" outside of quotation marks is ignored by the compiler and "case" is ignored in commands, labels and variable names but not inside text quotes. Therefore, these statements are identical:

```
MyData [ J + 5 ] = Temp * ( K + ABS ( M ) )
mydata[j+5]=temp*(k+abs(m))
```

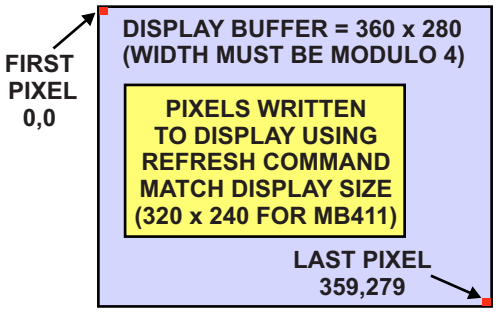
Tabs are replaced by a single space character by the compiler. Program lines can be extended by adding a space plus underline character at the end of the line which sometimes makes the program easier to read. For example, the statement:

```
If A>B And B>C And C>D Then J=1     is the same as:     If A>B And _
                                                        B>C And C>D Then _
                                                        J=1
```

A detailed explanation of every Microva BASIC command and unary function is shown on the following pages (unary functions are functions that require just one parameter). Commands are shown in blue and unary functions are shown in orange. Unary functions can be used as parameters in a command, however, commands cannot be used as parameters of functions or other commands.

Color Graphics Display

Microva BASIC supports color graphic displays. Displays are "double buffered" meaning commands that write data to the display (like Print) actually write to a display buffer in RAM memory rather than directly to the display. The Refresh command is then used to transfer the display buffer to the actual display. For color graphic displays the display buffer size can be set to any value up to the available amount of RAM memory. Each pixel on the display uses one byte in memory, therefore, a 320 x 240 display requires 76,800 bytes for the display buffer. However, the display buffer can be larger than the display size for example to hold "off screen" sprites that might be copied to the active portion of the screen using the Blit command when some event occurs. The Refresh command writes data to the display from anywhere within the display buffer. The first pixel in the display buffer, i.e. the upper most, left most pixel, is at location 0, 0. The bottom, right most pixel is at location (Width-1, Height-1) as shown in the 360 x 280 display buffer example below. At reset the kernel firmware sets the display buffer size to exactly match the display resolution. The model MB411 display resolution is 320 x 240, therefore, the display buffer size is set to 320 x 240 at reset. However, the display buffer size can be changed in the application program using system variables DisplayBufPntr, DisplayBufWidth and DisplayBufHeight. System variable DisplayBufPntr holds the starting RAM memory address, i.e. a pointer, to the display buffer. The example on the next page shows how to create the 360 x 280 display buffer. In the example, array "DisplayBuf" is set to 25,200 integers = 100,800 bytes = 360 x 280. The display buffer width and height can be changed anywhere in the application program, however, *the display buffer width must be modulo 4*. Use the RDI function to read the system variables (ex. J = rdi(DisplayBufWidth)).



RAM Allocation

The same RAM is used for the display buffer, kernel system variables and the application program variables. Table 1 shows the amount of RAM reserved for use by the kernel firmware. When the display buffer is assigned in the application program, as in the 360 x 280 example on the next page, then the compiler knows how much RAM is available for the program variables and will flag an error if the number of variables exceeds the maximum. For the example, assuming firmware version 4.6, the maximum amount of RAM that is available for variables in the application program = 128K - 4K - 360 x 280 = 26,176 bytes (6,544 integer or float variables). However, since the size of the display buffer can be changed at run time, if the display buffer is not explicitly assigned in the application, then the compiler has no way of knowing how much RAM should be allocated for the display and just assumes that the entire RAM minus the reserved kernel RAM is available for the user program (128K - 4K) and will not flag an error even if the number of user variables becomes large and begins to write into the display RAM area. As explained above, at reset, the firmware sets the display buffer size to exactly match the display size. So for the MB411 the default display buffer size = 76,800 bytes = 320 x 240. Therefore, the available application program RAM = 128K - 4K - 76,800 = 50,176 bytes (12,544 integer or float vars). If the number of vars in the application program approaches this limit, then the display buffer should be explicitly assigned so the compiler can check for a conflict. Note: Firmware version 5.2 uses an advanced CPU with a graphic display controller hardware module on chip. See Appendix B for details.

The Color Palette

As explained above, each pixel on the display is represented by one byte in the display buffer. Therefore, only 256 colors can be displayed at a time. However, the CPU actually writes 16 bits for each pixel on the display in the form of R5/G6/B5, that's 5 bits for red, 6 bits for green and 5 bits for blue (red = 0-31, green = 0-63, blue = 0-31). That means the display is capable of displaying 65,536 colors. But only 256 of these colors (out of 65,536) are on screen at any one time. This is accomplished by using a color palette in RAM which holds one 16 bit RGB value for each of the 256 color "index" values. The Microva BASIC graphic commands use the color index value. For example, if color index 5 holds the 16 bit color value for 100% blue (i.e. red=0, green=0, blue=31) then the following command will draw a blue dot to the display buffer at location X/Y: Dot X, Y, 5, DotSize The application program can read or write to the color palette in RAM using the compiler predefined constant ColorPalette which holds the address to the start of the color palette in RAM, i.e. the address of color index 0. The color palette data format is shown on the next page. For example, the command to write 100% blue to color index 5 is: WriteHalf 0x001F, ColorPalette + 5 * 2

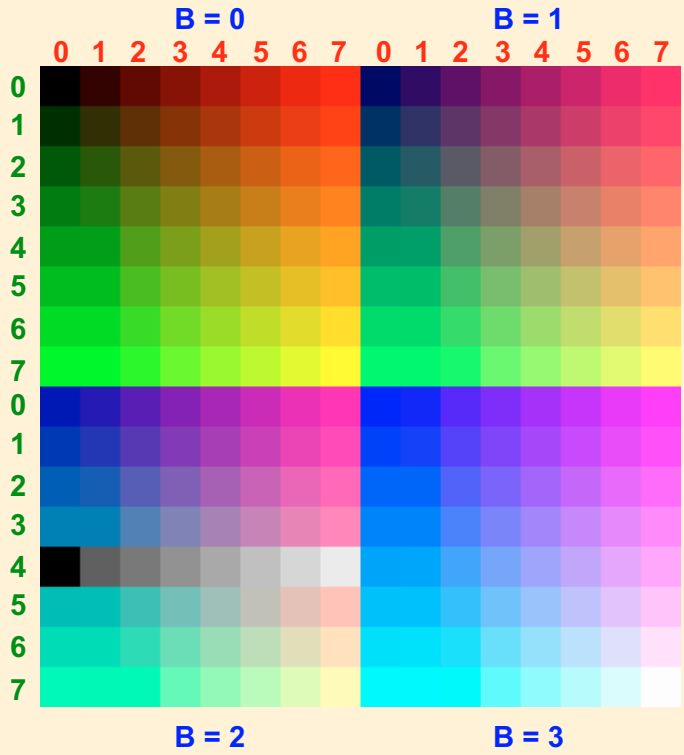
The Image command automatically loads a new color palette from the source image which could potentially overwrite all 256 colors in the color palette. At reset the kernel firmware loads the default color palette called SysColors, shown on the next page. The SysColors color palette treats each color index as if it was the 8 bit color value in the R3/G3/B2 format. For example, the 8 bit color value for 100% red (red=7, green=0, blue=0) is color index 0xE0 in the palette and the 8 bit color value for 100% yellow (red=7, green=7, blue=0) is color index 0xFC. Of course the actual color data in the palette is still 16 bits as described above. Several common colors from the SysColors color palette are shown in the table along with their color index. Note: the color names must be separately loaded from the Microva BASIC library, they are not automatically loaded by the compiler. So to write a yellow dot from the SysColors color palette the command is: Dot X, Y, yellow, DotSize

The SysColors color palette is shown in RGB order, not color index order, because that makes it easier to find a color based on the RGB value. Also, notice that the row where green = 4 and blue = 2 has been replaced with eight shades of gray since gray colors are commonly used. Which means there are actually 10 "gray" shades in the palette if black and white are included. The examples on the next page show how to re-load the SysColors color palette and how to display all the colors in the palette. The Microva BASIC compiler IDE software includes a utility for selecting colors.

The Color Palette Data Format

BYTE N+1								BYTE N+0							
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

The Default Color Palette = "SysColors"



	R	G	B	Index
black	0	0	0	0x00
red	7	0	0	0xE0
green	0	7	0	0x1C
blue	0	0	3	0x03
dkblue	0	0	2	0x02
ltblue	6	6	3	0xDB
yellow	7	7	0	0xFC
orange	7	5	0	0xF4
purple	4	0	2	0x82
white	7	7	3	0xFF
cyan	0	7	3	0x1F
dkgray	1	4	2	0x32
ltgray	6	4	2	0xD2
offwhite	7	4	2	0xF2
dkgreen	0	5	0	0x14
dkorange	7	4	0	0xF0

*see library "Colors.Lib" for predefined colors

```
;-------------------------------------------------------------------------------
;create a 360 x 280 display buffer
  Dim DisplayBuf(25200) As Integer
  WriteInt  addr(DisplayBuf), DisplayBufPntr
  WriteInt  360, DisplayBufWidth
  WriteInt  280, DisplayBufHeight
;-------------------------------------------------------------------------------
;re-load the SysColors color palette from ROM
   For J = 0 To 255
      WriteHalf  rdh(SysColors + 2 * J), ColorPalette + 2 * J
   Next J
;-------------------------------------------------------------------------------
;draw the SysColors color palette on a 320 x 240 display in RGB order as shown above
  For Y = 0 To 15
     For X = 0 to 15
        Red = X ! 8 { 5
        Green = Y ! 8 { 2
        Blue = X \ 8 + Y \ 8 * 2
        Rectangle  X * 20, Y * 15, 20, 15, Red + Green + Blue
     Next X
  Next Y
  Refresh
;-------------------------------------------------------------------------------
```

Result = **Abs**(Integer Expression)

Integer Expression	Integer expression which will be converted into its absolute value.
Result	Integer equal to the absolute value of integer expression.

Description

The ABS function returns the absolute value of an integer expression.

```
;-------------------------------------------------------------------------------
  J = -10
  K = abs(J * 2)          ;K = 20
;-------------------------------------------------------------------------------
```

Result = **ACos**(Integer Expression)

Integer Expression	Integer equal to the cosine of the angle x 10000. Range is 0 to 10000.
Result	Integer equal to the angle in degrees x 10. Range is 0 to 900 (i.e. 0 to 90.0 degrees).

Description

The ACos function returns the arccosine of the integer expression. The arccosine function uses integer math. The argument is the cosine of the angle x 10000. The result is the angle x 10 for which the cosine of that angle would be the argument. See the Cos function. The argument is restricted to the first quadrant of the unit circle, hence, the resultant angle will range from 0 to 90 degrees.

```
;-------------------------------------------------------------------------------
  J = 9239                  ;cosine of 22.5 degrees * 10000
  Ø = acos(J)               ;result = 225 (22.5 degrees * 10)
;-------------------------------------------------------------------------------
```

Result = **Addr**(Name)

Name	The name of an integer, float or string variable, array, table or appended file.
Result	Integer equal to the address of the variable or array in RAM or ROM.

Description

The Addr (**Addr**ess) function returns the memory address of a variable or array in RAM or ROM or the address of an appended file in ROM. Normally, variables are read or written using the variable name and elements in an array are read/written using the element index number. However, some commands require the memory address of the variable. For example, the FileRead command requires the start address of an array in RAM.

```
;-------------------------------------------------------------------------------
;read the same data from MyArray the normal way and directly using the "Addr" function

  J = MyArray[3]
  J = rdi(addr(MyArray) + 12)
;-------------------------------------------------------------------------------
```

Result = Ain(Integer Expression)

Integer Expression	Integer expression which equates to the pin number. Range is 0 to 127.
Result	Integer equal to the analog input value of the voltage on the input pin. Range is 0 to 1023 for 10 bit ADCs Range is 0 to 4095 for 12 bit ADCs

Description

The AIN function returns the analog input value of the voltage on the input pin. The AIN function uses the CPUs internal analog-to-digital converter (ADC) hardware. The ADC resolution is 10 or 12 bits depending on the firmware version (see the table below). Some devices use an external ADC with higher resolution in which case the AIN function is not used. The analog input pin must have the pin type set to analog input and the pin direction set to input before the AIN command is called. See the "Pin" commands for a full description of the pin numbers and the pin direction and pin types. See the device datasheet for the pin numbers which support ADC inputs. The ideal ADC input equations are:

ADC = 1023 * VIN / VAA for 10 bit resolution
ADC = 4095 * VIN / VAA for 12 bit resolution

Where VIN is the voltage on the input pin and VAA is the ADC voltage reference which depends on the hardware but in most cases is 3.3V. For example, with 1.5V on the analog input pin and with VAA = 3.3V, the returned value will be 465 for a 10 bit resolution ADC.

Caution: Exceeding the VAA voltage on the pin may damage the device.

The ADC equations above are only applicable when the input voltage is directly connected to the pin, however, in most cases the hardware is configured such that the external voltage (or current) does not directly connect to the pin. For example, the input may pass through circuitry which converts a 4-20mA current into the ADC input voltage and adds over voltage protection. See the device datasheet for a full description of the analog input circuitry.

The ADC hardware samples the pin voltage for a fixed amount of time and then converts the sampled voltage into the binary representation which also requires a fixed amount of time. Therefore, the total time required for the AIN function to execute is equal to this sample time plus the conversion time and is under 5 microseconds. Also, the analog input voltage source should have a source impedance of less than 5K ohms.

ADC Resolution

Firmware Version	ADC Resolution
2.5	10 bit (0-1023)
3.4	10 bit (0-1023)
4.6	10 bit (0-1023)
5.2	12 bit (0-4095)

```
;-------------------------------------------------------------------------------
; find the average of 100 samples of AIN1 with a 10us delay between samples

   AIN1 = 0
   For J = 1 To 100
     AIN1 = AIN1 + ain(PB12)
     DelayFast 10
   Next J
   AIN1 = AIN1 / 100
;-------------------------------------------------------------------------------
```

Append File Name
Append Subdirectory Name\File Name
Append File Path\File Name

File Name	The name of the file to append. Append file names cannot include spaces and the three letter file extension is required.
Subdirectory Name	The name of a subdirectory in the same directory as the program being compiled.
File Path	The full path name of the directory where the file to append resides.

Description

The Append command is used to append binary or text files to the MVO output ROM file. The Include and Append commands are the only non-comment statements allowed before the Program command. Each Append statement defines the location of a single source file which will be appended to the MVO ROM. Appended files are not compiled along with the application program, they are simply added to the compiled output file. Appended files always start on a modulo 4 ROM address. Appended files hold data such as images or fonts or hold pre-compiled software for interrupts. The Append file name cannot have spaces (unlike Include file names which can have spaces). The file name can be any length. The file three letter extension must be included in the file name. Append statements can be placed before or after the main program section or between subroutines. Use the Addr function to return the memory address of the file in ROM. The difference between the Include command and the Append command is the Include command is used to include ordinary text files in the application program that will be compiled along with the application program. The Append command simply appends raw data to the unused portion of the MVO output ROM file without compiling the data. Include files can have their own Append statements.

The Append command uses three methods to determine the location of the file:

1. If the file is in the same directory as the program being compiled just use the file name.
   Example: Append MyFile.txt

2. If the file is in a subdirectory in the same directory as the program being compiled include the subdirectory name.
   Example: Append MyLibrary\MyFile.txt

3. If the file is somewhere else on the computer use the full file path.
   Example: Append C:\MyFiles\MyLibrary\MyDir\MyFile.txt

```
;-------------------------------------------------------------------------------
;draw an image to the screen

  Append   Image Library\My_Bitmap_File.bmp

  Image  25, 40, addr(My_Bitmap_File)
  Refresh
;-------------------------------------------------------------------------------
```

Result = **Asc**(String Name)

String Name	The name of a string variable (not a string expression).
Result	Integer equal to the ANSI character code of the first character in the string.

Description

The Asc function returns the ANSI character code of the first character in the string. The argument must be the name of a string variable. If the string is empty the returned value is unknown. See the list of ANSI character codes in the appendix.

```
;-------------------------------------------------------------------------------
  MyString = "ÃBCDEFG"
  J = asc(MyString)        ;J = 195
;-------------------------------------------------------------------------------
```

Result = **ASin**(Integer Expression)

Integer Expression	Integer equal to the sine of the angle x 10000. Range is 0 to 10000.
Result	Integer equal to the angle in degrees x 10. Range is 0 to 900 (i.e. 0 to 90.0 degrees).

Description

The ASin function returns the arcsine of the integer expression. The arcsine function uses integer math. The argument is the sine of the angle x 10000. The result is the angle x 10 for which the sine of that angle would be the argument. See the Sin function. The argument is restricted to the first quadrant of the unit circle, hence, the resultant angle will range from 0 to 90 degrees.

```
;-------------------------------------------------------------------------------
  J = 3827                  ;sine of 22.5 degrees * 10000
  Ø = asin(J)               ;result = 225 (22.5 degrees * 10)
;-------------------------------------------------------------------------------
```

Result = ATan(Integer Expression)

Integer Expression	Integer equal to the tangent of the angle x 10000. Range is 0 to 2,000,000.
Result	Integer equal to the angle in degrees x 10. Range is 0 to 900 (i.e. 0 to 90.0 degrees).

Description

The ATan function returns the arctangent of the integer expression. The arctangent function uses integer math. The argument is the tangent of the angle x 10000. The result is the angle x 10 for which the tangent of that angle would be the argument. See the Tan function. The argument is restricted to the first quadrant of the unit circle, hence, the resultant angle will range from 0 to 90 degrees.

```
;-------------------------------------------------------------------------------
  J = 1145887               ;tangent of 89.5 degrees * 10000
  Ø = atan(J)               ;result = 895 (89.5 degrees * 10)
;-------------------------------------------------------------------------------
```

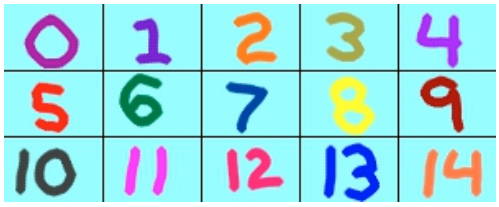
Blit SourceX, SourceY, DestinationX, DestinationY, W, H, Color

Source X, Y	Integer expressions which equate to the position in the display buffer of the upper/left corner of the source image block.
Destination X, Y	Integer expressions which equate to the position in the display buffer of the upper/left corner of the destination image block.
W, H	Integer expressions which equate to the width and height of the source (and destination) image blocks.
Color (optional)	The color index value to compare for the "transparent" color. If the value is not in the range from 0 to 255 all colors will be copied. Default value = 256.

Description

The Blit (**BL**ock **I**mage **T**ransfer) command copies a rectangular block of pixels from any source location in the display buffer to any destination location in the display buffer. The source and destination regions can overlap. The Blit command can be used to display sprites or to shift the display image. The optional Color parameter selects any of the 256 color index values to be transparent. All colors will be copied if the Color parameter is not specified or if Color is not in the range from 0 to 255. It is up to the application software to make sure the destination block width and height will fit within the display buffer.

```
;-------------------------------------------------------------------------------
;example sprite sheet
;each sprite = 60 x 40
;image size = 300 x 120
```




```
;blit the sprites for the numbers above to the display buffer position 5, 20

  Image  SX, SY, addr(SpriteSheet)
  For N = 0 To 14
     Blit  SX + 60 * (N ! 5), SY + 40 * (N \ 5), 5, 20, 60, 40
     Refresh
     Delay  5000
  Next N
;-------------------------------------------------------------------------------
```

Result = **Cfi**(Float Expression)
Result = **Cfid**(Float Expression)
Result = **Cfiw**(Float Expression)

Float Expression	Float value to convert to an integer.
Result	Integer result.

Description

The CFI (**C**onvert **F**loat to **I**nteger) function converts the IEEE 754 single precision (32 bit) floating point argument into its 32 bit integer equivalent. The integer result is rounded up.

```
;-------------------------------------------------------------------------------
   P = 3.4999        ;P = 3.4999
   J = cfi(P)        ;J = 3
;-------------------------------------------------------------------------------
   P = -3.4999       ;P = -3.4999
   J = cfi(P)        ;J = -3
;-------------------------------------------------------------------------------
   P = 3.5           ;P = 3.5
   J = cfi(P)        ;J = 4
;-------------------------------------------------------------------------------
   P = -3.5          ;P = -3.5
   J = cfi(P)        ;J = -4
;-------------------------------------------------------------------------------
```

The CFID (**C**onvert **F**loat to **I**nteger **D**irect) function directly converts the IEEE 754 single precision (32 bit) floating point argument into an integer value. The difference between function CFI and function CFID is in the first case the numeric value of the float is converted to the integer and in the second case the IEEE 754 format hex value itself is written to the integer. The examples below show the difference.

```
;-------------------------------------------------------------------------------
   P = 73.5          ;P = 73.5 (which is 0x42930000 in IEEE 754 format)
   J = cfi(P)        ;J = 74
   J = cfid(P)       ;J = 0x42930000
;-------------------------------------------------------------------------------
```

The CFIW (**C**onvert **F**loat to **I**nteger **W**hole portion only) function converts the "whole" portion of the IEEE 754 single precision (32 bit) floating point argument into an integer value. This function is the same as the CFI function except the integer result is not rounded up.

```
;-------------------------------------------------------------------------------
   P = 3.4999        ;P = 3.4999
   J = cfiw(P)       ;J = 3
;-------------------------------------------------------------------------------
   P = -3.4999       ;P = -3.4999
   J = cfiw(P)       ;J = -3
;-------------------------------------------------------------------------------
   P = 3.9           ;P = 3.9
   J = cfiw(P)       ;J = 3
;-------------------------------------------------------------------------------
   P = -3.9          ;P = -3.9
   J = cfiw(P)       ;J = -3
;-------------------------------------------------------------------------------
```

String Result = **Cfs**(Float Expression)
String Result = **Cfsf**(Float Expression)

Float Expression	Float value which will be converted into a string.
String Result	A string which represents the value of the float argument.

Description

The CFS (**C**onvert **F**loat to **S**tring) function returns a string which represents the numeric value of the float argument.

The CFSF (**C**onvert **F**loat to **S**tring and **F**ormat) function also returns a string which represents the numeric value of the float argument except the string result will be formatted to a fixed decimal number using the kernel system variable DecForm (the decimal format). See the NDF function page for a detailed explanation of decimal formatting.

```
;-------------------------------------------------------------------------------
  MyFloatVar = 43.125
  MyString = cfs(MyFloatVar)                ;MyString = "43.125"
  MyString = ndf(0x22) + cfsf(MyFloatVar)   ;MyString = "43.13"
;-------------------------------------------------------------------------------
```

String Result = **Chr**(Integer Expression)

Integer Expression	ANSI character code which will be converted into a string.
String Result	An ANSI string character equal to the least significant byte of the argument.

Description

The CHR (character) function returns a string character which represents the ANSI character code of the integer expression argument. See the appendix for a table of ANSI character codes. The difference between CIS and CHR is that CIS converts the integer into its numeric value whereas CHR converts the integer into its ANSI string character. For example, cis(65) returns the string "65" but chr(65) returns the string "A" (since character code 65 represents the capital letter A). ANSI character codes are 1 byte, therefore, the CHR function only uses the least significant byte of the integer argument (the other bytes are ignored). The CHR function is used to enter characters in a string that cannot be typed on the keyboard. For example, the semicolon character is used in Microva BASIC to indicate a comment so it cannot be entered in a string, use the CHR function instead such as:

"Bob's from Utah; Jane's from Ohio" becomes "Bob's from Utah" + chr(59) + " Jane's from Ohio"

To print the string which is the chemical symbol for water with a "sub 2" on a graphic display the command is:

```
Print  "H" + chr(183) + "O"
```

The CHR function is also used to include print control characters in the text string. See the Print command for an explanation of the print control codes.

```
;-------------------------------------------------------------------------------
;init var with string plus "carriage return" and "line feed"

   MyString = "Hello World" + chr(CR) + chr(LF)
;-------------------------------------------------------------------------------
```

Result = **Cif**(Integer Expression)
Result = **Cifd**(Integer Expression)

Integer Expression	Integer to convert into a float value.
Result	Float result.

Description

The CIF (**C**onvert **I**nteger to **F**loat) function converts the integer argument into the IEEE 754 single precision (32 bit) floating point format as shown in example:

```
;-------------------------------------------------------------------------------
   J = 123          ;J = 123
   P = cif(J)       ;P = 123.0 (which is 0x42F60000 in IEEE 754 format)
;-------------------------------------------------------------------------------
```

The CIFD (**C**onvert **I**nteger to **F**loat **D**irect) function directly converts the integer argument into the IEEE 754 single precision (32 bit) floating point format. The difference between function CIF and function CIFD is in the first case the integer value is assumed to be the "whole" portion of the floating point number and in the second case the integer value is assumed to already be in the IEEE 754 format, except its stored in an integer variable instead of a float variable (hence the need for CIFD conversion). The examples below show the difference.

```
;-------------------------------------------------------------------------------
   J = 123          ;J = 123
   P = cif(J)       ;P = 123.0 (which is 0x42F60000 in IEEE 754 format)
;-------------------------------------------------------------------------------
   J = 0x42F60000   ;J = IEEE 754 representation of integer value 123
   P = cifd(J)      ;P = 123.0 (which is 0x42F60000 in IEEE 754 format)
;-------------------------------------------------------------------------------
```

String Result = **Cis**(Integer Expression)
String Result = **Cisf**(Integer Expression)

Integer Expression	Integer value which will be converted into a string.
String Result	A string which represents the value of the integer argument.

Description

The CIS (**C**onvert **I**nteger to **S**tring) function returns a string which represents the numeric value of the integer argument.

The CISF (**C**onvert **I**nteger to **S**tring and **F**ormat) function also returns a string which represents the numeric value of the integer argument except the string result will be formatted to a fixed decimal number using the kernel system variable DecForm (the decimal format). See the NDF function page for a detailed explanation of decimal formatting.

```
;-------------------------------------------------------------------------------
  MyIntVar = 43725
  MyString = cis(MyIntVar)                ;MyString = "43725"
  MyString = ndf(0x23) + cisf(MyIntVar)   ;MyString = "43.725"
;-------------------------------------------------------------------------------
```

Close SerialPort1
Close SerialPort2
Close SpiPort

Description

The Close commands turn off serial port 1 (UART1) or serial port 2 (UART2) or the SPI hardware module and return the I/O pins to direct control of the application program. See the Open SerialPort and Open SpiPort commands for a description of these hardware modules. These commands have no parameters.

CLS Color

Color	An integer expression which equates to the color index used to clear the display buffer.

Description

The CLS (**CL**ear the **S**creen) command writes the color index value to every pixel location in the display buffer. The display buffer is automatically cleared at reset using the command:

```
CLS 0
```
Note: color 0 is black in the default SysColors color palette.

Firmware Version 5.2

Firmware version 5.2 uses an advanced CPU with a Graphic Display Controller (GDC) hardware module that controls the display timing and resolution. The display screen will have a passive color border around the center live area of pixels on the screen (live meaning the graphic display commands such as Print, Ellipse, Rectangle, etc can write to this area). The border can be left black, which is the default color at reset, or it can be changed in the application program. The CLS command does not update the border color, therefore, the border color must be updated separately using the predefined memory address GDCBorderColor as explained in Appendix B. See the example below.

```
;-------------------------------------------------------------------------------
;clear the display buffer using color index 0x1C (green in the default color palette)

 CLS  green
 Refresh
;-------------------------------------------------------------------------------
;firmware 5.2 command to set the screen blue

 WriteInt  0x0300, GDCBorderColor
 CLS  blue
 Refresh
;-------------------------------------------------------------------------------
```

Constant Name=Value, Name=Value, Name=Value, . . .

Name	The name of the constant.
Value	The integer or float data value of the constant.

Description

The Constant command is used to define integer or float constants. See the Integer and Float commands for a detailed explanation of these data types. The constant name can be used anywhere in the application program that the integer or float data value can be used. The Constant command can only be used after the Program statement but before the first program line in the program or between the subroutine definitions. The Microva BASIC Compiler predefines several commonly used constants. Many of these predefined constants are listed in the relevant command description. For example, the constants which equate to the pin numbers are shown on the PinOut command page. The tables below show most of the predefined constants. The bit constants shown in the table below make it easy to set, clear, toggle, and test individual bits in an integer variable as shown in the examples below.

Predefined "Bit" Constants Common To All Firmware Versions

b0 = 0x01	b8 = 0x100	b16 = 0x10000	b24 = 0x1000000
b1 = 0x02	b9 = 0x200	b17 = 0x20000	b25 = 0x2000000
b2 = 0x04	b10 = 0x400	b18 = 0x40000	b26 = 0x4000000
b3 = 0x08	b11 = 0x800	b19 = 0x80000	b27 = 0x8000000
b4 = 0x10	b12 = 0x1000	b20 = 0x100000	b28 = 0x10000000
b5 = 0x20	b13 = 0x2000	b21 = 0x200000	b29 = 0x20000000
b6 = 0x40	b14 = 0x4000	b22 = 0x400000	b30 = 0x40000000
b7 = 0x80	b15 = 0x8000	b23 = 0x800000	b31 = 0x80000000

Other Constants Common To All Firmware Versions

On = 1	PortDir =-16	Timer1Max = 0x7FFFFD78	I2C_Start= -1
Off = 0	PortLatch = 16	Pi = 3.1415927	I2C_Stop = -2
True = 1	PortOpenDrain= 32	SerialIO = 0	I2C_Done = -3
False = 0	PortPullup = 48	ModbusMaster = 1	I2C_Read = -4
	PortType =-32	ModbusSlave = 2	

Firmware Dependent Constants

ColorPalette	PortA
DisplayBufHeight	PortB
DisplayBufPntr	PortC
DisplayBufWidth	PortD
FilePntr	PortE
FontPntr	PortF
GdcBorderColor	PortG
IptVectors	PortH
MbRxCounter	
SysColors	
SysFont	

```
;-------------------------------------------------------------------------------
  MyVar = MyVar | (b7 + b23)          ;set bits 7 and 23
  MyVar = MyVar & not(b2 + b15)       ;clr bits 2 and 15
  MyVar = MyVar ^ (b1 + b14)          ;toggle bits 1 and 14
  If MyVar & b5 <> 0 Then J = J + 1   ;test bit 5, increment J if 1
;-------------------------------------------------------------------------------
```

Result = **Cos**(Integer Expression)

Integer Expression	The angle in degrees x 10. Range is 0 to 3599 (i.e. 0 to 359.9 degrees).
Result	Integer equal to the cosine of the angle x 10000.

Description

The Cos function returns the cosine of the angle x 10,000. The cosine function uses integer math. The result can easily be converted into floating point using the CIF function (see example below). The argument is the angle in degrees x 10.

```
;-------------------------------------------------------------------------------
  Ø = 225                 ;angle = 22.5 degrees
  J = cos(Ø)              ;integer result = 9239
  P = cif(J) / 1.0e+4     ;float result = 0.9239
;-------------------------------------------------------------------------------
```

Result = **Csf**(String Name)

String Name	The name of a string variable (not a string expression).
Result	Float equal to the numeric value of the string.

Description

The CSF (**C**onvert **S**tring to **F**loat) function returns the float value of a text string. The argument must be the name of a string variable. The string can be in standard decimal form (ex. 75.35) or in scientific notation (ex. 7.535e1). If the string is empty or if any of the following rules are violated, the returned value will be 0. The rules for a valid float string are:

1. The significand can include no more than 8 digits, therefore, the maximum number of chars in the significand is 10 (including the sign and decimal point, ex. -1.2345678e-15).
2. The exponent range is +/-30 and there can be only 1 or 2 digits in the exponent (3 chars maximum if the sign is included in the exponent, sign can be "+" or "-").

The string is considered completed (the float value will be returned) at the string end or when any "non-valid" float character is encountered in the string. The space and comma characters are considered non-valid characters. Also, a second decimal point in the number is considered non-valid.

The following are examples of "good" float strings (i.e. they will return the float values expected):

```
1234567.8e-2   5.3e12   +5      +5e3      .3    5e3     3.e+b     12345678e-30    12345678e30
99999999e30    5.3E3    5.3e3   5.3e-03   5.3   54321   3.e+3     .00000001e-30   5.3e-6,7.5
-352.62e-07    125      1.25    3.2eng    1.3.4 15.     3.2e1-2    0.2e+22        00327.650e2
0.0            0.0e2    0       -.3     .12345678e-5     1.23,51.5e-3
```

The following are examples of "bad" float strings (i.e. they will return 0):

```
1234567.80e-2     5.3e-012     5.3e003     3.2e-40     0123.45678
```

Result = **Csi**(String Name)

String Name	The name of a string variable (not a string expression).
Result	Integer equal to the numeric value of the string.

Description

The CSI (**C**onvert **S**tring to **I**nteger) function returns the integer value of a text string. The argument must be the name of a string variable. The string can be an integer (ex. "1234") or a hex value (ex. "0xFE39") or a binary value (ex. "0b110101"). If the string is empty or if any of the following rules are violated, the returned value will be 0. The rules for a valid integer string are:

1. For a "decimal" string there is a maximum of 9 digits if there is no sign.
   If the sign is included there is a maximum of 10 chars (9 digits plus the sign, sign can be "+" or "-").
2. For a "hex" string there is a maximum of 10 chars (8 digits plus "0x" prefix and prefix must be "0x" not "0X").
3. For a "binary" string there is a maximum of 34 chars (32 digits plus "0b" prefix and prefix must be "0b" not "0B").

The string is considered completed (the integer value will be returned) at the string end or when any "non-valid" integer character is encountered in the string. The non-valid character depends on the integer type. For example, "73.95" returns 73 since the decimal point is a non-valid character for an integer value. Likewise, string character "2" is a valid character for a decimal value but is non-valid for a binary string (ex. 0b112 returns integer value = 3).

The following are examples of "good" integer strings (i.e. they will return the integer values expected):

```
123456789   -123456789   +123456789   -23,14,57   0xF5A   0xf5a   0x89ABCdef   123oz
-1234   0x12345678     000     045     0b1    0b01105     0b11110000111100001111000011110000
```

The following are examples of "bad" integer strings (i.e. they will return 0 or a value not expected):

```
0123456789   0x123456789    0X55    0xG3    0B10    +0x3E    0b2    0x123feet
- 1234                                                0b11110000111100001111000011110001
```

Delay Milliseconds
DelayFast Microseconds

Milliseconds	An integer expression which equates to the delay time in milliseconds. Range is 1 to approximately 40,000.
Microseconds	An integer expression which equates to the delay time in microseconds. Range is 1 to approximately 40,000,000.

Description

The Delay and DelayFast commands pause the application program for the amount of time specified. The delay time is accurately measured using the CPU system clock which is generated from the highly accurate crystal oscillator. Interrupts continue to operate during the pause. The Delay and DelayFast commands block the application program from executing other commands. Therefore, in most cases, it is better to create program delays by using a timer derived from the millisecond timer, rather than using the delay commands. See the Timer function for more details.

```
;-------------------------------------------------------------------------------
;turn on the red LED for 1 second

PinOut  rled:on
Delay   1000
PinOut  rled:off
;-------------------------------------------------------------------------------
```

Dim Name=Value, Name=Value, Name=Value, . . . **As Integer**
Dim Name=Value, Name=Value, Name=Value, . . . **As Float**
Dim Name, Name, Name, . . . **As String**

Name	The name of the variable. To define an array include the array size (the number of elements in the array) in parenthesis after the name.
Value (optional)	The default value for the integer or float variable or array. The default value is zero if no value is assigned. Strings cannot be assigned a default value.

Description

The Dim (**Dim**ension) command is used to define integer, float or string variables. The Dim command can define one or more variables or variable arrays of the same type in a single line of code. Variables can also be defined using the Integer, Float and String commands. See those commands for a detailed explanation of the variable types. For integer and float type variables a default value can be assigned to each individual variable or array. All variables are set to their default value by the kernel firmware at reset. The default value is zero for integer and float variables if the variable is not explicitly assigned a default value. When an array is assigned a default value all elements in the array are assigned to that default value. String vars are always assigned a default value equal to the "null string" (a string with length = 0). String vars must include the string array size which ranges from 8 to 256 and must be evenly divisible by 4. The Integer, Float and String commands can be used to assign additional variable attributes beyond what the Dim command can assign such as a flag marking the variable as nonvolatile and the Modbus address. The Dim command can only be used after the Program statement but before the first program line in the program or between the subroutine definitions. All Microva BASIC variables are "public" meaning they can be read or written within any subroutine. Variables cannot be defined within a subroutine (which would make them "private").

```
;-------------------------------------------------------------------------------
  Dim JN, Ø=53.825, X(40) = 3.75 As Float        ;define 2 vars plus an array w/ 40 elements
  Dim AKÙî(12), Text1(36), Text2(128) As String  ;strings can't be assigned default values
;-------------------------------------------------------------------------------
```

Dot X, Y, Color, Size

X, Y	X and Y are integer expressions which equate to the center position of the dot drawn to the display buffer.
Color	An integer expression which equates to the color index of the dot.
Size	An integer expression which equates to the size of the dot. Range is 1 to 33.

Description

The Dot command draws a dot (point or spot) to the screen using Color. The dot size range is 1 to 33 pixels. The dot will be centered on X/Y. The application software must make sure the dot does not draw outside of the display buffer.

```
;-------------------------------------------------------------------------
;draw a dot

 Dot  25, 45, Blue, 5
 Refresh
;-------------------------------------------------------------------------
```

Ellipse X, Y, W, H, Color, Line Width

X, Y	X and Y are integer expressions which equate to the center position of the ellipse drawn to the display buffer.
W	An integer expression which equates to the ellipse width. Must be greater than 2.
H	An integer expression which equates to the ellipse height. Must be greater than 2.
Color	An integer expression which equates to the color index of the ellipse.
Line Width (optional)	An integer expression which equates to the ellipse outline width and also determines which quadrants to draw. Range is 0 to 33.

Description

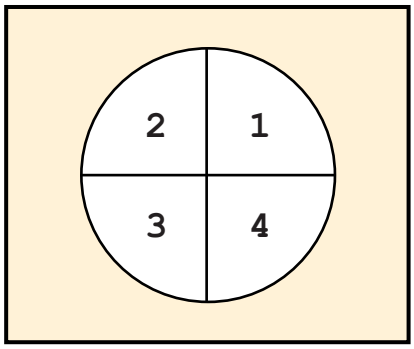
The Ellipse command draws a solid (filled) ellipse or an ellipse outline to the display buffer. If the Line Width parameter is not included or if the line width value is 0 the ellipse will be drawn solid. When the line width parameter is included and is not 0 only the ellipse outline will be drawn. The application software must make sure the ellipse does not draw past the display buffer RAM. Both the solid ellipse and the ellipse outline are drawn using the color index value. The Line Width parameter determines the ellipse outline width and also determines which quadrant of the ellipse to draw. The lower 8 bits of the line width (bits 0 to 7) set the outline width. The minimum line width is 0 which means the ellipse will be drawn solid. The maximum line width is 33.

Bits 8 thru 11 of Line Width determine which of the four quadrants of the ellipse are drawn- for both the outline and the solid ellipse.

```
Bit 11 = quadrant 4 (0 = draw, 1 = don't draw)
Bit 10 = quadrant 3 (0 = draw, 1 = don't draw)
Bit  9 = quadrant 2 (0 = draw, 1 = don't draw)
Bit  8 = quadrant 1 (0 = draw, 1 = don't draw)
```

The example below shows how to draw the ellipse outline only in quadrants 1 and 3. The line width parameter is not needed to draw a solid ellipse with all four quadrants.

Ellipse Quadrants




```
;-------------------------------------------------------------------------------
;draw quadrant 1 and 3 of an ellipse with line width = 7

 Ellipse  55, 75, 20, 40, Blue, b11 + b9 + 7
 Refresh
;-------------------------------------------------------------------------------
```

Exit

Description

The Exit command is used to exit a For...Next loop before the loop completes as shown in the example below. This command has no parameters.

```
;-------------------------------------------------------------------------------
  For J = 0 To Ubound(MyTable)         ;check if MyTable holds data = K
    If MyTable[J] = K Then Exit
  Next J
  If J <= Ubound(MyTable) Then Print "found K" Else Print "no K"
;-------------------------------------------------------------------------------
```

Result = **fAbs**(Float Expression)

Float Expression	Float expression which will be converted into its absolute value.
Result	Float equal to the absolute value of float expression.

Description

The FABS function returns the absolute value of a float expression.

```
;-------------------------------------------------------------------------------
  P = -2.3e-2
  F = fabs(P * 21.5)          ;F = 0.4945
;-------------------------------------------------------------------------------
```

Result = **FileAppend**(String Expression, Date/Time)

String Expression	The string which will be appended to the open file.
Date/Time (optional)	The address of an integer array which holds the date and time information to write to the open file date/time attribute. The first 7 values in the array must be: DateTime[0] = month (1-12) DateTime[1] = date (1-31) DateTime[2] = year (0-99) DateTime[3] = day (0-6) (Sunday = 0) DateTime[4] = hour (0-23) DateTime[5] = minute (0-59) DateTime[6] = second (0-59)
Result	A direct or indirect integer variable which holds the result of the command: n = the size of the new file after the string was appended -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized or is corrupted) -3 = no file is opened or file was corrupted -4 = illegal new file size (new file size > 2GB) -6 = illegal date/time data -7 = sector write failed, sector was not able to be written within 500ms -9 = can't create file, no free clusters found (or card needs defragmentation)

Description

The FileAppend command appends a string to the file and optionally updates the file date/time attribute. The file must be opened using the FileOpen command before the string can be appended to the file. The file size attribute in the file directory will be changed to reflect the new file size after the string is appended. Typically, when a file is changed the file date/time attribute is also changed. The file date/time attribute is normally displayed when the file name is shown on a computer using any modern operating system. Keep in mind when the file name is displayed, the time value shown may be adjusted for daylight savings time by the operating system (i.e. if the time shows 9:05am on March 1st, it may show 10:05am on March 31 due to the local time change even though the time attribute was written = 9:05am). If no date/time parameter is included in the command the file date/time attribute will not be changed. The FileAppend command can take anywhere from a few milliseconds up to hundreds of milliseconds to complete depending on the file size and how fragmented the memory card is (see the FileRead command for an explanation of fragmented files) and the memory card speed class (written on the card).

```
;-------------------------------------------------------------------------------
;append "Hello World" to file, don't change file date/time

 J = FileAppend("Hello World")
;-------------------------------------------------------------------------------
;append MyString + "Hello World" to file, update file date/time using array DateTime

 J = FileAppend(MyString + "Hello World", addr(DateTime))
;-------------------------------------------------------------------------------
;append 4 bytes of data to file

 J = FileAppend(chr(Data1) + chr(Data2) + chr(Data3) + chr(Data4))
;-------------------------------------------------------------------------------
```

Result = **FileDelete**()

Result	A direct or indirect integer variable which holds the result of the command: 0 = file deleted -1 = no response from memory card (card not installed). -2 = can't initialize card (not an SDHC memory card with FAT32 formatting) -3 = file corrupted (file size or start cluster doesn't match "FileOpen" data) -7 = sector write failed (memory card is write protected or write time exceeded max allowed = 500ms)

Description

The FileDelete command deletes the currently opened file on the memory card. In order to delete the file, the file's directory must be writeable, if not the command returns an error code (for example if the memory card is write protected).

```
;-------------------------------------------------------------------------------
; Create a new calibration data file and delete the previous file if it exists.

 Sub NewCalDataFile()
    J = OpenSDHC()
    If J < 0 Then Return "Error"
    J = FileOpen("CALDATA.TXT")
    If J >= 0 Then J = FileDelete()
    J = FileNew("CALDATA.TXT")
 End Sub
;-------------------------------------------------------------------------------
```

Result = **FileList**(Number of Files, Destination Address, Starting File Num, Data Size)

Number of Files	An integer expression which is the number of files to list.
Destination Address	An integer expression which is the address of an integer array in RAM to store the file list data. Must be evenly divisible by 4.
Starting File Num	An integer expression which is the starting file number of the first file to list.
Data Size	An integer expression which is the size of each file record. Must be evenly divisible by 4. The minimum value is 24 (bytes).
Result	A direct or indirect integer variable which holds the result of the command: n = the number of file records returned (0 if no files found) -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized using "OpenSDHC")

Description

The FileList command is used to obtain a list of the Number of Files in the root directory beginning with the Starting File Num. Files are numbered starting with 0 so if there are ten files in the root directory they will be numbered 0 thru 9. The file number changes as files are added and deleted from the memory card. Each file record returned, i.e. each file list, is a mix of integer, half integer and string data. The format is: FileSize + FileTime + FileDate + ShortName + FileType + LongName

Where FileSize is an integer value, FileTime and FileDate are half integer values (two bytes each) and ShortName, FileType and LongName are standard Microva BASIC strings. ShortName is the file name from the 8.3 "short file name" format with all spaces removed and will therefore range from 1 to 8 ANSI characters. See the FileOpen command for an explanation of the 8.3 file name format. FileType is the file extension from the 8.3 format name with all spaces removed and will therefore range from 1 to 3 ANSI characters. LongName is the file long name unmodified. Note that all files have a short file name but not all files have a long file name. Also, the integer and half integer data is in little endian format and is copied directly from the file directory FAT. The location of the binary and string data within each file record is fixed, for example the FileType string always starts at offset 17 within the record even if the ShortName string is less than 8 characters. The file record address offsets are:

Data	Addr Offset	Num Bytes
FileSize	0	4
FileTime	4	2
FileDate	6	2
ShortName	8	9
FileType	17	4
LongName	21	Data Size - 21

For example, the record for the file named "My Data File.txt" is:

Address	Description
0-3	file size
4-5	file time, bits 15:11 = hours (0-23), bits 10:5 = minutes (0-59), bits 4:0 = seconds / 2 (0-29, i.e. 0 to 58 secs)
6-7	file date, bits 15:9 = year - 1980 (0-127, i.e. 1980 to 2107), bits 8:5 = month (1-12), bits 4:0 = date (1-31)
8	ShortName size = 8
9-16	"MYDATA~1"
17	FileType size = 3
18-20	"TXT"
21	LongName size = 16
22-37	"My Data File.txt"

Only files in the root directory on the memory card will be listed. Sub directories will not be listed. Each record is written to the Destination Address which must be an integer array in RAM. The size of each record is fixed and is equal to Data Size. For example, if Data Size was 40 bytes the records for the first three files would be written to Destination Address + 0, + 40, + 80. Therefore, the size of the destination address array in bytes = Number of Files x Data Size. The long file name string will be truncated if it does not fit within (Data Size - 21). The FileList command returns the number of file records written to the destination address up to the maximum number specified in the Number of Files parameter. Therefore, if Result equals Number of Files then there are likely more files in the directory than those listed in which case FileList can be executed again starting at the next file record. The example on the next page shows how to display the file records for the first 40 files on the memory card.

Result = **FileList**(Number of Files, Destination Address, Starting File Num, Data Size)

Description (continued)

```
;-------------------------------------------------------------------------------
;display the file records (date + time + short file name + long file name) for the
;first 40 files on the memory card (assumes 736 x 430 display resolution)

  Dim Month, Date, Hours, Mins as Integer
  Dim J, K, L, M, FileNames(880), FileSize, Year As Integer
  Dim LongName(80), ShortName(12), FileType(8) As String

  WriteInt  addr(F0810A), FontPntr
  M = OpenSDHC()
  M = FileList(40, addr(FileNames), 0, 88)
  For J = 0 To M - 1
     K = addr(FileNames) + J * 88
     FileSize = rdi(K)
   ;time
     L = rdh(K + 4)
     Hours = L } 11
     Mins  = L } 5 & 0x3F
   ;date
     L = rdh(K + 6)
     Month = L } 5 & 0x0F
     Date = L & 0x1F
     Year = (L } 9) + 1980
   ;name
     ShortName = linein#(K + 8)
     FileType  = linein#(K + 17)
     LongName  = linein#(K + 21)
   ;print record
     Print 0, J*10, white, ndf(0x20) + cisf(Month) + "/" + cisf(Date) + "/" + _
      cis(Year) + " " + cisf(Hours) + ":" + cisf(Mins) + " " + ShortName + "." + _
      FileType + " " + LongName
  Next J
  Refresh
;-------------------------------------------------------------------------------
```

Result = **FileLoadPgm**()

Result	A direct or indirect integer variable which holds the result of the command: -1 = file size wrong, range is 128 < file size <= max MVO ROM size

Description

The FileLoadPgm command loads a new application program into the device. The file must already be open using the FileOpen command. Only the files size is checked to see that the file will fit in the application program Flash ROM. If the file fits in the ROM then the entire application ROM is first erased, then the new file is loaded into ROM, then the CPU is reset. Therefore, if the new program is valid, it will begin execution immediately. If the program is not valid, for example if the program is for a different firmware version, then the reset will cause the CPU to enter program mode and, if the device has a display, the display will show "no program" because the ROM has been erased.

```
;-------------------------------------------------------------------------
;load a new application program

  J = OpenSDHC()
  J = FileOpen("PONG.MVO")
  J = FileLoadPgm()
;-------------------------------------------------------------------------
```

Result = **FileNew**(File Name, Date/Time)

File Name	A string expression which is the name of the new file.
Date/Time (optional)	The address of an integer array which holds the date and time information to write to the new file date/time attribute. The first 7 values in the array must be: DateTime[0] = month (1-12) DateTime[1] = date (1-31) DateTime[2] = year (0-99) DateTime[3] = day (0-6) (Sunday = 0) DateTime[4] = hour (0-23) DateTime[5] = minute (0-59) DateTime[6] = second (0-59)
Result	A direct or indirect integer variable which holds the result of the command: n = file was opened, n is the size of the file in bytes -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized or is corrupted) -3 = unable to open new file -4 = file already exists but file size exceeds 2GB -5 = illegal file name -6 = illegal date/time data -7 = sector write failed, sector was not able to be written within 500ms -9 = can't create file, no free clusters found (or card needs defragmentation)

Description

The FileNew command creates a new file on the memory card. If a file with the file name already exists on the memory card then the FileNew command is identical to the FileOpen command (the date/time parameter is ignored). If the file name does not exist, and the card is not full, the new file will be created with size = 0 bytes. The File Name must follow the file name rules shown on the FileOpen command page. If the date/time parameter is included it will be written to the new file date/time attribute. If the date/time is not included the new file date/time attribute will be set to: Jan 1, 2010, 12:00:00 (am). Keep in mind when the file name is displayed on a computer, the time value shown may be adjusted for daylight savings time by the operating system (i.e. if the time shows 9:05am on March 1st, it may show 10:05am on March 31 due to the local time change even though the time attribute was written = 9:05am). Use the FileAppend or FileResize commands to increase the size of the file.

```
;-------------------------------------------------------------------------------
;create new file and append "Hello World" to file

 J = FileNew("MYFILE.TXT", addr(DateTime))
 J = FileAppend("Hello World")
;-------------------------------------------------------------------------------
```

Result = **FileOpen**(File Name)

File Name	A string expression which is the name of the file to open.
Result	A direct or indirect integer variable which holds the result of the command: n = file was opened, n is the size of the file in bytes -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized or is corrupted) -3 = file not found -4 = file found but file size exceeds 2GB -5 = illegal file name

Description

Before data can be read from or written to a file, the file must be opened using the FileOpen command. And before the FileOpen command can be used the SDHC memory card must be "mounted", i.e. initialized, using the OpenSDHC command. The File Name string is used to search for the file on the memory card. Only the memory card root directory will be searched. The File Name must use the "8.3" format naming convention with all upper case letters (and the 3 character file extension is required). Long file names can be opened using their "short" filename as described below. All files have a short filename but not all files have a long file name. The FileList command can be used to find the long and short filenames for all the files in the root directory on the memory card.

The rules for valid "short" File Names are:

1. Must be in 8.3 format.
2. Letters must be all upper case.
3. "Long" file names can be opened using the 8.3 format since all long file names also have a corresponding "short" file name. For example, the file name: `This is a long file name.txt` can be opened using the file name: `THISIS~1.TXT`. To convert a long file name to a short file name do the following: Remove all spaces, capitalize all letters, keep the first six characters, add "~1", add the extension.
4. The following characters are permitted in the file name (any others return "illegal file name"):
   Upper case letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
   Numbers: 0123456789
   Punctuation characters: ! # $ % & ' ( ) - @ ^ _ ` { } ~
   ANSI codes 0x80 thru 0xFF (see ANSI codes in appendix)

Only one file can be open at a time. There is no "FileClose" command. To open a new file while another file is already opened, just execute the FileOpen command with the new file name. The FileOpen command can be used at any time, even on the same file that is already opened, for example to check the size of the file after writing new data to the file or to see if the memory card is still inserted.

```
;-------------------------------------------------------------------------------
  J = FileOpen("123.123")              ;OK
  J = FileOpen("1.BIN")                ;OK
  J = FileOpen("MY_FILE.TXT")          ;OK
  FileSource = "MY_FILE"
  J = FileOpen(FileSource + ".TXT")    ;OK
  J = FileOpen("123.abc")              ;Error: letters must be upper case
  J = FileOpen("123")                  ;Error: file extension is required
  J = FileOpen("123.AB")               ;Error: file extension requires 3 chars
  J = FileOpen("MYNEWFILE.TXT")        ;Error: file name length = 8 max + extension
;-------------------------------------------------------------------------------
```

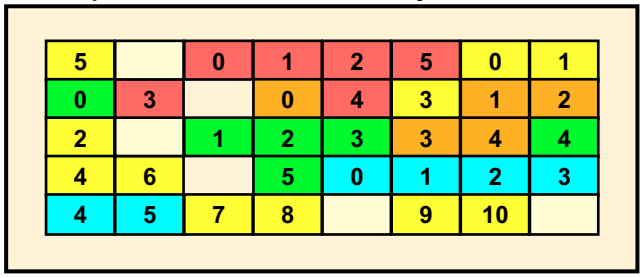
Result = **FileRead**(Sector Number, Address)

Sector Number	An integer expression which is the sector number in the file to read or -1 which indicates to read "the next sector" from the file when the file is not fragmented.
Address	An integer expression which is the address of an integer or float array in RAM to store the data read from the file.
Result	A direct or indirect integer variable which holds the result of the command: 0 = sector was successfully read -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized using "OpenSDHC" and "OpenFile") -3 = file is corrupted (FAT and file directory don't match) -4 = sector number does not exist (sector number > file size) -8 = illegal Destination Array Address (address must exist in RAM)

Description

The FileRead command reads one sector of data from an already opened file (see FileOpen command). Use the FileReadLine command to read string data from a text file. Files are stored in "sector format" on the SDHC memory card. A sector is 512 bytes (128 float or integer variables). The number of sectors in the file depends on the file size. For example, a file 1085 bytes in size would have 3 sectors: sector 0 (512 bytes), sector 1 (512 bytes) and sector 2 (61 data bytes plus 451 unknown bytes). Attempting to read from sector number 3 would produce an error. When reading from sector 2, the first 61 bytes from the sector will be saved to the destination array along with 451 bytes of unknown data. The FileRead command does not check the size of the destination data array (the size must be >= 512 bytes). A sector can be read an unlimited number of times. The FileRead command completes the sector read in anywhere from a few hundred microseconds up to tens of milliseconds or more depending on the file size and how fragmented the memory card is (explained below) and the memory card speed class (written on the card). The figure below shows an example representation of five files stored on a memory card. As shown, the files have become fragmented as previous files were deleted, new files were added or when files already stored on the card were appended with new data. The file sectors are not necessarily adjacent on the memory card or in numerical order. Therefore, before the FileRead command can read the data, the data sector location on the memory card must be found. Depending on how many files are on the card and the length of the file being read, this may require reading the File Allocation Table (FAT) several times (each time is an additional sector read). For example, to read sector 875 of an 8 megabyte file, there might need to be 4 FAT sector reads just to find the data sector location on the memory card, plus the actual data sector read, i.e. 5 total sector reads to get 1 sector of data. This is why large fragmented files can slow down the FileRead command. In this example the FileRead might execute in 5ms instead of the normal 1ms. That extra few milliseconds probably doesn't matter when the application program is only occasionally reading from the file. However, in some cases the extra time to search for the data sector locations is not acceptable (for example with streaming audio).

Example File Sectors On A Memory Card



Streaming Data Mode

The FileRead command can be used in "streaming data" mode by setting the Sector Number parameter to -1. In this mode the FileRead command will keep track of the last sector read and just read the next sector on the memory card every time the command is executed without searching the FAT for the sector locations. Streaming data mode only works when the file is continuous on the memory card (such as the blue file in the figure above). The best way to keep files on the memory card continuous (non-fragmented) is to start with a new memory card (or a newly formatted card) and only write complete files to the card. Do not delete files on the card or change a file size or append data to files on the card or create directories or subdirectories. The first sector read from the streaming file must be read using the regular FileRead command with the starting sector number. Subsequent FileRead commands can then use -1 for the sector number. Also, in streaming data mode the application program is responsible for determining when the end of the file is reached.

```
;-------------------------------------------------------------------------------
  K = OpenSDHC()                          ;Read sectors 5 to 10 from file to array "MyData"
  K = FileOpen("MYFILE.DAT")
  For J = 0 To 5
     K = FileRead(J+5, addr(MyData) + J * 512)
  Next J
;-------------------------------------------------------------------------------
```

Result = **FileReadLine**(String Name)

String Name	The name of a string variable where the text read from the file will be written.
Result	A direct or indirect integer variable which holds the result of the command: n = length of the text string read from the file 0 = end of file -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized using "OpenSDHC" and "OpenFile") -3 = file is corrupted (FAT and file directory don't match)

Description

The FileReadLine command reads one line of text from a text file. The file must already be open using the FileOpen command. Use the FileRead command to read integer or float data from a data file. The text file must have been saved in the ANSI or ASCII or "raw text" or unicode UTF-8 format. If the file is in unicode UTF-8 format only the character codes 0x00 thru 0x7F (the ASCII character codes) shown in the ANSI table in the appendix can be used. Formatted text such as from a DOC or PDF file will not be read properly. Most text editor programs, such as Word or Notepad, have the option to save the file in the ANSI file format using the "save as" command. Each text line in the file must be terminated with the "end of line" (EOL) characters. The EOL is CR or LF or CR + LF where CR is the carriage return character code (ANSI code 0x0D) and LF is the line feed character code (ANSI code 0x0A). The EOL for Windows text files is CR + LF. The EOL for Linux text files is LF.

The system variable FilePntr is used to set the starting location in the file to search for the next text string. FilePntr is a predefined constant which holds the memory address of the variable. Location 0 is the first byte in the file and location (file length - 1) is the last byte in the file. The FileOpen command sets FilePntr to 0 so there is no need to write to FilePntr when reading text lines from the start of the file. The FileReadLine command begins searching for a complete line of text, i.e. a line of text terminated with the EOL as described above, at the FilePntr location in the file. The line of text is returned in the String Name variable. The text line will be truncated to fit in String Name if the variable's size is not large enough to hold the entire text line. After each text line is returned the FilePntr is automatically updated to the file location of the start of the next text line in the file. The returned text line will include the EOL characters. For example, for a Windows text file, the text line "Hello" will return the string length = 7 ("Hello" + CR + LF). Note that the last line in a text file often does not include the EOL.

```
;-------------------------------------------------------------------------------
;print all text lines from a file

  J = OpenSDHC()
  J = FileOpen("MYFILE.TXT")
  J = FileReadLine(MyTextLine)
  While J > 0
     Print MyTextLine
     J = FileReadLine(MyTextLine)
  Wend
;-------------------------------------------------------------------------------
;return "World" from a file which has just two lines:
;   Hello
;   World

  WriteInt  7, FilePntr
  J = FileReadLine(MyTextFile)
;-------------------------------------------------------------------------------
```

Result = **FileRename**(File Name)

File Name	A string expression. The currently open file will be renamed to "File Name".
Result	A direct or indirect integer variable which holds the result of the command: 0 = file was renamed -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized or is corrupted) -3 = file corrupted (file size or start cluster doesn't match "FileOpen" data) -5 = illegal file name -7 = sector write failed, sector was not able to be written within 500ms (card may be write protected)

Description

The FileRename command is used to change the name of the currently open file. Only the file name will be changed, the file data and the file date/time attributes will not be changed. See the FileOpen command for a description of the valid file names.

```
;-------------------------------------------------------------------------------
;rename the currently open file to "MYFILE.TXT"

 J = FileRename("MYFILE.TXT")
;-------------------------------------------------------------------------------
```

Result = **FileResize**(New Size, Date/Time)

New Size	An integer expression which equates to the new file size in bytes. The maximum file size = 2GB.
Date/Time (optional)	The address of an integer array which holds the date and time information to write to the open file date/time attribute. The first 7 values in the array must be: DateTime[0] = month (1-12) DateTime[1] = date (1-31) DateTime[2] = year (0-99) DateTime[3] = day (0-6) (Sunday = 0) DateTime[4] = hour (0-23) DateTime[5] = minute (0-59) DateTime[6] = second (0-59)
Result	A direct or indirect integer variable which holds the result of the command: n = file size was changed, n is the size of the file in bytes -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't read sector data (card was not properly initialized or is corrupted) -3 = file corrupted (file size or start cluster doesn't match "FileOpen" data) -4 = illegal new file size, max is 2GB (range is 0 to 0x7FFF FFFF) -6 = illegal date/time data -7 = sector write failed, sector was not able to be written within 500ms (card may be write protected) -9 = can't create file, no free clusters found (or card needs defragmentation)

Description

The FileResize command changes the size of the currently open file on the memory card. The file size attribute in the file directory and the FAT table will be changed to reflect the new file size. When the file size in increased, the additional data appended to the file is unknown (not necessarily 0). When the file size is decreased, data is lost from the end of the file. If the date/time parameter is included it will be written to the file date/time attribute. If the date/time is not included the file date/time attribute will not be changed. Keep in mind when the file name is displayed on a computer, the time value shown may be adjusted for daylight savings time by the operating system (i.e. if the time shows 9:05am on March 1st, it may show 10:05am on March 31 due to the local time change even though the time attribute was written = 9:05am). The FileResize command can be used to change the file date/time attribute without changing the file size by setting the new file size equal to the original file size and providing the new date/time parameter.

```
;-------------------------------------------------------------------------------
;create new file "MYFILE.BIN" with size = 4000 bytes (data unknown)

 J = OpenSDHC()
 J = FileNew("MYFILE.BIN", addr(DateTime))
 J = FileResize(4000)
;-------------------------------------------------------------------------------
```

Result = **FileWrite**(Sector Number, Address)

Sector Number	An integer expression which is the sector number in the file to write or -1 which indicates to write "the next sector" in the file when the file is not fragmented.
Address	An integer expression which is the address of the source data in ROM or RAM.
Result	A direct or indirect integer variable which holds the result of the command: 0 = sector was successfully written -1 = no response from memory card (card was not initialized using "OpenSDHC" or the card was removed). -2 = can't write sector data (card was not properly initialized using "OpenSDHC" and "OpenFile") -3 = file is corrupted (FAT and file directory don't match) -4 = sector number does not exist (i.e. sector number > file size) -7 = sector write failed (memory card is write protected or write time exceeded max allowed = 500ms)

Description

The FileWrite command writes one sector of data to an already opened file (see FileOpen command). Use the FileAppend command to write string data to a text file. All files are stored in "sector format" on the SDHC memory card. A sector is 512 bytes (128 float or integer variables). The sector must already exist in the file and will be over written with the new source data. The number of sectors in the file depends on the file size. For example, a file 1085 bytes in size would have 3 sectors: sector 0 (512 bytes), sector 1 (512 bytes) and sector 2 (61 bytes plus 451 unknown bytes). Attempting to write to sector number 3 would produce an error. When writing to sector 2, the first 61 bytes from the source data array will be written along with 451 bytes of unknown data. The FileWrite command does not check the size of the source data array. It always writes 512 bytes starting at the source data address even if the source array is less than 512 bytes long.

Due to the type of memory used in an SDHC memory card, a single sector can "only" be written approximately 100,000 times total before that sector on the card becomes damaged. Therefore, the same file sector should not be written more than this number of times. The FileWrite command does not update the file directory information or the File Allocation Table (FAT) since the file name, file length and file time/date is not changed. Only the one data sector in the file on the memory card is written. The FileResize command can be used to update the file time/date attributes if needed as well as increase (or decrease) the file size.

The FileWrite command waits for the sector to complete writing on the SDHC memory card before returning. This can take anywhere from less than a millisecond up to 500 milliseconds. After 500mS, if the sector write has not completed, the FileWrite command will return with an error (usually that means the memory card is write protected). The write speed depends on the file size and how fragmented the memory card is (see the FileRead command for an explanation of fragmented memory cards) and the memory card speed class (written on the card).

The FileWrite command can be used in "streaming data" mode by setting the Sector Number to -1. This mode is used to write data to files which are continuous (non-fragmented) when write speed is a priority. See the FileRead command explanation of "streaming data" mode for more details.

```
;------------------------------------------------------------------------
;Write the first 4 sectors of file

  K = OpenSDHC()
  K = FileOpen("MYFILE.DAT")
  For J = 0 To 3
     K = FileWrite(J, addr(MyData) + J * 512)
  Next J
;------------------------------------------------------------------------
```

Result = **fiSqr**(Float Expression)

Float Expression	Float expression which will be converted into its inverse square root. Must be greater than 0.
Result	Float equal to the inverse square root of float expression.

Description

The FISQR function returns the inverse square root of a float expression (i.e. 1 divided by the square root).

```
;-------------------------------------------------------------------------------
  P = 0.08
  Ê = fisqr(P * 8.0)          ;Ê = 1.24999
;-------------------------------------------------------------------------------
```

FlashErase Page Number
FlashWrite Data, Address

Page Number	Integer expression which is the page number to erase.
Data	Integer expression which is the integer value to write.
Address	Integer expression which equates to an address in Flash ROM. Must be modulo 4.

Description

The FlashErase command erases a page in Flash ROM. The FlashWrite command writes a single integer value to a memory address in Flash ROM. The FlashWrite memory address must be modulo 4 (i.e. evenly divisible by 4) and be within the unused portion of the application program ROM or the command will just return (i.e. do nothing). Likewise, the FlashErase page number must equate to an unused page in the application program or the command will just return. Use the RDB, RDH, RDI or RDF functions to read data from Flash ROM. The Flash ROM memory (ROM) holds the firmware kernel and the application program. Typically, the application program does not use all of the available ROM memory. The unused ROM can be used to store data needed in the application program such as calibration data for analog inputs or outputs. Before a ROM memory location can be written with new data it must be erased. The entire application program section of the ROM is erased each time a new application program is written to the device. Therefore, the unused portion of the application program ROM is ready to be written with new data after a new program is loaded. All erased ROM memory locations will return -1 (0xFFFF FFFF) when read.

Any erased ROM memory location can be written using the FlashWrite command. Sometimes the ROM must be written with new data but the ROM memory location is not erased such as when the ROM has previously been written with calibration data and the device is now due for re-calibration. In this case, the FlashErase command can be used to erase a portion of the ROM. The ROM is divided into pages. The size of a page depends on the firmware version as shown in the table below. The FlashErase command erases one full page in the ROM. The Microva BASIC kernel firmware will not allow a ROM page to be erased if it holds any portion of the application program. Therefore, only unused pages can be erased in the ROM (i.e. pages which hold no part of the application program). Use the SYS function to return the number of used and unused pages in the application program. Sys(0) returns the number of pages used, Sys(1) returns the number of unused pages and Sys(2) returns the starting address of the application program in the Flash ROM. The size of the application program ROM space depends on the firmware version. Page numbers start at zero, for example, if there are 2 used pages and 2 unused pages then page numbers 0 and 1 are the used pages and pages 2 and 3 are unused. The Microva BASIC Compiler displays the program size and the total program ROM size when the program is compiled successfully. For example, the compiler output might be:

```
Total Program Size = 5498 (6% of 90112)        ;for page size = 4096, pages 0-1 are used, pages 2-21 are free
```

Due to the nature of Flash ROM type memory, the Flash ROM can "only" be erased a maximum of 20,000 times. Likewise, the same memory location can only be written 20,000 times. Erasing or writing the same ROM memory location more than 20,000 times may cause damage to the ROM. The FlashWrite command requires up to 500us to complete. The FlashErase command can take up to 30ms to complete. All CPU interrupts are turned off during the FlashErase and FlashWrite command execution. CPU interrupts are described in more detail at the beginning of this document.

Flash ROM Page Size

Firmware Version	Page Size
2.5	4K
3.4	4K
4.6	4K
5.2	16K

```
;-------------------------------------------------------------------------------
;erase the last page in Flash ROM
  FlashErase  sys(0) + sys(1) - 1
;-------------------------------------------------------------------------------
;write float data to last address in ROM and read back (assumes 4096 byte page size)
  LastAddr = sys(2) + 4096 * (sys(0) + sys(1)) - 4
  FlashWrite  cfid(MyFloat), LastAddr
  MyFloat = rdf(LastAddr)
;-------------------------------------------------------------------------------
```

Float Name, Default Value, NV Flag, Modbus Address

Name	The name of the float variable. To define a float array include the array size in parenthesis after the name.
Default Value	The float will be set to this value at reset.
NV Flag (optional)	Nonvolatile variables retain their value when the power is off. Check the device data sheet to see if NV vars are supported. The Program command must include the +n option in order to use the NV flag option. Both individual variables and arrays can be designated as nonvolatile. 0 = variable is volatile 1 = variable is nonvolatile
Modbus Address (optional)	An integer constant which is the Modbus slave address of the variable. Each float variable uses two Modbus addresses. The addresses are only used when serial port 1 (UART1) is set to Modbus slave mode. When defining an array the Modbus addresses are incremented for each variable in the array. Modbus slave addresses must be unique. Range is 1 to 4095.

Description

The Float and Dim commands are used to define floating point variables. Microva BASIC float values use the IEEE 754 standard 32 bit floating point data format (called a "single") with a range of approximately +/-1.0e-38 to +/-1.0e+38. A float array is defined in exactly the same way as an integer array (see the Integer command). Float values can be written as ordinary decimal values (ex. 37.255) or in scientific notation (ex. 3.7255e+01 or 0.37255e+2 or 3725.5e-2 which are all the same as 37.255). Float values must include the decimal point (ex. 37255e-03 will generate a compiler error but 37255.0e-03 will not) and the float exponent must include the sign (ex. 3.7255e1 will generate an error but 3.7255e+1 will not). Therefore, zero must be written 0.0 not 0.

Nonvolatile (NV) Variables

Nonvolatile variables retain their value when the power is shut off. The NV Flag option can be used for devices that include the EERAM memory for nonvolatile variables. There is no difference between volatile and nonvolatile variables during normal program execution, i.e. there is no speed penalty for using nonvolatile variables. If the Program command includes the +n option then the NV flags must be included for each Float variable defined. If the Program command does not include the +n option then the NV flags should not be included. Each nonvolatile float variable uses one EERAM address. So a nonvolatile float array of 10 elements would use 10 EERAM addresses. See the NvVarsLoad command page for more information on nonvolatile variables.

Modbus Address

The Modbus address is optional. The address must be written after the NV flag option if that option is included. The address is only used in Modbus slave mode. All Modbus addresses must be unique. Float variable "change bits" are updated three ways (see the explanation for "change bits" at the beginning of this document):

1. By the application program running in the device.
2. When the device is in Modbus slave mode: by the Modbus master when the master writes to the variable.
3. When the device is in Modbus master mode: when a variable is read from a Modbus slave node using the Modbus command.

Modbus slave addresses are stored in a ROM table. The size of the table is equal to 6 times the highest Modbus address, i.e. if the highest address is 250, then the table size = 1500 bytes. Therefore, Modbus slave addresses should be assigned starting with address one and increasing as needed (Modbus address 0 is reserved for the node identification string). For that reason, when defining an Float variable, the maximum Modbus slave address allowed is 4095. However, in Modbus master mode any address can be used (the Modbus address range is 0 to 65,535). Float variables are always treated as 32 bit values for Modbus communication, therefore, two Modbus addresses are used for each variable. The Float variable Modbus address is not used when the device is in Modbus master mode because in master mode the Modbus command sets the addresses that will be read or written. See the Modbus command for more information.

```
;-------------------------------------------------------------------------------
  Float   MaxPressure,  1.25e+02,  1,  5     ;MaxPressure is NV, Modbus address = 5/6
  Float   PressData(4),     0.0,  1,  10    ;Modbus addresses = 10/11, 12/13, 14/15, 16/17
;-------------------------------------------------------------------------------
```

For Count = Start Value **To** End Value **Step** Offset
<one or more statements>
Next Count

Count	The loop counter. Must be a direct or indirect integer variable. If it is an indirect variable, the array index must be a constant or another variable name not an expression (i.e. MyVar[K] is okay, MyVar[K*5] is not).
Start Value	An integer expression which is the initial value for the loop
End Value	An integer expression which is the compare value for the loop.
Offset (optional)	An integer constant which is added to Count when the Next statement is encountered. The range for Offset is -32768 to +32767. The default value is 1.

Description

The For...Next statement is used to create a program "loop", i.e. a block of statements which will repeat until the loop count is completed. Count holds the loop count and will be initialized with the result of Start Value when the For...Next loop begins. End Value is also evaluated when the loop begins and the result is saved in temporary memory until it is needed by the Next statement. Both Start Value and End Value are evaluated just one time- at the start of the For...Next loop. When the Next statement is encountered the Offset is added to Count and the result is compared with the End Value previously stored in temporary memory.

The comparison depends on whether Offset is positive or negative. If Offset is positive (or zero) the program loop will repeat as long as Count is less than or equal to End Value. If Offset is negative the program loop will repeat as long as Count is greater than or equal to End Value. The loop will always execute at least one time regardless of the initial conditions of Start Value and End Value since the conditions are not checked until the Next statement is encountered. For the default Step value of +1, the loop will end with Count = End Value + 1. For example, the statement: `For J = 1 To 5` will execute the loop 5 times and will end with J = 6.

The value of Count can be changed inside the For...Next loop if needed- for example to increment Count with a variable (instead of the Step constant) or to use a non-linear method to increment Count. In these cases the Offset value should be set to 0. The Exit and/or Return statements can be used within the For...Next loop in order to exit before the loop completes as shown below. The Goto statement may not be used within a For...Next loop.

```
;-------------------------------------------------------------------------------
  Factorial = 1                         ;find the factorial of N
  For J = 1 To N
    Factorial = Factorial * J
  Next J
;-------------------------------------------------------------------------------
  For J = 0 To Ubound(MyData) Step 2    ;multiply even elements in float array by 0.25
    MyData[J] = MyData[J] * 0.25
  Next J
;-------------------------------------------------------------------------------
  For J = 0 To Ubound(MyTable)          ;check if MyTable holds data = K
    If MyTable[J] = K Then Exit
  Next J
  If J <= Ubound(MyTable) Then Print "found K" Else Print "no K"
;-------------------------------------------------------------------------------
```

Result = **fSqr**(Float Expression)

Float Expression	Float expression which will be converted into its square root. Must be greater than 0.
Result	Float equal to the square root of float expression.

Description

The FSQR function returns the square root of a float expression.

```
;-------------------------------------------------------------------------------
  P = 21.125
  Ê = fsqr(P * 8.0)          ;Ê = 13.000
;-------------------------------------------------------------------------------
```

Goto Label

Label	The location in the main program or subroutine where program execution will continue.

Description

The Goto command jumps from one location in the main program section to another location in the main program section or from one location in a subroutine to another location in the same subroutine. The Goto command cannot be used to jump out of the main program section or jump out of a subroutine. Label indicates the new location for program execution. The Goto command cannot be used within a For...Next loop.

```
;-------------------------------------------------------------------------------
;freeze while in standby mode
   If StandBy = On Then Goto PgmDone
   ReadInputs
   WriteOutputs
   RefreshScreen
PgmDone:
   End Program
;-------------------------------------------------------------------------------
```

String Result = **Hex**(Integer Expression)

Integer Expression	Integer value which will be converted into a string of hexadecimal digits.
String Result	A string which represents the hexadecimal value of the integer argument.

Description

The HEX (**HEX**adecimal) function returns a string which represents the hexadecimal value of the integer expression argument. The returned hexadecimal string is always 8 characters. The string is not prefixed with "0x".

```
;-------------------------------------------------------------------------------
;convert integer values to hexadecimal string
  J = 8
  MyString = hex(J * 4)              ;MyString = "00000020"
  MyString = "0x" + hex(J * -525)    ;MyString = "0xFFFFEF98"
;-------------------------------------------------------------------------------
```

Result = I2C(Integer Expression)

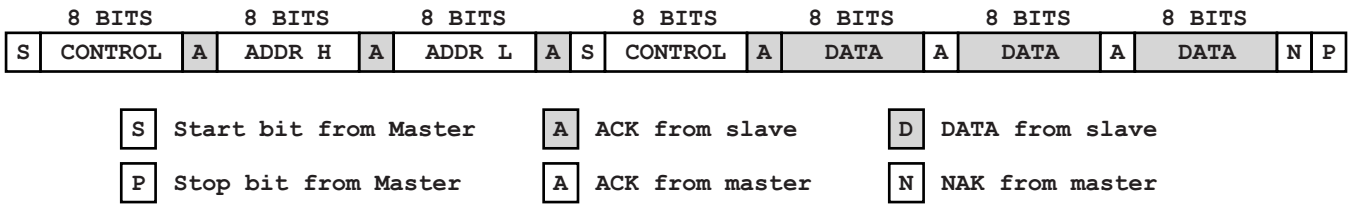
Integer Expression	The address of an integer array in RAM or ROM that holds the I2C script.
Result	Integer result: 0 = no error, -1 = received NAK from the slave device.

Description

The I2C function is used to read and write data to I2C compatible devices. The I2C I/O port must be opened using the Open I2CPort command before the I2C function can be used. The I2C function uses a simple script in an integer array to control how data is read or written to the I2C slave device. The script must be in RAM when reading data from an I2C device. The script can be in RAM or ROM when writing to an I2C device. The I2C function firmware checks each integer value in the source array. If the value is zero or positive then the least significant byte is written to the I2C slave device. If the value is negative then the firmware will either transmit the Start bit or transmit the Stop bit or read one or more bytes from the slave or just end, depending on the negative value. The negative value "I2C script commands" are all predefined constants. The I2C integer array script values are:

Integer Value	Action Taken
0 or positive	write the least significant byte to the slave, if the slave does not ACK the data end the command (return value = -1) otherwise continue
I2C_Start= -1	send the start bit
I2C_Stop = -2	send the stop bit and end the command (return value = 0)
I2C_Done = -3	end the command without sending the stop bit (return value = 0)
I2C_Read = -4	read one or more bytes from the slave, ACK each byte except the last byte send NAK, then send the stop bit and end the command (return = 0)

This method provides the most versatility when reading/writing to the wide variety of I2C slave devices. For example, the data sheet for the 47L16 EERAM describes the method to read 3 bytes as:



The script for this transaction is shown below. Note that data written to the slave must be acknowledged (ACK) by the slave or the I2C function will end immediately with the return value = -1 (NAK). Data read from the slave is ACK'ed by the master except for the final byte read which is NAK'ed. The script value for read (-4) must be followed in the array by the number of bytes to read. The bytes read are written to the array following the script, therefore, the array must be large enough to hold the script plus the bytes read (one byte for each integer in the array). For this example, the three bytes read will be at array locations 8, 9 and 10. The I2C function automatically transmits the Stop bit after the final byte is read. Therefore, the script command for the Stop bit is not needed after the "read bytes" command, however, it is needed when the script is only writing data (in which case the array can be in ROM).

```
;-------------------------------------------------------------------------------
; read three bytes from 47L16 EERAM

  Dim I2C_Data(11) As Integer

  I2C_Data[0] = I2C_Start    ;start bit
  I2C_Data[1] = 0xA0         ;write byte: 7 bit device addr + write bit (0)
  I2C_Data[2] = 0x05         ;write byte: memory addr high
  I2C_Data[3] = 0x55         ;write byte: memory addr low
  I2C_Data[4] = I2C_Start    ;new start bit
  I2C_Data[5] = 0xA1         ;write byte: 7 bit device addr + read bit (1)
  I2C_Data[6] = I2C_Read     ;read bytes from memory then send stop bit
  I2C_Data[7] = 3            ;num bytes to read
  I2C_Data[8] = ?            ;1st byte read
  I2C_Data[9] = ?            ;2nd byte read
  I2C_Data[10]= ?            ;3rd byte read

  J = i2c(addr(I2C_Data))
;-------------------------------------------------------------------------------
```

If Comparison Then True Statement Else False Statement

Comparison	A comparison between two integer expressions or two float expressions or a string name and a string expression. If the comparison is true the "True Statement" (or statements) will be executed. If the comparison is false the "False Statement" (or statements) will be executed.
True Statement	The statement (or statements) to execute if the comparison is true.
False Statement (optional)	The statement (or statements) to execute if the comparison is false.

Description

The If...Then...Else statement is used for decision making. The comparison must be made between two integer expressions or two float expressions or between a string name and a string expression, i.e. both sides of the comparison must be of the same type, integer or float or string values. The If...Then...Else statement can be used in the single line form or the "block" form. In both cases the Else statements are optional. If...Then...Else statements can be nested using the block form but not within the single line form. The block form of the statement must be terminated using the End If statement. The block form can also include an unlimited number of ElseIf qualifiers. The table on the following page shows the comparison operators.

Integer Comparisons

When comparing very large integer values, keep in mind that the comparison is actually done with an internal subtraction. For example, the comparision If J > K will internally subtract J from K and check if the result is negative. If the result of the subtraction overflows (or underflows) a 32 bit integer value then the comparison will be wrong. For example, the comparison If 5 > -2147483645 will give an incorrect result because (-2147483645 - 5) "underflows" a 32 bit signed integer value because the most negative integer value is -2147483648. However, the comparison If -5 > -2147483645 is valid because (-2147483645 + 5) does not underflow.

Float Comparisons

When comparing floating point numbers it is best to not use "equal" in the comparison. That's because float values are converted to IEEE 754 binary format. In many cases the binary version of the decimal number does not exactly match. For example, in the equation P = 0.1, P is set to decimal 0.1 but the IEEE 754 version of 0.1 is actually 0.09999999. Therefore, the If...Then...Else statement If P * 10.0 = 1.0 Then J = 1 Else J = 0 sets J = 0 not 1 as expected due to these tiny floating point rounding errors.

String Comparisons

For string comparisons only the "equal" and "not equal" comparisons are allowed. Also, the left side of the string comparison must be a string name. The right side of the comparison can be any string expression. Also, case matters, i.e. "Moo" is not equal to "moo".

And/Or Qualifiers

The And operator and the Or operator can be used to combine several comparisons in a single If...Then...Else statement. The comparisons are evaluated independently and do not need to be the same type, i.e. they can be mixed integer, float or string compares. The comparisons are evaluated in order from left to right and the results are combined using Boolean logic. For example, in the statement If A > B And B > C Or B < D Then M = M + 1 there are 3 comparisons which are evaluated in order. Assume the result of the first comparison is X the result of the second comparison is Y and the result of the third comparison is Z. Results X, Y, and Z are boolean values 1 or 0 (true or false). The compare results are placed back into the If...Then...Else statement and re-evaluated using standard Boolean logic to determine the final result. The new statement becomes: If (X And Y Or Z) = 1 Then M = M + 1 Note that changing the order of the Boolean operators changes the result, i.e. X And Y Or Z is not the same as Y Or Z And X. The comparison is always evaluated in order from left-to-right and parenthesis cannot be used to change the order, i.e. X And (Y Or Z) is not allowed.

Change Bits

The Changes, NoChanges and ChangesTo qualifiers can be used with individual integer or float variables but not with array or string variables. Each time a variable is assigned a new value, that variable's "change bit" is set or cleared. If the new value does not equal the old value then the bit is set. If the new value is equal to the old value the bit is cleared. In this way it is possible to compare for when a variable changes or doesn't change or changes to a specific value. See the section about "change bits" at the beginning of this manual for more information. Also see the examples below.

If Comparison Then True Statement Else False Statement

Description (continued)

Comparison Operators

=	equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
<>	not equal
And	Boolean "And"
Or	Boolean "Or"
Changes	changes to any
ChangesTo	changes to a value
NoChanges	no changes

```
;-------------------------------------------------------------------------------
;change bit comparisons

  K = 5
  J = 1
  J = 5
  If J ChangesTo K Then M = 1 Else M = 0  ;M = 1
  J = 5
  If J ChangesTo K Then M = 1 Else M = 0  ;M = 0
;-------------------------------------------------------------------------------
;single line form examples

  If MyString = "Hello" + chr(CR) And J = 5 * K Then PinOut RLed:On Else PinOut RLed:Off
  If abs(J - 7) = MyVar[K * 5] * (M - cos(A) + L) Or K = 8 * (-3 + M) Then GetNewData
;-------------------------------------------------------------------------------
;block form can include "ElseIf" (both "ElseIf" and "Else" are optional)

  If J = 5 Or K = -3 + M Then
     If Alarm = Off Then
        Relay5 = On
        PinOut  23:Relay5
     End If
  ElseIf J = 5 Or K = 0 Then
     PinOut RLed:On, GLed:Off
  ElseIf MyString = "Hello" Then
     PinOut RLed:Off, GLed:On
  Else
     For L = 1 To K
        MyArray[L - 1] = 2 * DataIn
     Next L
  End If
;-------------------------------------------------------------------------------
```

Image X, Y, Image Address, Color

X, Y	X and Y are integer expressions which equate to the position in the display buffer to draw the image starting with the upper, left corner of the image.
Image Address	An integer expression which equates to the address of the image source file in RAM or ROM.
Color (optional)	For Monochrome (1 bit/pixel) Bitmaps The color index to draw for pixels with bit 1. For Color (8 bit/pixel) Bitmaps The color index which is transparent, i.e. not drawn to the display buffer. The transparent color must be within the range from 0 to 255 otherwise all colors will be drawn. The default Color value is 256.

Description

The Image command draws an image to the display buffer. The source image must use the device independent bitmap format. The standard file name extension for a bitmap source file is "bmp". See Microva application note *AN165 BMP File Format* for the bitmap file specification. The source bitmap must use the monochrome (1 bit/pixel) or color palette (8 bit/pixel) formats. It is up to the application software to make sure the bitmap fits within the display buffer.

Monochrome Bitmap Source

The bitmap bit 0 pixels are transparent, i.e. they are not drawn to the display buffer. The bitmap bit 1 pixels are drawn using the color index in Color. The bitmap source file color palette is not used for monochrome bitmaps.

Color Bitmap Source

The bitmap source must use the 8 bit/pixel format (256 colors maximum). The Image command replaces the display color palette with the image color palette. For example, color red is index 224 (0xE0) in the SysColors color palette which is automatically loaded at reset. However, once the image is loaded, red will no longer be at index number 224. Red may still exist in the color palette but it will most likely be at a different index number. Its also possible that there will be no red color in the palette at all, for example if there were no red shades in the image. The SysColors palette can be reloaded at any time. See the description of the color palette at the beginning of this manual. The Image command Color option is used to define the transparent color, i.e. the color index number which will not be drawn to the display buffer.

Firmware 5.2

Firmware 5.2 (but not the other firmware versions) forces color index 0x00 in ColorPalette to black (R/G/B value = 0x0000) for the reasons explained in the appendix. If the bitmap color 0 is not already black then the firmware will search the palette to see if black exists at another index location. If it does then that index value and index 0 will be swapped. If black does not exist in the palette then the firmware will search for the closest color to black and the closest color to color 0 in the palette. If black is closest then that color index will be swapped with color 0 in the image. If there's a color in the palette which is closer to color 0 than any color is to black, then color 0 in the image will be replaced with that closest color and the original color 0 will be set to black.

```
;-------------------------------------------------------------------------------
;draw an image to the display from ROM
  Append  MyBitmap.bmp
  Image  24, 45, addr(MyBitmap)
  Refresh
;-------------------------------------------------------------------------------
;load a 256 x 190 pixel image from the memory card and display
;Note: Also see the library function "ImageLoad" which loads a bitmap directly to the
;      display buffer without requiring a separate image RAM buffer
  Dim MyData(12430) As Integer   ;bitmap image file size = 49718
  J = OpenSDHC()
  J = FileOpen("SAMPLE.BMP")
  For K = 0 To 97
     J = FileRead(K, addr(MyData) + K * 512)
  Next K
  Image 0, 0, addr(MyData)
  Refresh
;-------------------------------------------------------------------------------
```

Include File Name
Include Subdirectory Name\File Name
Include File Path\File Name

File Name	The name of the file to include.
Subdirectory Name	The name of a subdirectory in the same directory as the program being compiled.
File Path	The full path name of the directory where the file to include resides.

Description

The Include command is a pre-processor directive (meaning it is executed before the program is compiled) which is used to include external source files in the application program. Each Include statement defines the location of a single source file which will be compiled along with the application program. If used, the Include statements must be the first non-comment lines in the source program file- before the Program statement and before any Append statements. The included files are simply appended to the application program in the order that the Include statements appear and are compiled along with the application program. The included file must be an ordinary text file the same as the application program. The file name can be any standard file name on disk (file name spaces are allowed). Typical file name extensions for Include files are "txt", "bas" or "lib", however, any file name extension will work. Microva BASIC library source code usually uses the "lib" file name extension.

Included files cannot have their own Include statements but they can have Append statements. The difference between the Include command and the Append command is the Include command is used to include ordinary text files in the application program that will be compiled along with the application program. The Append command appends raw binary or text files to the unused portion of the MVO output ROM file without compiling the files. Include commands are used to add library subroutines. Append commands are used to add data files like fonts or images.

The Include command uses three methods to determine the location of the file:

1. If the file is in the same directory as the program being compiled just use the file name.
   Example: Include MyFile.txt

2. If the file is in a subdirectory in the same directory as the program being compiled include the subdirectory name.
   Example: Include MyLibrary\MyFile.txt

3. If the file is somewhere else on the computer use the full file path.
   Example: Include C:\MyFiles\MyLibrary\MyDir\MyFile.txt

```
;-------------------------------------------------------------------------------
;return the natural logarithm of a float between 0.0 and 10.0 from table

  Include   Math Library\Natural Logarithms Table.txt

  J = ln[cfi(MyFloat * 10.0)]
;-------------------------------------------------------------------------------
```

Result = **InStr**(Start, Source String, Search String)

Start (optional)	An integer expression which is the starting position in "Source String" to begin searching for "Search String".
Source String	A string variable name which is the source string.
Search String	A string expression which is the string to find in "Source String".
Result	A direct or indirect integer variable which is the position of "Search String" within "Source String". 0 = search string not found n = position of search string within source string

Description

The InStr command returns the position of the Search String within the Source String beginning at the Start position in the source string. If the start parameter is not specified the search begins at the first position in the source string. The first character in the source string is at position one, the second character is at position two, etc. If the search string is not found within the source string then the result will be zero. Upper and lower case letters are not equal, i.e. string "Moo" is not the same as string "moo".

```
;-------------------------------------------------------------------------------
;example "InStr" statements

  MyString = "This is a test"
  J = InStr(MyString, "t")          ;J = 11
  J = InStr(MyString, "is")         ;J = 3
  J = InStr(4, MyString, "is")      ;J = 6
;-------------------------------------------------------------------------------
;remove "leading zeros" from MyString

  MyString = "0005.3"
  J = InStr(MyString, "0")
  While J = 1 And len(MyString) > 1
     MyString = Mid(MyString, 2)
     J = InStr(MyString, "0")
  Wend
;-------------------------------------------------------------------------------
;read comma separated variables (CSV)

  MyData = "    5.3,    2.3e-4   ,   09.25   "
  For J = 0 To 2
     MyData = trim(MyData)
     Data[J] = csf(MyData)
     K = InStr(MyData, ",")
     MyData = Mid(MyData, K + 1)
  Next J
;-------------------------------------------------------------------------------
```

Integer Name, Default Value, NV Flag, Modbus Address

Name	The name of the integer variable. To define an integer array include the array size in parenthesis after the name.
Default Value	The integer will be set to this value at reset.
NV Flag (optional)	Nonvolatile (NV) variables retain their value when the power is off. Check the device data sheet to see if NV vars are supported. The Program command must include the +n option in order to use the NV flag option. Both individual variables and arrays can be designated as nonvolatile. 0 = variable is volatile 1 = variable is nonvolatile
Modbus Address (optional)	An integer constant which is the Modbus slave address of the variable. The address is only used when serial port 1 (UART1) is set to Modbus slave mode. The integer can be treated as 16 bit or 32 bit for Modbus communication. Add the "+" suffix to the address to designate the integer as 32 bits for Modbus. When defining an array the Modbus addresses are incremented for each variable in the array. Modbus slave addresses must be unique. Range is 1 to 4095.

Description

The Integer and Dim commands are used to define integer variables. A Microva BASIC integer is a 32 bit signed integer value with a range of -2,147,483,648 to +2,147,483,647 (0 to 0xFFFF FFFF). When defining an integer array include the size of the array directly after the array name. For example, to define an array named "MyArray" with 32 variables and a default value of 0xFF the statement would be: Integer MyArray(32), 0xFF Elements in the array are referred to by index. Use square brackets when referring to the individual element. The first element in the array is element 0 and the last element is (array size - 1). So, for this example, the elements are: MyArray[0] to MyArray[31]. The kernel firmware sets all elements in the array to the default value at reset/power on.

Nonvolatile (NV) Variables

Nonvolatile variables retain their value when the power is shut off. The NV Flag option can be used for devices that include the EERAM memory for nonvolatile variables. There is no difference between volatile and nonvolatile variables during normal program execution, i.e. there is no speed penalty for using nonvolatile variables. If the Program command includes the +n option then the NV flags must be included for each Integer variable defined. If the Program command does not include the +n option then the NV flags should not be included. Each nonvolatile integer variable uses one EERAM address. So a nonvolatile integer array of 10 elements would use 10 EERAM addresses. See the NvVarsLoad command page for more information on nonvolatile variables.

Modbus Address

The Modbus address is optional. The address must be written after the NV flag option if that option is included. The address is only used in Modbus slave mode. All Modbus addresses must be unique. Integer variable "change bits" are updated three ways (see the explanation for "change bits" at the beginning of this document):

1. By the application program running in the device.
2. When the device is in Modbus slave mode: by the Modbus master when the master writes to the variable.
3. When the device is in Modbus master mode: when a variable is read from a Modbus slave node using the Modbus command.

Modbus slave addresses are stored in a ROM table. The size of the table is equal to 6 times the highest Modbus address, i.e. if the highest address is 250, then the table size = 1500 bytes. Therefore, Modbus slave addresses should be assigned starting with address one and increasing as needed (Modbus address 0 is reserved for the node identification string). For that reason, when defining an Integer variable, the maximum Modbus slave address allowed is 4095. However, in Modbus master mode any address can be used (the Modbus address range is 0 to 65,535).

Integer variables can be treated as 16 bit or 32 bit values for Modbus communication. One Modbus address is needed for a 16 bit integer. Two addresses are needed for a 32 bit integer. Add the "+" suffix to the address to treat the variable as a 32 bit integer in Modbus slave mode. If the integer is declared as 16 bits, when the Modbus master reads the integer only the lower 16 bits of the integer value will be transmitted. Likewise, if the master writes to the 16 bit integer address only the lower 16 bits in the integer are written (the upper 16 bits are not changed). The Integer variable Modbus address is not used when the device is in Modbus master mode because in master mode the Modbus command sets the addresses that will be read or written. Note that Integer variables are always treated as 32 bit values in Microva BASIC regardless of the assigned Modbus address (with or without the "+" suffix). See the Modbus command for more information.

Result = **KeyIn**(Integer Expression)

Integer Expression	When the integer expression equates to 0 the touch screen X position is returned. When the integer expression equates to 1 the touch screen Y position is returned. The integer expression is not used when reading a keypad but must be included.
Result	Integer equal to the key code or the X or the Y touch screen position.

Description

The KeyIn function returns the key code from a keypad or the X or Y position from a touch screen.

Keypad

The standard Microva BASIC keypad consists of just four keys: UP, DOWN, PLUS and MINUS with key codes: UP=8, DOWN=4, PLUS=2, MINUS=1. The KeyIn function scans the keypad and returns the key code in under a microsecond. The integer expression is not used for keypads, however, it must be included. Typically, the UP/DOWN keys are used to move the cursor on the display and the PLUS/MINUS keys are used to increment or decrement a variable which is being edited. The keypad is usually read 20 times per second in order to "debounce" the switches as shown in the example below.

Touch Screen

The KeyIn function uses the CPU's analog-to-digital converter (the same ADC used by the AIN function) to read the X and Y finger touch position on the touch screen. Therefore, the KeyIn function execution time is around 3 to 5 microseconds. The function returns either the X position or the Y position depending on the function parameter. The X and Y positions must be read alternately. The range of X and Y depends on the ADC resolution which depends on the firmware version. See the AIN command page for the ADC resolutions.

Ideally, for a 10 bit ADC, the X and Y values will range from 0 to 1023 where 0 is the left/top of the display and 1023 is the right/bottom of the display. However, actual touch screens have a dead zone around the display perimeter so that the returned X and Y values will range from around 100 for the left/top to around 900 for the right/bottom. And these X/Y values will be slightly different for every display. Also, even while pressing the exact same position on the touch screen, each time the KeyIn function is called it will return a value that will vary from the previous reading by about one or two counts (plus or minus).

```
;-------------------------------------------------------------------------------
;reset the hardware if the user presses the keypad UP and MINUS keys together
 Constant UP=b3, DOWN=b2, PLUS=b1, MINUS=b0
 If Timer50ms Changes Then KeyCode = keyin(0)
 If KeyCode ChangesTo UP + MINUS Then ResetCPU
;-------------------------------------------------------------------------------
;get touch X and Y and scale the result for a 320 x 240 display
  If Timer10ms Changes Then
     TC = 1 - TC
     If TC = 0 Then
        TX0 = TX1
        TX1 = keyin(0)
     Else
        TY0 = TY1
        TY1 = keyin(1)
     End If
     If TX1 > 106 And TY1 > 110 And abs(TX1-TX0) < 5 And abs(TY1-TY0) < 5 Then
        TX = Interpolate(TX1, 106, 0, 935, 319)
        TY = Interpolate(TY1, 110, 0, 895, 239)
     End If
  End If
;-------------------------------------------------------------------------------
```

String Result = **LCase**(String Name)

String Name	The name of a string variable (not a string expression).
String Result	A string which is a copy of the string in "String Name" except with all lower case letters.

Description

The LCase (**L**ower **Case**) function converts all upper case letters in the source string to lower case. See the ANSI character code table in the appendix for upper and lower case letters. Use the UCase function to convert lower case letters to upper case.

```
;-------------------------------------------------------------------------------
;convert MyString to all lower case

  MyString = "Hello World!"
  MyString = lcase(MyString)     ;MyString = "hello world!"
;-------------------------------------------------------------------------------
```

Result = **Len**(String Name)
Result = **Lenx**(String Name)

String Name	The name of a string variable (not a string expression).
Result	Integer result.

Description

The Len (**Len**gth) function returns the number of characters in the string. Non-printable control codes are included in the result. See the list of ANSI character codes in the appendix. If the string is empty the returned value is 0.

The Lenx function returns the width of the text in pixels if the string was printed to the display using the active font. The active font is the font whose address is in FontPntr. Lenx is only used for devices with a graphic display. If there are multiple lines of text in the string by using the carriage return (CR) control code then the value returned will be the width of the widest line of text in the string.

```
;-------------------------------------------------------------------------------
  MyString = chr(NC) + chr(blue) + "Hello" + chr(CR)
  J = len(MyString)                  ;J = 8
  J = lenx(MyString)                 ;J = width in pixels of "Hello" using active font
;-------------------------------------------------------------------------------
```

Line X0, Y0, X1, Y1, Color, Line Width

X0, Y0	X0 and Y0 are integer expressions which equate to the position in the display buffer of one end of the line.
X1, Y1	X1 and Y1 are integer expressions which equate to the position in the display buffer of one end of the line.
Color	An integer expression which equates to the color index of the line.
Line Width	An integer expression which equates to the line width. Range is 1 to 33.

Description

The Line command draws a line to the display buffer from position X0, Y0 to position X1, Y1. The line width can range from 1 to 33 pixels and is centered on the line end points. The application software must make sure the line is drawn within the display buffer. For example, if the line starts at Y position 10 and is drawn with width = 30 then the line will draw past the display buffer RAM and potentially cause the program to crash.

```
;-------------------------------------------------------------------------------
;draw a line to the display buffer using color index = blue

 Line  125, 40, 30, 150, Blue, 5
 Refresh
;-------------------------------------------------------------------------------
```

String Result = LineIn(Address)
String Result = LineIn#(Address)

Address	An integer expression which equates to an address in RAM or ROM which is the start of a line of ANSI characters (text) which will be returned as a string.
String Result	A string which is one line of ANSI characters (text).

Description

The LineIn function is used to convert raw ANSI character (byte) data from an integer/byte array in RAM or ROM into the Microva BASIC string format. The Microva BASIC string format consists of an array of character (byte) data where the first byte indicates the number of characters in the string. See the String command page for an example. The LineIn function and the LineOut command are used to manipulate arrays of strings. The String command is used to create a single string variable and the Table command can be used to create an array of strings in ROM. However, there is also a need to manipulate string arrays in RAM. Often string (text) data consists of an array of byte data. For example, when a text file is read from a memory card into RAM using the FileRead command (instead of the FileReadLine command) the text data will consist of a continuous block of ANSI character codes (bytes) with no easy way to determine where the ends of each line of text is. The LineIn function is used to extract the string data from the byte data stored in the integer array. LineIn recognizes two types of text string terminators: EOL and null.

Null Terminated Text

A null terminated text string ends with zero, i.e. character code 0x00. The LineIn function returns all characters up to, but not including, the null terminator.

EOL (End of Line) Terminated Text

Text created using a text editor (such as Notepad) will have the "end of line" (EOL) designator after every line of text. The EOL designator is "carriage return" (ANSI code 0x0D) or "line feed" (ANSI code 0x0A) or carriage return and line feed together (0x0D followed by 0x0A). The EOL for Windows text files is CR + LF. The EOL for Linux text files is LF. The LineIn function returns all characters up to and including the EOL.

LineIn returns a maximum of 255 ANSI characters. The destination string must be large enough to receive the full line of text or the excess characters will be truncated. Also, keep in mind that sometimes the last line of text in a text file does not include the EOL since the EOL is only used to indicate where the line ends. If the last line of text in the byte data array does not include the EOL then the LineIn function will continue searching for the EOL past the actual data until either the 255 character limit is reached or the EOL is found or a zero is found. The LineOut command is the complement of the LineIn function and is used to convert a Microva BASIC string into raw ANSI character/byte data.

LineIn vs LineIn#

The difference between LineIn and LineIn# is LineIn returns a line of text by searching for the EOL or null terminator, however, LineIn# does not search for the line terminator. LineIn# assumes the string is already in the correct Microva BASIC string format, i.e. the first byte at the source address is the number of characters in the string and the following bytes are the ANSI character code data. See the FileList command for an example of why the LineIn# function is useful.

```
;-------------------------------------------------------------------------------
;assume integer array MyData() consists of just two lines:
;   "Hello" + CR + LF
;   "World" + 0

  MyText = linein(addr(MyData))        ;MyText = "Hello" + CR + LF
  MyText = linein(addr(MyData)+7)      ;MyText = "World"
  MyText = linein(addr(MyData)+9)      ;MyText = "rld"
;-------------------------------------------------------------------------------
```

LineOut String Expression, Address
LineOut# String Expression, Address

String Expression	The string to convert into raw ANSI character (byte) data and save to integer array.
Address	An integer expression which is an address in an integer array in RAM to store the converted string data.

Description

The LineOut command is used to convert a Microva BASIC string into raw ANSI character (byte) data and save the data to an integer array. The LineIn function is the complement of the LineOut command. See the LineIn function for an explanation of why it is useful to be able to convert from raw byte data into Microva BASIC string variable format. Normally, string data (text) is saved to a string variable. However, when working with an array of strings which must be modified, it is necessary to be able to save the string data to an integer variable array in RAM. The Table command can also save an array of strings, however, the string data in the table is in ROM and cannot be modified. For example, text data saved to a file on the memory card must be in raw ANSI character format, i.e. raw byte data, not Microva BASIC string format. The LineOut command is used to perform the conversion. Command LineOut simply copies the string data from the string expression to the destination address without the "number of characters" byte which is the first byte in a Microva BASIC string. The destination address must be an address in RAM. The EOL (end of line) characters (explained on the LineIn function page) are not automatically appended to the string expression, however, they will be included if they are part of the string. See the appendix for a table of ANSI character codes.

LineOut vs LineOut#

The difference between the LineOut and LineOut# commands is the LineOut command skips the first byte in the string but the LineOut# command includes the first byte, i.e. the entire Microva BASIC format string is copied.

```
;-------------------------------------------------------------------------------
;search string array "LastNames" for "Smith", replace with "Jones"
;assumes array "LastNames" uses 16 chars per last name (15 chars plus CR)

 For J = 1 To NumNames
    K = addr(LastNames) + 16 * (J - 1)
    Name = linein(K)
    If Name = "Smith" + chr(CR) Then LineOut "Jones" + chr(CR), K
 Next J
;-------------------------------------------------------------------------------
```

LoadVarDefs

Description

The kernel firmware sets all integer, float and string variables to their default value at reset/power on. The default values are defined when the variables are created. See the Dim, Integer, Float and String commands for an explanation of the variable default value. The LoadVarDefs command resets all variables to their default values and clears all variable "change bits". See the explanation of change bits at the beginning of this manual. This command has no parameters.

String Result = **Mid**(String Name, Start, Length)

String Name	A string variable name which is the source string.
Start	An integer expression which is the starting location in the source string.
Length (optional)	An integer expression which is the length of the string to return in "String Result".
String Result	A string which is a portion of the source string "String Name".

Description

The Mid command returns a portion of the source string beginning at the Start position in the source string. The Length parameter determines the number of characters returned. If length is not specified or if the number of characters remaining in the string (from the starting position) is less than length, then the string result will include the entire remainder of the source string beginning at the start position. The first character in the source string is at position one, the second character is position two, etc. The command just returns (i.e. does nothing) if start <= 0 or if length <= 0 or if start > (the length of the source string). The Mid command can be used repeatedly in a string variable assignment (as shown below), however, it cannot be used within another command or function that requires a string parameter. For example, Print Mid(MyString, 2, 4) is not permitted.

```
;-------------------------------------------------------------------------------
;return "cow" from "The cow jumped"
  MyString = "The cow jumped"
  AnimalName = Mid(MyString, 5, 3)

;change "The cow jumped" to "The horse jumped"
  MyString = Mid(MyString, 1, 4) + "horse" + Mid(MyString, 8)
;-------------------------------------------------------------------------------
```

Result = **Modbus1**(Function Code, Node Num, Address, Param1, Param2, ..., Param14)
Result = **Modbus2**(Function Code, Node Num, Address, Param1, Param2, ..., Param14)

Function Code	Code Name Description 0x03 RdVars read 16 bit vars (1 min, 125 max) 0x04 RdInputs read 16 bit vars (1 min, 125 max) 0x06 WrVar write a single 16 bit var 0x10 WrVars write 16 bit vars (1 min, 123 max) 0x03 RdVars32 read 32 bit vars (1 min, 62 max) 0x10 WrVars32 write 32 bit vars (1 min, 61 max)
Node Num	An integer expression which is the Modbus slave device node number for communication. The range is from 1 to 247. For write function codes the node number can be set to zero to indicate "broadcast".
Address	An integer expression which is the address of the first variable in the packet to read or write ("Param1"). The range is from 0 to 65535. The addresses are incremented by one or two for each additional parameter.
Param1-Param14	Integer or float variable names or expressions. To read or write an array set Param1 to the array name (array can be integer, float or string).
Result	A direct integer variable which holds the result of the command: 0 = normal response received from slave (i.e. no error) -1 = no response received from slave within "max response time" -2 = partial response received from slave but response packet was not completed within "max response time" -3 = full packet response received but with CRC or communication error -4 = error exception response: slave doesn't recognize function code -5 = error exception response: slave illegal address or illegal data -6 = all other errors

Description

The Modbus commands are used to read or write data to Modbus compatible devices made by Microva or other manufacturers. Command Modbus1 uses serial port 1. Command Modbus2 uses serial port 2. Modbus is a master/slave (client/server) type network which allows one master at a time on the network and up to 247 slaves. Each slave must be assigned a unique node number. The master has no node number. Modbus data packets start with the node number that the master wishes to communicate with and the starting address for the variables to be read or written. The addresses are incremented for each variable in the packet. Note, these are Modbus virtual addresses that exist in the slave, the actual addresses of the variables in RAM memory, in both the master and the slave, do not need to be sequential and are unrelated to the Modbus address. If any of the addresses do not exist in the slave device the slave will return an error code. Modbus data packets from the master or the slave device end with a CRC code which is used to determine if the packet data was corrupted. The slave must respond within the "maximum response time" window. See the Open SerialPort command for an explanation of how to set the Modbus max response time. For a more detailed explanation of Modbus see the Microva application notes *"AN130 What is Modbus?"* and *"AN132 Microva Modbus RTU Implementation"*.

The Modbus commands can read or write 16 bit or 32 bit data. One address is needed for 16 bit data. Two addresses are used for 32 bit data. Integer variables can be treated as 16 bit or 32 bit. Float variables are 32 bit. Usually, 16 bit data is read or written using the RdVars or WrVars functions. The RdVars32 and WrVars32 functions are used to read and write 32 bit data. For 32 bit data, Microva BASIC assumes the data order in the Modbus packet is "big endian", i.e. the first byte read or written is the most significant byte followed by the less and less significant bytes. That means the first address in the packet is for the upper 16 bits of Param1 and the next address is for the lower 16 bits of Param1. For example, the command J = Modbus2(RdVars32, 5, 8, MyData) reads the upper 16 bits of MyData from address 8 in node number 5 and the lower 16 bits from address 9. If MyData was specified as a 16 bit value in the slave device then the exact same command can be used except the function code name would be changed to RdVars and only address 8 would be read. And, since Microva BASIC integers are 32 bits, only the lower 16 bits will be updated in the integer variable MyData. Therefore, MyData will range from 0 to 65,535. However, if MyData can be negative, for example if MyData was a temperature reading, then it would need to be converted from the unsigned 16 bit value read over the Modbus to the signed 32 bit value used in Microva BASIC. The example on the next page shows how to do the conversion.

The Modbus commands can be used to read or write individual variables or an entire array of either 16 bit or 32 bit values. For individual vars, the number of vars read or written is just the number of parameters included in the command. To read or write an entire array just use the array name for the first parameter (and there can only be 1 parameter). For example, if MyArray was a float array with four elements then the Modbus command J = Modbus2(RdVars32, 5, 22, MyArray) would read addresses 22 thru 29 from

Result = **Modbus1**(Function Code, Node Num, Address, Param1, Param2, ..., Param14)
Result = **Modbus2**(Function Code, Node Num, Address, Param1, Param2, ..., Param14)

Description (continued)

the slave device node 5. For 16 bit variables the maximum array size allowed in one Modbus read command is 125 and for the Modbus write command the maximum array size is 123. For 32 bit variables the maximum array size for a read is 62 and for a write the maximum is 61. The variable's "change bit" is updated in the master when the variable data is read from the slave. Therefore, the Modbus master can be programmed to take action only when a variable changes in the slave device. Likewise, the change bits in the slave are updated when variables in the slave are written by the master. Variable change bits are described at the beginning of this document. Node number 0 is the "broadcast" node number and can only be used for write function codes. There is no response from the slaves when data is broadcast so there is no way to know if the data was properly received. Also, after sending the broadcast command, the application program must allow a "bus idle" time of at least 4 character times before sending the next Modbus command as required in the Modbus specification. This bus idle time is inserted automatically for non-broadcast commands since non-broadcast commands always require a slave response.

Read and Write Strings

Strings can be read or written using the RdVars32 and WrVars32 function codes since strings are treated as just ordinary 32 bit integer arrays. The maximum string size = 248 bytes for the RdVars32 command and the maximum string size = 244 bytes for the WrVars32 command. The entire string array is read or written even if the actual string length in the array is shorter. For example, if MyString length = 80 bytes but the string in MyString is "Hello", when the MyString array is read or written all 80 bytes are sent. Therefore, when reading or writing strings, the string array size in the slave must be the same size or larger than string array size in the master. The number of Modbus addresses required for the string = string array size / 2.

The Node Identification String

All Modbus devices made by Microva reserve address 0 for the node identification string. The command to read the node ID string from any Microva Modbus device is:

```
Dim  NodeID(48)  As String
J = Modbus2(RdVars32, NodeNum, 0, NodeID)
```

The variable address must equal 0 and the destination string (NodeID in the example but any name can be used) must have a size of 48 bytes. The format of the returned node ID string is: Firmware Version + "," + Application Program Name. For example, a typical node ID string read from a Microva model MB122 device is: `"2.3.1,MB122 1.8"` If the node ID string is longer than 48 bytes it will be truncated to fit (actually 47 bytes because the first byte in the string indicates the string length). The node ID can be read from devices in Program mode or Run mode.

```
;-------------------------------------------------------------------------------------------
J=ModBus1(RdVars,    7, 2, VA7, MyVar, VB3, V1, MD6) ;read 5 16 bit vars from node 7, addrs 2 to 6
J=ModBus1(RdVars32, 7, 2, Var1, Var2, Var3)          ;read 3 32 bit vars, addrs = 2/3, 4/5, 6/7
J=ModBus1(RdVars, 5, 0, Time[0], Time[1], Time[2])   ;Error: can't read individual array vars
J=ModBus1(WrVars, 7, 5, abs(MyData[K]), MyData[M/2]);but individual array vars are OK for write
J=ModBus1(RdVars, 5, 2, MyArray)                     ;use array name to rd/wr 16 or 32 bit arrays
J=ModBus1(RdVars, 5, 2, MyArray1, MyArray2)          ;Error: only 1 array name per Modbus statement
J=ModBus1(WrVars32, 5, 2, MyString)                  ;read or write a string using array name
J=ModBus1(WrVars32, 5, 4, "Hello World")             ;Error: strings must use array name
J=ModBus1(RdVars32, 5, 0, NodeID)                    ;read Node ID string (address must = 0)
J=ModBus1(WrVars, N+3, M*K, VB5, 5*DG, (Min-5)*60)   ;node num, addrs, params can use expressions
;-------------------------------------------------------------------------------------------
;convert an unsigned 16 bit integer read over Modbus to a signed 32 bit integer

  J = Modbus2(RdVars, 1, 5, uTemp)
  Temp = (uTemp } 15) * 0xFFFF0000 | uTemp
;-------------------------------------------------------------------------------------------
```

String Result = **Ndf**(Integer Expression)

Integer Expression	An integer expression which equates to the new decimal format and will be written to the kernel system variable DecForm. The decimal format is only used by the CISF or CFSF functions. DecForm is a hex value that indicates the number of digits to display to the left of the decimal point and the number of digits to display to the right of the decimal point. The value must be chosen so that the maximum number of characters in the string result is 10 (including the decimal point and the minus sign if the number is negative). For floating point numbers the decimal format must include at least one digit to the right of the decimal point.
String Result	The NDF function returns the null string.

Description

The NDF (**N**ew **D**ecimal **F**ormat) function writes a new decimal format value to the system variable DecForm. The CISF and CFSF functions use the decimal format to convert an integer or float value into a string with a fixed decimal position. That makes it easy to display fixed point integer or float data. Use a hex value to define the decimal format. The hex value indicates the number of digits to display to the left of the decimal point and the number of digits to display to the right of the decimal point. The hex value must be chosen so that the maximum number of characters in the string result is 10 (including the decimal point character and the minus sign character if the number is negative).

For example, suppose a temperature sensor returns an integer value equal to the temperature in Celsius x 10 (so 2753 = 275.3C). This temperature can be displayed using the statement: Print "Temp=" + ndf(0x31) + cisf(Temp) + "C"
The 0x31 decimal format means variable Temp will be displayed as a fixed point integer with 3 digits max on the left of the decimal point and 1 digit on the right of the decimal point. Therefore the 0x31 number string will consist of a maximum of five characters (the decimal point is a character). Leading zeros are removed in the formatted string but trailing zeros are retained. So the length of the formatted string will range from 3 to 5 characters. Also, the decimal form determines the range of the number that can be displayed. The 0x31 decimal format means variable Temp can display a number ranging from -99.9 to 999.9. The minus sign is considered 1 character in the string, the plus sign is assumed. Here are some example values for Temp with DecForm = 0x31 shown with the CISF function formatted string result:

```
 Temp     Formatted   Number
 Value     String     Chars      Comment
-------   ---------   ------   ------------------------------------------------
   0         0.0         3      leading zeros are removed except for the ones position
   123       12.3        4
   1257      125.7       5
  -857       -85.7       5
  -1255      ---.-       5      most negative number with 0x31 format is -999 (-99.9)
  11875      +++.+       5      most positive number with 0x31 format is 9999 (999.9)
```

To display an integer number with no decimal point but with a maximum fixed number of characters set the number of digits to the right of the decimal to 0. For example, decimal form 0x50 displays from 1 to 5 characters with a range from -9999 to 99999. For floating point numbers DecForm must have at least one number to the right of the decimal. The CFSF function returns the formatted string "rounded up" to fit the decimal format. For example, if DecForm = 0x22 and K = 3.1275 then the formatted string returned from CFSF(K) would be "3.13". The value in K is not changed, its still 3.1275, only the string result is rounded up.

Once DecForm is set by the NDF function its value will not change until another NDF function call changes it. Function CIS actually just calls the CISF function except it temporarily sets DecForm = 0xA0. And function CFS just calls the CFSF function except it temporarily sets DecForm = 0x63. Note that DecForm = 0x20 is the only form that will not have the leading zero removed. That's so time values display correctly like "12:05" not "12:5". Sometimes it is necessary to be able to select the decimal format at "run time". In this case, instead of using a constant decimal form as in the examples above, the decimal form can be defined by another variable.

```
;-------------------------------------------------------------------------------------------
;display Pi with 2, 3 and 6 decimal places
;  result is:   3.14, 3.142, 3.141593

  Print ndf(0x12) + cfsf(Pi) + ", " + ndf(0x13) + cfsf(Pi) + ", " + ndf(0x16) + cfsf(Pi)
;-------------------------------------------------------------------------------------------
```

Result = **Not**(Integer Expression)

Integer Expression	Integer expression which will be converted into its ones complement value.
Result	Integer equal to the ones complement value of integer expression.

Description

The Not function returns the ones complement (i.e. bit inversion) of an integer expression.

```
;-------------------------------------------------------------------------------
  J = 0x8AC92E85  ;J = 1000 1010 1100 1001 0010 1110 1000 0101
  K = not(J)      ;K = 0111 0101 0011 0110 1101 0001 0111 1010
;-------------------------------------------------------------------------------
```

Result = **Nva**(Name)

Name	The name of a nonvolatile integer, float or string variable or array.
Result	Integer equal to the starting address of the nonvolatile variable or array in EERAM. The range is 0 to 511. Result is -1 if the named variable is volatile.

Description

The Nva function returns the address of the nonvolatile variable or array in EERAM. See the NvVarsLoad and NvVarsSave commands for a detailed description of nonvolatile variables.

```
;-------------------------------------------------------------------------------
  K = nva(MyVar)   ;K = the address of MyVar in EERAM
;-------------------------------------------------------------------------------
```

NvVarsLoad Name1, Name2, Name3, etc.
NvVarsSave Name1, Name2, Name3, etc.
NvVarsSave# Data, Address

Name (optional)	The name of a nonvolatile integer, float or string variable or array to load or save from or to EERAM. The order does not matter. Max of 16 names per command. If no names are included then all nonvolatile variables and arrays will be loaded or saved.
Data	An integer or float expression which equates to the data to write to EERAM.
Address	An integer expression which equates to the EERAM address to write. The EERAM address range is 0 to 511.

Description

Microva BASIC supports nonvolatile (NV) variables. NV variables retain their value when the power is turned off which is useful for storing calibration values, program settings, run times, etc. NV variables and arrays are declared when the variables or arrays are defined using the Integer, Float or String commands. NV variables and arrays require EERAM memory. Check the device data sheet to determine if NV variables are supported. The kernel firmware automatically loads all variables with their default value when the device is powered on or reset. Subsequently, in the application program RESET subroutine, NV variables are loaded from the external EERAM memory using the NvVarsLoad command. The NvVarsLoad command reads the previously stored data from EERAM to the variable or array in RAM. The NvVarsSave command writes the variable or array data from RAM to EERAM. EERAM can be read or written an unlimited number of times. There is a maximum of 512 NV variables. When the NvVarsLoad or NvVarsSave command has no name parameters then all NV variables will be loaded or saved. Each individual variable is loaded or saved to EERAM in under 100us. Therefore, depending on how many NV vars there are in the program, loading or saving all NV vars can take many milliseconds (200 vars would take around 20ms). Therefore, usually all the NV vars are loaded in the reset subroutine, however, NV vars are saved one at a time when the variable is edited. When a new application program is loaded into the device the EERAM is automatically written with the default values for all the declared NV variables and arrays. Also, when the device is in Modbus slave mode, any NV variables written by the Modbus master will automatically be saved to EERAM. These are the only two cases where the kernel firmware writes NV variables to EERAM. In all other cases the application program is expected to read and write the NV variables as needed.

NvVarsSave vs NvVarsSave#

The NvVarsSave command saves one or more variables using the variable or array name. The NvVarsSave# command saves only one variable using integer or float data and the EERAM address, not the variable name. Therefore, the NvVarsSave# command can save NV variables "indirectly" within a subroutine. Use the NVA function to find the nonvolatile variable's address in EERAM. The example below shows why this is useful.

Operator Entered vs Program Generated NV Vars

Nonvolatile variables are typically used to save program settings input by the operator using the HMI (**H**uman **M**achine **I**nterface, i.e. the display and keypad or touch screen). For example, the operator might enter the temperature at which an alarm relay should turn on. As the temperature alarm value is edited on the display it is also saved using NvVarsSave. However, some NV vars are not input by a human operator. For example, the run time of a motor might be refreshed every second in the program. Therefore, the nonvolatile integer variable which holds the total run time in seconds is saved to the EERAM every second. But the power can fail at any time and if it happens to fail exactly when the integer is being saved the data could be corrupted. Therefore, for this type of program generated NV var, to prevent possibly corrupted data, the application program should check the power supply voltage just before saving the variable. Saving a single variable takes under 100us so if the external power supply voltage is high enough, even if the power fails when the variable is being saved, the data will still have enough time to be written to the EERAM. Microva devices that include EERAM usually also include circuitry so the external power supply voltage can be read in the application program for these type of program generated NV variables.

```
;-------------------------------------------------------------------------------
; save individual and array NV vars by name

  NvVarsSave  AlarmTemp, MaxPressure, SettingsArray
;-------------------------------------------------------------------------------
; save one nonvolatile variable in an array

  NvVarsSave#  SensorSpan[Channel], nva(SensorSpan) + Channel
;-------------------------------------------------------------------------------
```

Open I2CPort SCL Pin Num, SDA Pin Num, Baud Rate, Delay

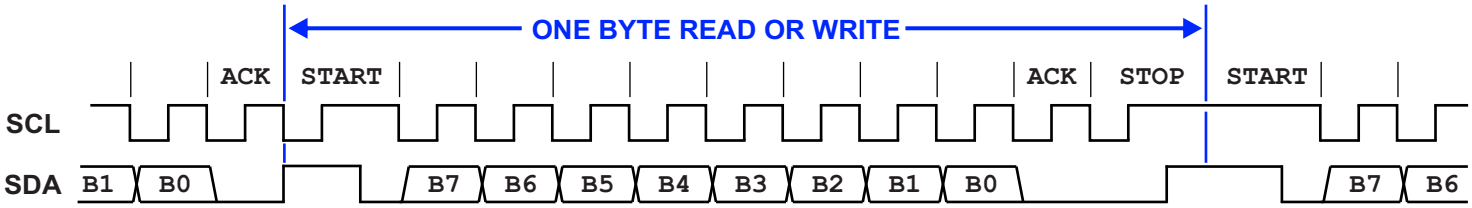
SCL Pin Num	An integer expression which equates to the SCL pin number.
SDA Pin Num	An integer expression which equates to the SDA pin number.
Baud Rate	An integer expression which sets the I2C clock frequency in kHz. For example, 400 = 400kHz, 1000 = 1MHz.
Delay (optional)	An integer expression which sets the delay time (in microseconds) for the delay which will be automatically inserted after every ACK or NAK clock. The default value is 0.

Description

The I2C bus is used to transmit and receive synchronous, i.e. with a clock, serial data. I2C requires two pins, they are:

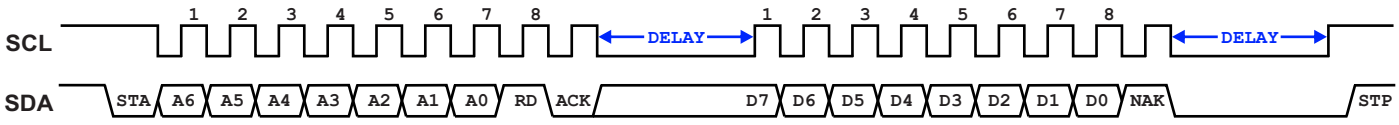
SCL the I2C clock pin
SDA the I2C data pin

Any two pins on the same port can be used for I2C, i.e. both pins must be on PortA or PortB or PortC, etc. The Open I2CPort command sets both pins to be open drain outputs with logic level 1. The port is opened in I2C master mode (slave mode is not supported). The I2C device must have pull up resistors on both SCL and SDA. The Baud Rate parameter sets the clock speed to any frequency. The pull up resistors should match the fastest baud rate expected. Typical values are 10K ohm for 100kHz, 4.7K ohm for 400kHz and 2K ohm for 1MHz. Use the I2C function to communicate with the I2C device. The communication is implemented in firmware so the CPU's I2C hardware module is not used. That means there is no limit to the number of pins that can be used for I2C peripheral devices. The I2C function communicates using the most recent Open I2CPort command parameters. To change to a different set of I2C pins, just use a new Open I2CPort command. There is no Close I2CPort command needed because the I2C hardware module is not used. If required, the SCL and SDA pins can be returned to normal logic I/O, i.e. not open drain, using the Pin commands. See the I2C function page for a more detailed explanation. The drawing shows a single data byte transfer (the first START command is actually a "re-start" since the previous command ended with ACK instead of STOP).



Clock Stretching

Some I2C devices hold the clock line low to indicate to the master that they are not ready to accept new commands or data. This is known as clock stretching. The I2C function does not support clock stretching on any arbitrary clock pulse, however, it does support clock stretching for the clock directly after the ACK or NAK clock which is the most common way that devices implement clock stretching. This is done with the Delay parameter. The delay is automatically inserted after every ACK or NAK clock as shown in the drawing below. Check the I2C device data sheet for the maximum clock stretch delay time. The default value for Delay is zero, i.e. no delay is inserted.




```
;-------------------------------------------------------------------------
;open I2C with SCL = PD0, SDA = PD1 and frequency = 1MHz

  Open I2CPort  PD0, PD1, 1000
;-------------------------------------------------------------------------
```

Open SerialPort1 Mode, Data Form, Baud Rate, Modbus Variable
Open SerialPort2 Mode, Data Form, Baud Rate, Modbus Variable

Mode	A constant which is the serial port mode. The predefined constants are:
	`SerialIO = 0`
	`ModbusMaster = 1`
	`ModbusSlave = 2 (for SerialPort1 only)`

Data Form	An integer expression which is the serial I/O data form. The data forms are:
	`0 = 10 bits/char, no parity, 1 stop bit (keyword 8N1)`
	`1 = 11 bits/char, no parity, 2 stop bits (keyword 8N2)`
	`2 = 11 bits/char, even parity, 1 stop bit (keyword 8E1)`
	`3 = 11 bits/char, odd parity, 1 stop bit (keyword 8O1)`

Baud Rate	An integer expression which is the serial I/O baud rate (keywords shown in blue). The baud rates are: `0 = 2400` `1 = 4800` `2 = 9600` `3 = 19200` `4 = 38400` `5 = 57600` `6 = 115200` `7 = 250000` `8 = 500000` `9 = 1000000`
Modbus Variable (optional)	In Modbus master mode: An integer expression which is the "maximum response time" in milliseconds. Range is 1 to 16,383.
	In Modbus slave mode: An integer expression which is the slave node number. Range is 1 to 247.

Description

The serial ports are used to transmit and receive asynchronous (i.e. no clock) serial data. The CPU has hardware modules dedicated to serial I/O called UARTs (Universal Asynchronous Receiver & Transmitter). The Open SerialPort1 command is used to open serial port 1 (UART1). The Open SerialPort2 command opens serial port 2 (UART2). The Modbus, SerialOut and SerIn commands are used to read and write data to the open serial port. The serial port receive pins (U1RX and U2RX) must be set to digital inputs using the PinType and PinDir commands before the port is opened. Likewise, the serial port transmit pins (U1TX and U2TX) must be set to digital outputs before the port is opened. The transmit pins should also be set to the idle or "mark" state (a logic 1) using the PinOut command before the port is opened. If the serial port is RS-485 compatible then the transmit enable pins (U1TE and U2TE) must also be set to digital outputs (with TX off, i.e. logic 0) before the port is opened. The kernel firmware automatically sets U1RX, U1TX and U1TE to the correct (idle) state at reset. However, the UART2 pins may not be in the correct state at reset depending on the hardware configuration (see the device data sheet). The Open SerialPort command takes control of the serial I/O pins. The Close SerialPort command turns off the UART and returns the RX/TX/TXE pins to control of the application program. Microva BASIC programmable hardware always includes serial port 1 since serial port 1 is used to load the application program into the device, but not all hardware includes serial port 2. Also, depending on the hardware model, each serial port might use raw logic level serial I/O, or true RS-232 level I/O or RS-485 level I/O.

Data Form

The Data Form parameter sets the serial communication data format. The data form can be entered using the keyword or indirectly using an integer variable or expression. All serial I/O data is transmitted/received as an 8 bit unsigned byte (range = 0 to 255). Each serial I/O "character" consists of a start bit + the 8 bit data + an optional parity bit + 1 or 2 stop bits. The parity bit is an error check bit inserted in the character which the receiver can use to help determine if the data was corrupted (at the expense of the extra bit per character). Network packet data based communication protocols such as Modbus add cyclic redundancy error check codes to each packet which are a more thorough method of error checking.

Baud Rate

The Baud Rate parameter sets the serial communication bit speed. The baud rate can be entered using the keyword or indirectly using an integer variable or expression. For example, at 9600 baud a bit is transmitted or received every 104 microseconds, i.e. 1/9600. Therefore, for data form 8N1 (which uses 10 bits per character) each character will take 1.04 milliseconds to send or receive.

Mode

The Mode parameter sets the communication mode for the serial port. Serial IO mode is used to send and receive text or binary data from any device that uses serial communication. Serial IO mode is also useful for sending text messages from the hardware to the Microva BASIC IDE connect screen (or to any computer terminal screen) for debugging or to view information from the device.

Open SerialPort1 Mode, Data Form, Baud Rate, Modbus Variable

Description (continued)

The Modbus Variable should not be included in Serial IO mode. See the SerialOut command page for more information. The Modbus master and slave modes are used for communication with Modbus devices. Modbus slave mode can only be opened on serial port 1. See the Modbus command page for more information.

Maximum Response Time

In Modbus master mode the "maximum response time" is the amount of time the Modbus master will wait after sending a request to the Modbus slave device before returning a communication error. The max response time depends on the baud rate, the data format, the size of the response packet and how long it takes for the slave device to finish a program loop. Typical values for max response time are 10ms for 250,000 baud rate and 130ms for 19,200 baud rate. The max response time should be kept as short as possible because the application program running in the master is paused while waiting for the slave to respond. Normally the slave responds in just a few milliseconds, however, if the slave device is offline the master program will be paused for max response time. Keep in mind that slave devices with a graphic display take longer to respond since refreshing the display in the slave device can take tens of milliseconds. The equation to set the max response time is:

Max Response Time = slave device app program max loop time + slave response packet time + bus idle time (4 characters)

See Microva application note *"AN132 Microva Modbus RTU Implementation"* for a more detailed description of the max response time. Also, the max response time can be changed between Modbus requests by just re-opening the serial port with the new max response value (the serial port does not need to be closed first).

Node Number

The "node number" is required when Modbus Slave mode is selected (for serial port 1 only). In Modbus slave mode a CPU interrupt program is enabled which receives Modbus packet data from the Modbus master in the background while the normal application program is running. The data is temporarily stored in a buffer until the kernel firmware can act on it at the end of every program loop (see the flow chart at the start of this manual). When the Modbus master writes to a variable in the slave device, if that variable in the slave is written with new data then the variable's "change bit" will be set in the slave device. If the variable in the slave is written to by the master but the data does not change then the variable's change bit will be cleared in the slave device. In that way the application program running in the slave device can know when new data was written to a variable by the master.

MbRxCounter

System variable MbRxCounter is reset at power on and incremented each time a valid, error free Modbus request is received to the node number or to node number 0 when Port 1 is opened in Modbus slave mode. Typically, the green LED on the slave node should blink each time a valid Modbus request packet is received from the master. The green LED should not blink if the request is addressed to a different node. That makes it easy to verify the communication setup and wiring between the Modbus master and slave device. The example below shows the software that should be included in the application program in order to blink the LED.

Modbus Internal Communication Settings

Typically, when the serial port is opened in Modbus slave mode, the hardware internal communication settings are used to set the node number, baud rate and data form. For Microva Modbus I/O devices these internal communication settings can be changed just using the PGM switch on the PCB, no computer is necessary. The settings can also be changed from the computer on the "connect" screen in the Microva BASIC Compiler software. The communication settings are preserved even if a new application program is written to the device. However, it is up to the application program whether or not the internal communication settings are used when opening the serial port. The example below shows how to open the serial port in Modbus slave mode using the internal communication settings. See the SaveComSettings command for information on the internal communication settings data format.

```
;-------------------------------------------------------------------------------
;open serial port using key words
   Open SerialPort1  ModbusSlave, 8N1, 250000, 1
;-------------------------------------------------------------------------------
;open serial port using the internal communication settings
   NodeNum  = sys(3) & 0xFF
   BaudRate = sys(3) } 10 & 0x0F
   DataForm = sys(3) } 8 & 0x03
   Open SerialPort1  ModbusSlave, DataForm, BaudRate, NodeNum
;-------------------------------------------------------------------------------
;blink the green LED briefly each time a valid Modbus request packet is received
   MbRxCount = rdi(MbRxCounter)
   If MbRxCount Changes Then GLedOffTime = Timer1ms + 15
   If Timer1ms < 100 Then GLedOffTime = 0
   If Timer1ms < GLedOffTime Then PinOut GLed:On Else PinOut GLed:Off
;-------------------------------------------------------------------------------
```

Open SpiPort Mode, Data Size, Baud

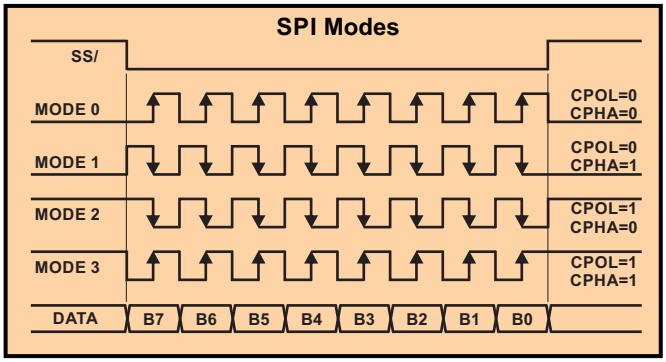
Mode	An integer expression which is the SPI port mode. Range is 0 to 3.
Data Size	An integer expression which is the number of bits read/written during each SPI data I/O transaction. The valid values are 8, 16 or 32 bits (i.e. 1, 2 or 4 bytes).
Baud	An integer expression which sets the SCLK frequency. The range is 0 to 511. $SCLK = \frac{PbClk}{2 * (BAUD + 1)}$ OR $BAUD = \left(\frac{PbClk}{2 * SCLK}\right) - 1$

Description

The SPI (Serial Peripheral Interface) bus is used to transmit and receive synchronous (i.e. with a clock) serial data. The CPU has one or more hardware modules used for SPI I/O. When the device includes the memory card socket one SPI module will be reserved for use by the kernel firmware for file I/O. Another SPI module is dedicated for use in the application program and can be opened using the Open SpiPort command. SPI I/O usually requires four pins, they are:

SCLK = the SPI serial clock
SS/ = the slave select (active low)
MOSI = the master output, slave input for writing to the slave device
MISO = the master input, slave output for reading from the slave device

Some slave peripheral devices can only be read (or written) in which case the MOSI (or MISO) pin is not required. The SPI hardware module only controls the SCLK, MOSI and MISO pins. The SPI slave device enable (SS/) is controlled by the application program. There can be several SPI slave devices which all share SCLK, MOSI and MISO but which have independent SS/ pins. At reset and in program mode all I/O pins are set to analog or digital inputs. The SPI pins SCLK, SS/ and MOSI must be set to digital outputs and MISO must be set to a digital input using the PinDir command before the port is opened.



The Open SpiPort command takes control of the SPI I/O pins. The Close SpiPort command turns off the SPI hardware and returns the SCLK, MOSI and MISO pins to control of the application program. The Open SpiPort command can be called even if the port is already opened, for example to change the SCLK frequency, however, be aware that the command briefly closes the port when writing the new Mode, Data Size and Baud parameters (i.e. SCLK and MOSI will briefly return to their values assigned in the application program). The Data Size parameter sets the number of bits read/written during each SPI I/O transaction. SPI data is read/written using the SPI function. The SPI function transmits and receives either the least significant byte, or the least significant two bytes or all four bytes from a single integer value, depending on the value of Data Size. The Baud parameter sets the SCLK frequency by dividing down the PbClk frequency. The PbClk frequency depends on the firmware version as shown in the table below. The SPIMode parameter sets the SCLK polarity (called "CPOL") and the data phase (called "CPHA"). The mode determines on which clock edge the data is changed and on which edge the data is "strobed" (i.e. read by the master or slave device). The figure above shows the four SPI modes for the four different SCLK and data phases. The data (MISO and MOSI) must change on the inactive edge of SCLK as shown. The SCLK strobe edge is shown with an arrow. Data is transferred starting with the most significant bit, i.e. bit 7 for 8 bit data, bit 15 for 16 bit data and bit 31 for 32 bit data. See the SPI function for more information and examples.

PbClk Frequencies

Firmware Version	PbClk
2.5	40MHz
3.4	100MHz
4.6	100MHz
5.2	40MHz

```
;-----------------------------------------------------------
;open SPI port (mode 1, 16 bit data, SCLK = PbClk / 10)
;  Note: SCLK, SS/ and MOSI pins must be set to outputs
;        and MISO must be set to an input

 Open SpiPort  1, 16, 4
;-----------------------------------------------------------
```

Result = **OpenSDHC**()

Result	A direct or indirect integer variable which holds the result of the command: n = total number of sectors on the memory card -1 = no response from memory card (card not installed). -2 = can't initialize card (not an SDHC memory card with FAT32 formatting) -3 = can't read sector 0 (not an SDHC memory card with FAT32 formatting) -4 = format error (format not FAT32, 512 bytes/sector, 2 FAT copies and 1 partition)

Description

Microva BASIC supports SDHC micro memory cards with FAT32 formatting. SDHC micro memory cards range from 4GB to 32GB. The memory card must have "SDHC" printed on the card. Standard SDHC memory cards (marked "SDHC") as well as SDHC cards marked "Ultra" or "Ultra Plus" or "Ultra High Speed", printed with "SDHC I" or "SDHC II" or "SDHC UHS-I" or "SDHC UHS-II" will work with Microva BASIC. The old style (under 4GB) standard "SD" cards will not work with Microva BASIC. Memory cards marked "SDXC" or "SDIO" also will not work with Microva BASIC.

Before file data can be read or written to the memory card the card must be "mounted" (i.e. initialized) using the OpenSDHC command. Also, if the memory card is removed from the socket or anytime the power is turned off, the card must be re-mounted before it can be used. The OpenSDHC command can take anywhere from a few milliseconds up to hundreds of milliseconds to complete, depending on the memory card brand and the card speed rating. The SDHC memory card socket includes a switch which indicates if a memory card is inserted (see the device data sheet). The switch input can be used as a quick way to determine if a memory card is inserted before executing the OpenSDHC command. The file I/O command descriptions in this document (such as FileRead, FileWrite, etc) include examples of how the file I/O command is used. These examples all assume the SDHC micro memory card has already been mounted using OpenSDHC.

Memory cards are addressed using "sectors". A sector is 512 bytes (128 float or integer variables). The sector data can be used as integer data or float data or text (string) data. If the card was successfully mounted, the OpenSDHC command will return the total number of sectors available for file I/O on the card. Note that a few percent of the memory card is reserved for use by the memory card itself. Therefore, the total number of sectors returned by the OpenSDHC command may not exactly match the memory card size. For example, a 4GB memory card may return the total number of sectors = 7725056 not 8388608 as expected.

```
;-------------------------------------------------------------------------------
;mount the memory card and append "Hello World" to "MyFile"

  J = OpenSDHC()
  J = FileOpen("MYFILE.TXT")
  J = FileAppend("Hello World")
;-------------------------------------------------------------------------------
```

PinDir PinNum:PinData, PinNum:PinData, PinNum:PinData, PinNum:PinData, . . .
PinIn PinNum:VarName, PinNum:VarName, PinNum:VarName, PinNum:VarName, . . .
PinOut PinNum:PinData, PinNum:PinData, PinNum:PinData, PinNum:PinData, . . .
PinType PinNum:PinData, PinNum:PinData, PinNum:PinData, PinNum:PinData, . . .

PinNum	An integer constant which is the pin number. Range is 0 to 127.
PinData	An integer expression. The least significant bit (b0) of PinData is written. For PinDir: 0 sets the pin to output and 1 sets the pin to input. For PinType: 0 sets the pin to digital I/O and 1 sets the pin to analog input.
VarName	A direct or indirect integer variable to store data read from the pin. Data is 0 or 1.

Description

PinDir sets the direction of an I/O pin using bit 0 of PinData (0=output, 1=input)
PinIn reads the voltage on the input pin and translates it to a logic level (returned data is 0 or 1)
PinOut translates the logic level of bit 0 of PinData to a voltage and writes the voltage to the output pin
PinType changes the pin type between digital I/O and analog input using bit 0 of PinData (0=digital, 1=analog)

Microva BASIC supports up to 128 individual I/O pins on up to 8 different I/O ports. Each I/O port can have up to 16 I/O pins. The I/O ports are labeled PortA thru PortH. All pin names are predefined constants. The pins for PortA are named PA0, PA1,...,PA15. The pins for PortB are named PB0, PB1, etc. The pin numbers are assigned consecutively starting with PortA. All pin numbers are shown in the table on the next page. The Pin commands use the pin numbers, which must be a constant. Use the Constant command to assign names to the pin numbers which are more descriptive such as "RedLed".

The PinIn command can only read the pin input if the pin has been configured as an input using the PinDir command and configured as a digital I/O pin using the PinType command. Likewise, the PinOut command only changes the pin output if the pin was previously configured as a digital I/O pin with direction = output. There is a maximum of 16 pin numbers permitted for each pin command, however, pin numbers in the command can be from any of the I/O ports and can be in any order. Any pin numbers in the command from the same I/O port will all be read or written at the same instant. First all pins from PortA are read or written, then any pins from PortB are read or written, then any pins from PortC, etc. Therefore, there is no input or output "pin skew" (delay) between pins read or written on the same I/O port. That is important when reading or writing parallel data from or to fast external devices.

For example, in the statement: PinOut PA2:Rly1, PA3:Rly2, ~PC5:Off, PD2:1, ~PA12:Alm, PC7:Aux, ~PD15:Horn
Pins PA2, PA3 and PA12 from PortA will be written first and all at once. Pins PC5 and PC7 from PortC will be written second. Lastly, the pins from PortD will be written. Only the least significant bit (bit 0) of the PinData value is written to the output pin, the other bits (Bits 1 to 31) are ignored. So if integer variable "Rly2" held the value 75 then pin PA3 will be set to 1. Also, the tilde can be used before the pin number in any of the pin commands to indicate to the compiler to use inverted logic. In the statement above the tilde before pin PC5 means pin PC5 will be set to 1 since "off" is a constant = 0 and the tilde means to invert the logic.

At reset all pins which are user configurable, i.e. any pins which are not fixed in the hardware, are set to analog inputs if the pin can be configured for analog input or to digital input if the pin cannot be configured as an analog input. Digital input pins that are not used should not be left floating because they can draw excessive current. Therefore, in the Reset subroutine in the application program any of the user configurable pins which are not used should be set to digital outputs. Pins that are used as digital inputs or outputs must have their pin type set to 0 (digital I/O) using the PinType command. The PinDir command can be used to change the pin directions. Alternatively, the WriteInt command can be used to set the pin types or directions as explained below. Before, a pin input can be read the pin direction must be set to 1 (input). Any pins that are used as analog inputs must have their pin type set to 1 (analog input) and their pin direction set to 1 (input). See the AIN function for more information on analog inputs.

How to Read or Write All 16 Port Pins Directly

The advantage of the pin commands described above is that they make it easy to control one or more pins on one or more I/O ports without affecting other pins on the port. However, it is also possible to read or write all 16 pins on one port with a single variable by using the WriteInt command or the RDI function since the port registers are accessed using ordinary memory addresses. All port addresses are predefined constants in the compiler (PortA, PortB, PortC, etc). For example, the statement J = rdi(PortD) would set variable J equal to the 16 bits read from PortD. And the statement WriteInt J, PortD would write the lower 16 bits from J to PortD. However, do not write to ports using this method if the port has pins reserved for use by the kernel firmware such as pins controlled by the SPI hardware module or display control pins. Each port has several memory addresses associated with it which controls the port hardware. For example, it is possible to enable pullup resistors on one or more of the port pins and to configure open drain outputs. These extra port registers are accessed using a fixed address offset from the main port address. The address offset constants are shown in the table on the next page. Note that the pin pullup resistors are very weak and highly variable and therefore the pullup resistors should only be used for contact switch type inputs. Externally driven inputs such as open collector inputs should use an external dedicated pullup resistor. For example, to enable a pullup resistor on pins PC5 and PC14 use the command:

```
WriteInt  b5 + b14, PortC + PortPullUp
```

PinDir PinNum:PinData, PinNum:PinData, PinNum:PinData, PinNum:PinData, . . .
PinIn PinNum:VarName, PinNum:VarName, PinNum:VarName, PinNum:VarName, . . .
PinOut PinNum:PinData, PinNum:PinData, PinNum:PinData, PinNum:PinData, . . .
PinType PinNum:PinData, PinNum:PinData, PinNum:PinData, PinNum:PinData, . . .

Description (continued)

The Port Latch

Each port has a latch. The port latch holds the data bits written to the port which are output to the pins when the pins are set to outputs. For example, J = rdi(PortA) reads the logic states of PortA, both the input pins and the output pins, however, J = rdi(PortA + PortLatch) reads only the PortA latch. The PinIn command reads the state of the pins not the latch. When the port is written the port latch is also written with the same data. So if a port pin is configured as an open drain output and the latch pin value is set to 1, the value read from the port pin might be 0 due to external circuitry holding the pin low but the value read from the latch will still be 1.

Port Pin Numbers

PIN	A	B	C	D	E	F	G	H
00	00	16	32	48	64	80	96	112
01	01	17	33	49	65	81	97	113
02	02	18	34	50	66	82	98	114
03	03	19	35	51	67	83	99	115
04	04	20	36	52	68	84	100	116
05	05	21	37	53	69	85	101	117
06	06	22	38	54	70	86	102	118
07	07	23	39	55	71	87	103	119
08	08	24	40	56	72	88	104	120
09	09	25	41	57	73	89	105	121
10	10	26	42	58	74	90	106	122
11	11	27	43	59	75	91	107	123
12	12	28	44	60	76	92	108	124
13	13	29	45	61	77	93	109	125
14	14	30	46	62	78	94	110	126
15	15	31	47	63	79	95	111	127

Port Control Register Address Offsets

Constant	Offset	Description
PortType	-32	Analog Input Enable (0=off,1=on)
PortDir	-16	Tristate Control (0=output,1=input)
PortLatch	+16	Port Output Latch
PortOpenDrain	+32	Open Drain Control (0=off,1=on)
PortPullUp	+48	Pullup Control (0=off,1=on)

```
;-------------------------------------------------------------------------------------------
;this example shows how the pin state can be read into an individual variable or an array
  PinIn   ~PB5:PgmSwitch, PB7:MyArray[J+5]
;-------------------------------------------------------------------------------------------
;this example shows that pin data can be an expression
  PinOut  PB5:AlarmFlags, PC3:AlarmFlags}1, PD12:AlarmFlags}2
;-------------------------------------------------------------------------------------------
;the following statements all set pin PB12 = 0.
  Constant  RedLed=PB12
  PinOut  ~RedLed:On
  PinOut  PB12:Off
  PinOut  28:0
;-------------------------------------------------------------------------------------------
;read all 16 bits from PortB, write all 16 bits to PortA
  PortB_Data = rdi(PortB)
  WriteInt  0xFE2A, PortA
;-------------------------------------------------------------------------------------------
```

Print String Expression
Print X, Y, Color, String Expression

String Expression	The string to print to the screen.
X, Y (optional)	X and Y are integer expressions which equate to the position of the upper, left corner of the string printed to the display buffer. Default X=10 and default Y=10.
Color (optional)	An integer expression which equates to the color index of the printed text. Default value = 0xFF which is white in the reset color palette SysColors.

Description

The Print command writes a text string to the display. Displays are "double buffered" meaning the print command does not write directly to the display, instead it writes to the RAM display buffer. The Refresh command writes the display buffer to the display. See the description of color graphic displays at the beginning of this manual. For the first form of the Print command, which does not include X, Y, and Color, the display buffer is automatically cleared using the CLS command, then the Print command is executed then the Refresh command is executed. Therefore, the command Print "Hello World" is the same as:

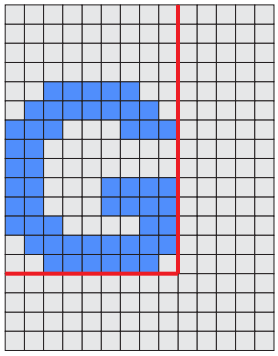
```
CLS  3
Print  10, 10, 0xFF, "Hello World"
Refresh
```

Note that color index 3 is blue and color index 0xFF is white in the reset color palette SysColors. When the X, Y and Color values are included in the Print command the CLS and Refresh commands are not automatically executed and they must be separately executed in the application program. That makes it easier in the application program to print several items to the display buffer and then later write the entire buffer to the display. The Print command writes text to the display using the active font. The firmware kernel system variable FontPntr holds the address of the active font. Any number of fonts can be used in the application program. Each font file includes the data to draw one size font. Characters in the font have variable width and height.

Font Names

Microva BASIC font files can use any name, however, the file extension is always ".mvf". Most fonts are named based on the size of the capital letter "G". So for the Font F0914A the statement Print 0, 0, blue, "G" would look like the image shown below. The font name includes a suffix letter which is A, H or N. The A suffix is used for fonts that include all 192 ANSI character codes shown in Appendix A. The H suffix indicates that the font only includes half of the characters, character codes 0x20 thru 0x7F, known as the ASCII characters. The N suffix indicates that the font only includes the numbers (character codes 0x20 thru 0x3F). A full font that includes all of the 192 ANSI characters can be used to print essentially all Latin based Western Languages such as English, German, Italian, Spanish, French, Danish, Swedish, Portuguese, etc. Several royalty free finished fonts can be downloaded from the Microva website. Its also easy to make additional fonts as explained in Appendix C. There is one pre-loaded font in all firmware versions that support graphic displays called the system font. The table on the next page shows the system font names. All other fonts used in the application program need to be loaded separately as explained below.

Font F0914A



Font Attributes

The font character pixels are written to the display buffer using the text color. The character background pixels are not written to the display buffer so the background is transparent. Most font characters have one or more vertical offset pixels as shown in the example to the right. Fonts include the spacing between characters, called the kerning, and the spacing between words and the spacing between lines. These spacings are the same for all characters in the font, however, the spacings can be changed using the print control codes as explained on the next page. The Print command does not check to see if the text string writes past the display buffer. Therefore, the application software must make sure the text string does not write outside of the display buffer into areas of memory that may cause the program to crash. The Lenx function can be used to find the width of the printed text string in pixels.

The Active Font

Font files must be loaded into ROM or RAM before the font can be used. Usually all the fonts that are used in the application program are loaded into ROM when the application is compiled using the Append command such as: Append C:\Font Files\F0914A.mvf System variable FontPntr holds the active font, i.e. the font used by the Print commands. FontPntr holds the address of the font in memory. Use the WriteInt command to change the active font like: WriteInt addr(F0914A), FontPntr The system font names shown in the table on the next page hold the memory address of the system font in the firmware so the Addr function is not needed to change to these fonts. For example, for firmware 3.4, the command is: WriteInt F0810A, FontPntr

Print String Expression
Print X, Y, Color, String Expression

Description (continued)

The Print Control Codes

The Print command recognizes ANSI character codes 0x21 thru 0x7F and 0xA0 thru 0xFF as characters which are drawn to the display. The Print command also recognizes several "print control codes" which are shown in red in Appendix A. The print control codes are predefined constants (except for "SP" which is the keyboard space bar). The control codes only change the parameters for the string being printed. For example, the new line spacing (NL) control code will change the line spacing whenever the carriage return (CR) is encountered in the text string. But when a new Print command is executed the line spacing will revert to the line spacing defined in the active font. Use the CHR function to execute the print control codes. Most print control codes require a data byte which is the next character in the string. See the examples below. The print control codes are:

Name	Code	Description	Data Byte	Data Byte Range
SP	0x20	word space	<none>	
CR	0x0D	carriage return	<none>	
LF	0x0A	line feed (no effect)	<none>	
NC	0x0E	new text color	color index	0 to 255
NP	0x0F	new text position	offset from current posn	-128 to 127
NK	0x0C	new char spacing	new character space	0 to 255
NW	0x0B	new word spacing	new word space	0 to 255
NL	0x07	new line spacing	new line space	0 to 255

System Font Names

Firmware Version	Font Name
2.5	<none>
3.4	F0810A
4.6	F1018A
5.2	F1421A

```
;-------------------------------------------------------------------------------------------
;CLS and refresh is not needed for this form of the print command

 Print  "MyVar = " + cis(MyVar)
;-------------------------------------------------------------------------------------------
;CLS and refresh commands are needed for this form

 CLS White
 Print  60, 80, Blue, "Hello World"
 Refresh
;-------------------------------------------------------------------------------------------
;print "red green blue" in the proper colors using the NC control codes

 CLS White
 Print 0, 0, red, "red" + chr(NC) + chr(green) + "green" + chr(NC) + chr(blue) + "blue"
 Refresh
;-------------------------------------------------------------------------------------------
;print "World" with proper kerning

 Print "W" + chr(NP) + chr(-3) + "orld"
;-------------------------------------------------------------------------------------------
```

Program Firmware Version, Program Name, Compiler Option

Firmware Version	The firmware version number must be specified. The firmware versions allowed are shown in Table 1 at the beginning of this manual.
Program Name	A string expression which equates to the application program name.
Compiler Option (optional)	There is one compiler option allowed: +n = include the NV flag for Integer/Float/String commands

Description

The Program command tells the compiler which firmware version is being used so that the compiler can show an error if the application program exceeds the available RAM or ROM. The Include and Append commands are the only non-comment statements allowed in the source program file before the Program command. The firmware version number and program name will be displayed on devices with a display when in program mode. Also, on the Microva BASIC IDE "connect" screen, the Read Node ID command will display the firmware version number and the program name. The program name can include the carriage return (so the name can be displayed on multiple lines) and any ANSI characters shown in the appendix. When the target device has a display, the program name must be kept short enough to fit on the display (that is the application programmer's responsibility and is not checked in the firmware). The compiler option +n tells the compiler to include the nonvolatile (NV) flag options when defining variables using the Integer, Float or String commands. The +n option can only be used for devices that support nonvolatile variables using EERAM memory. See the NvVarsLoad command for a description of nonvolatile variables.

```
;-------------------------------------------------------------------------------
 Include  Graphic Display Library.txt
 Include  Motor Control Library.txt
 Append   C:\Images\Other\Lena.bmp
 Program  4.4, "My Application 1.0" + chr(CR) + "copyright 2017 MyCompany", +n
;-------------------------------------------------------------------------------
```

Result = rdb(Address)
Result = rdf(Address)
Result = rdh(Address)
Result = rdi(Address)

Address	An integer expression which equates to an address in the CPU address space. For rdb() Address can be any value. For rdf() Address must be evenly divisible by 4. For rdh() Address must be even. For rdi() Address must be evenly divisible by 4.
Result	Result depends on the command: rdb() returns a byte (8 bits) rdf() returns a float (32 bits) rdh() returns a "half" integer (16 bits) rdi() returns an integer (32 bits)

Description

The RDB (read byte), RDF (read float), RDH (read half integer) and RDI (read integer) functions return data from a memory address anywhere in the CPU address space. Therefore, these functions can be used to read data from RAM or ROM or from the CPU hardware registers such as the I/O port registers. The source data type, i.e. integer, float or string, is not checked. Use the write commands (WriteByte, WriteHalf, WriteInt and WriteFlt) to write data to any address in the CPU address space.

```
;-------------------------------------------------------------------------------
;read data from array using memory address

  fData[0] = 1.125
  fData[1] = 2.125
  fData[2] = 3.125
  fData[3] = 4.125
  fData[4] = 5.125

  P = rdf(addr(fData) + 12)         ;P = 4.125
;-------------------------------------------------------------------------------
;read 16 bit data from PortD

  J = rdh(PortD)
;-------------------------------------------------------------------------------
```

Rectangle X, Y, W, H, Color, Line Width

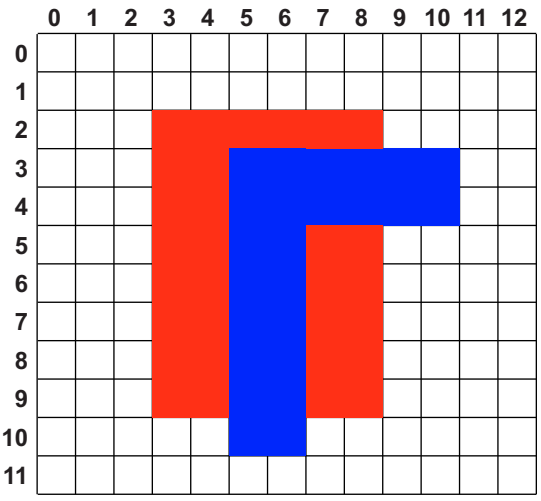
X, Y	X and Y are integer expressions which equate to the position of the upper, left corner of the rectangle drawn to the display buffer.
W	An integer expression which equates to the rectangle width.
H	An integer expression which equates to the rectangle height.
Color	An integer expression which equates to the color index of the rectangle.
Line Width (optional)	An integer expression which equates to the rectangle outline width and also determines which sides of the rectangle outline to draw.

Description

The Rectangle command draws a solid (filled) rectangle or a rectangle outline to the display buffer. If the Line Width parameter is not included or if the line width value is 0 the rectangle will be drawn solid. When the line width parameter is included and is not 0 only the rectangle outline will be drawn. The line width does not change the overall width and height of the rectangle. The application software must make sure the rectangle does not draw past the display buffer RAM. Both the solid rectangle and the rectangle outline are drawn using the color index value. The Line Width parameter determines the rectangle outline width and also determines which of the four sides of the rectangle to draw. The minimum line width is 1. The maximum line width is the lesser of W/2 or H/2 or 255. The lower 8 bits of Line Width (bits 0 to 7) set the width of the outline. Bits 8 thru 11 of Line Width determine which of the four sides of the rectangle outline are drawn:

```
Bit 11 = left side   (0 = draw, 1 = don't draw)
Bit 10 = right side  (0 = draw, 1 = don't draw)
Bit  9 = top side    (0 = draw, 1 = don't draw)
Bit  8 = bottom side (0 = draw, 1 = don't draw)
```

```
;-------------------------------------------------------------------------------
;draw the rectangles below
  Rectangle  3, 2, 6, 8, Red
  Rectangle  5, 3, 6, 8, Blue, b10 + b8 + 2
  Refresh
;-------------------------------------------------------------------------------
```



Refresh SourceX, SourceY

Source X, Y (optional)	Integer expressions which equate to the position in the display buffer of the upper/left corner of the source image block. Default values = 0. Note: For firmware version 5.2 SourceX must be evenly divisible by 4.

Description

Displays are "double buffered" meaning display write commands like Print write to the internal display buffer in RAM, not to the actual display. The Refresh command writes the pixel data from the display buffer to the display. The display buffer can be larger than the actual display resolution which is useful for storing sprites or graphics "off screen". The Refresh command writes a rectangular block of pixels (the source image block) from anywhere within the display buffer to the display. The source image block size exactly matches the display size. For the model MB411 hardware the display resolution is 320 x 240, therefore, the source image block size is 320 x 240 (76,800 bytes). The display buffer can be the same size or larger than the display size, but not smaller. See the description of color graphics display at the beginning of this manual.

Firmware Version 5.2

The SourceX value must be evenly divisible by 4 for firmware version 5.2. Also, firmware version 5.2 uses an advanced CPU with an on-chip **G**raphic **D**isplay **C**ontroller (GDC) hardware module. The GDC outputs color 0 from the color palette during the horizontal and vertical blanking intervals. Therefore, color 0 in ColorPalette must be black (the color palette is described at the beginning of this manual).

```
;-------------------------------------------------------------------------------
;refresh the display starting at position X = 16, Y = 20 in the display buffer

 Refresh  16, 20
;-------------------------------------------------------------------------------
```

ResetCPU

Description

The ResetCPU command resets the CPU (microprocessor) which is the same as turning the power off/on. This command has no parameters. The ResetCPU command can be used to force the unit into program (PGM) mode. For example, on devices with a display and keypad the DOWN arrow key could be used to force the unit into PGM mode in software by executing the ResetCPU command. This would force the unit into PGM mode without having to cycle the power.

Return

Return Integer Expression
Return Float Expression
Return String Expression

Integer Expression	The result of the integer expression will be returned.
Float Expression	The result of the float expression will be returned.
String Expression	The result of the string expression will be returned.

Description

The Return command returns an integer, float or string value to the "calling" subroutine or to the main program. When Return is executed with no parameter, the value returned will be unknown. The Return command exits the main program loop if it is executed in the main program or exits the subroutine in which is was executed. Therefore, it performs the same function as the End Program or End Sub commands, except with the added function of returning a value to the calling subroutine. There is no reason to return a value when Return is executed in the main program loop since any returned data is lost when the next program loop begins.

Subroutines can be nested and can be executed as commands or functions as described on the Sub command page. For example, MySub (with two parameters) can be called like:

```
MySub  Param1, Param2
```

or

```
J = MySub(Param1, Param2)
```

In the second case MySub is being treated as a function and is expected to return a value. The Return command is used in MySub to return that value. The returned value is not type checked against the function result variable type, i.e. it is up to the programmer to make sure the returned value is the same type as the function result variable. For the example above, that means MySub should return an integer value (assuming J is an integer variable).

```
;-------------------------------------------------------------------------------
;use linear interpolation to calibrate an Analog-to-Digital value

  J = Interpolate(ain(2), CalData[K+0], CalData[K+1], CalData[K+2], CalData[K+3])

  Dim X, X0, Y0, X1, Y1 As Integer

  Sub Interpolate(X, X0, Y0, X1, Y1)
     Return  (X - X0) * (Y1 - Y0) / (X1 - X0) + Y0
  End Sub
;-------------------------------------------------------------------------------
```

Result = **Rev**(Integer Expression)

Integer Expression	Integer expression value which will be reversed.
Result	Integer equal to the reverse byte order of the integer expression.

Description

The REV function swaps the byte order of an integer value which is useful for changing data in "big endian" format to "little endian" format or vice versa.

```
;-------------------------------------------------------------------------------
  J = 0x12345678
  K = rev(J)       ;K = 0x78563412
;-------------------------------------------------------------------------------
```

Result = **Rnd**(Seed Value)

Seed Value	Integer expression which is the "seed" for the random number. Any value can be used to start the random number sequence. Use a seed value of 0 to return the next random number in the sequence once the sequence has begun.
Result	Integer equal to a random number generated from the integer expression "seed".

Description

The RND function returns a pseudo random number. It is not possible to generate a truly random number using a synchronous finite state machine (i.e. a computer). Truly random numbers can only be generated from processes in nature. However, it is possible to simulate a random number to such a degree that for all practical purposes the number can be assumed to be truly random. The RND function uses a pseudo random number generator algorithm to return a "random" number in the standard range for a Microva BASIC integer value, i.e. the range of the returned random number is from -2,147,483,648 to +2,147,483,647. The algorithm requires a non-zero seed value to begin the calculation. The seed can be any number in the standard integer range. Once the random number calculation sequence is started, subsequent random numbers in the sequence can be returned by setting the seed argument value to 0. For the same starting seed value, the returned random numbers in the sequence will be identical. Therefore, the seed value should be chosen such that it also has a degree of randomness. Good seed values are Sys(4) (see the SYS function) and also the value of a timer such as timer(0) or timer(1) after an external event occurs. The returned random integer value can be used to generate integer or float random number in any range desired. See the examples below.

```
;-------------------------------------------------------------------------------
;start random sequence, return 5 random numbers in the range 0 to 23

  Random[0] = abs(rnd(sys(4))!24)
  Random[1] = abs(rnd(0)!24)
  Random[2] = abs(rnd(0)!24)
  Random[3] = abs(rnd(0)!24)
  Random[4] = abs(rnd(0)!24)
;-------------------------------------------------------------------------------
;get next random number in range 1 to 10

  J = abs(rnd(0)!10) + 1
;-------------------------------------------------------------------------------
;get next random float value in range 0.0 to 1.0

  P = fabs(cif(rnd(0))) / 2147483648.0
;-------------------------------------------------------------------------------
```

SaveComSettings COM Settings

COM Settings	An integer expression which equates to the new communication settings.

Description

The devices internal communication settings (COM settings) are typically used to open serial port 1 (UART1). See the Open SerialPort command for a description of the serial I/O ports and the baud rate and data form values. The COM settings are stored in the CPU Flash ROM and are preserved even when a new application program is written to the device. The COM settings can be changed by the application program and can also be changed on the Microva BASIC Compiler IDE "connect" screen. The COM settings are stored as a single 32 bit integer value. The COM settings act as a replacement for physical DIP switches. The SaveComSettings command is used to write the new COM settings to Flash ROM from within the application program. The Sys(3) function can be used to read the COM settings in the application program. The COM settings bits are:

Bits	Description	Range	Factory Setting
b31-b16	other data	0-0xFFFF	0 (not used)
b15-b10	baud rate	0-9	250K or 1M (see table below)
b9 - b8	data form	0-3	8/N/1
b7 - b0	node num	1-247	1

In order to write a new application program to the device, the compiler IDE COM settings and the device COM settings must be the same. Also, when devices are connected on a Modbus network, all nodes on the network must use the same baud rate and data form and slave devices must have unique node numbers. Usually serial port 1 is opened in the reset subroutine in the application program using the COM settings, but not always, it depends on the application program. All Microva Modbus RTU peripheral I/O devices open serial port 1 in Modbus slave mode using the COM settings at reset (see the example below). The Microva Modbus RTU I/O devices also allow the installer to change the COM settings in the field by holding the PGM switch on the PCB and counting the number of LED blinks. No computer is needed, for example, to change the device node number in the field. See the device datasheet. The SaveComSettings command writes the new COM settings to the internal Flash ROM and then resets the CPU. Keep in mind, the Flash ROM can "only" be written a maximum of 20,000 times, beyond that number the Flash ROM may be damaged. Bits 31 thru 16 of the COM settings integer value are not used in the kernel firmware and can be used in the application program for any purpose (or ignored). This "other" data can also be written from the Microva BASIC Compiler IDE when the COM settings are written. See the section on "communication settings" at the beginning of this manual for more information about COM settings.

Baud Rates

Value	Baud Rate
0	2400
1	4800
2	9600
3	19200
4	38400
5	57600
6	115200
7	250000
8	500000
9	1000000

Data Formats

Value	Data Bits	Parity	Stop Bits
0	8	None	1
1	8	None	2
2	8	Even	1
3	8	Odd	1

Default Baud Rates At Reset

Firmware Version	Baud Rate
2.5	250K
3.4	1M
4.6	1M
5.2	1M

```
;-------------------------------------------------------------------------------
;open the serial port using the internal communication settings
   NodeNum  = sys(3) & 0xFF
   BaudRate = sys(3) } 10 & 0x0F
   DataForm = sys(3) } 8 & 0x03
   Open SerialPort1  ModbusSlave, DataForm, BaudRate, NodeNum
;-------------------------------------------------------------------------------
;save new COM settings then reset the CPU

 BaudRate = 3    ;BaudRate = 19200
 DataForm = 2    ;DataForm = 8/E/1
 NodeNum = 5
 SaveComSettings   (BaudRate{10) + (DataForm{8) + NodeNum
;-------------------------------------------------------------------------------
```

Select Case Compare Expression

Compare Expression	An integer expression or a float expression or a string name which will be used to compare with the "Case" statements.

Description

The Select Case statement is used to execute one or more lines of code based on the value in Compare Expression. The compare expression can be an integer expression or a float expression or a string name. Compare Expression is compared with the values in the Case statements. The Case statement values must be the same type as the compare expression type, i.e. integer or float or string. When the value in the Case statement matches the value of Compare Expression, the statements for that case will be executed. The Case statements can be in the single line form or the block form. Use a colon after the case values to indicate the single line form. If...Then...Else statements can be used in the block form but not the single line form of the Case statement. Both the single line form and block forms can be used within the same Select Case command. The Select Case command can include an unlimited number of Case statements. The Case Else statement is optional. If Compare Expression does not match any of the Case statements, the Case Else statement or statements will be executed. The Select Case command must be ended with the End Select statement as shown in the examples below. When comparing float values it is usually best to compare a range of values using the To keyword rather than exact values. See the If...Then...Else command for an explanation of why float values can sometimes be inexact. There can be several comparison expressions in each Case statement. The expressions must be separated by commas. The To keyword can be used with integer and float expressions to select a range of comparison values. For example,

```
Case 5 To 7, 11 To 13, 2
```

will be true if the Compare Expression is equal to 5, 6, 7, 11, 12, 13 or 2. When using the To keyword, the compare range values must be in order of lesser value to greater value. For example, if the compare value is -6, the statement Case -10 To -5 will be true, however, the statement Case -5 To -10 will be false because the compare values are in the wrong order. When comparing strings, the Compare Expression must be a string variable name, however, the Case statements can include one or more string variable names or string expressions such as this example which compares for 3 different strings:

```
Case "hello", "The " + AnimalName + "jumped" + chr(CR), MyString
```

```
;-------------------------------------------------------------------------------
;compare string

 MyString = "Horse"

 Select Case  MyString
    Case "Horse": Print "brayed"
    Case "Cat":   Print "meowed"
    Case "Bird":  Print "chirped"
    Case "Cow":   Print "mooed"
    Case Else :   Print "unknown"
 End Select
;-------------------------------------------------------------------------------
;select case statements can be nested and can use one line or block form

  Select Case  abs(J * 5)
     Case 5, 9, 22 To 30: M = M + 1
     Case abs(K - MyData[L + 2]), abs(MyData[Q]), XZ * XB: M = M + 15
     Case 45 To L * 2
        N = 3 * (M + 2)
        Select Case 2 * YZ
           Case  1 To 10: MyData[5] = XA
           Case 11 To 20: MyData[6] = XB
        End Select
     Case  55
         M = M + 35
         N = 4 * (M + 2)
     Case Else
         M = M + 45
         N = 5 * (M + 2)
  End Select
;-------------------------------------------------------------------------------
```

SerialOut1 String Expression
SerialOut1 Num Bytes, Address
SerialOut2 String Expression
SerialOut2 Num Bytes, Address

String Expression	The string of data to transmit using UART1 or UART2.
Num Bytes	An integer expression which is the number of bytes of data to transmit from the source data array. Range is 1 to any.
Address	An integer expression which is the address of an integer array in RAM or ROM which holds the source data to transmit.

Description

The SerialOut command is used to write data to the serial port. Use the SerIn function to read data from the serial port. See the Open SerialPort command for a description of serial I/O. The serial port must be opened before the SerialOut command can be used. Serial data is transmitted as an 8 bit unsigned byte (range is 0 to 255). The SerialOut command can use a string expression or an integer array as the source for the serial output data. When using an integer array source, only the lower 8 bits of each 32 bit integer value in the array is transmitted, i.e. only the least significant byte is transmitted. String data is, by definition, in byte (character) format. The SerialOut command pauses the application program until all bytes in the source string or source array have been transmitted. If the serial port is RS-485 compatible, the transmit enable pin (U1TE or U2TE) must be set high before the data is transmitted and reset after the data is transmitted. In Modbus master or slave modes the RS-485 transmit enable pins are controlled automatically.

The SerialOut command is often used to send data to the Microva BASIC Compiler connect menu terminal window for program debugging. When sending text to the terminal window append the CR (carriage return) character (ANSI char code = 0x0D) to the string in order to move to a new line in the terminal (see examples below and see the SerIn function). See the appendix for a table of ANSI character codes which can be displayed in the terminal window.

```
;-------------------------------------------------------------------------------
;display the value of MyVar on the Microva BASIC "connect" screen using RS-485 I/O

PinOut      U1TE:On
SerialOut1  "MyVar = " + cis(MyVar) + chr(CR)
PinOut      U1TE:Off
;-------------------------------------------------------------------------------
;transmit the hexadecimal value of MyVar using RS-232 I/O

SerialOut1  "0x" + hex(MyVar) + chr(CR)
;-------------------------------------------------------------------------------
;transmit the lower 3 bytes of MyVar (least significant byte first) using RS-232 I/O

SerialOut1  chr(MyVar) + chr(MyVar } 8) + chr(MyVar } 16)
;-------------------------------------------------------------------------------
;This example shows how to transmit DMX-512 data packets. DMX-512 (digital multiplex)
;is an RS-485 master/slave network communication standard typically used for
;controlling lighting and animatronics during stage shows. The first element in the
;source data array is the DMX-512 "start code" (which is typically 0). Each additional
;element in the array holds one channel of data. This example assumes U1TE = already on.

  PinOut     U1TX:0                  ;TX the DMX-512 "break"
  DelayFast  100
  PinOut     U1TX:1                  ;TX the DMX-512 "make-after-break" (MAB)
  DelayFast  12
  Open  SerialPort1  SerialIO, 1, 7  ;DMX-512 data form = 8/N/2, baud = 250000
  SerialOut1  11, addr(DmxData)      ;TX start code + 10 DMX channels
  Close SerialPort1                  ;UART must be turned off to return control of U1TX
;-------------------------------------------------------------------------------
```

Result = SerIn(Integer Expression)

Integer Expression	Read a byte of data from UART1 if integer expression evaluates to 1. Read a byte of data from UART2 if integer expression evaluates to 2. Do nothing if integer expression evaluates to any other value.
Result	Integer which is the result of the function: n = data byte read from UART (range is 0 to 255) -1 = UART receive buffer is empty -2 = UART receive buffer has overflowed -3 = UART has detected a frame error -4 = UART has detected a parity error

Description

The SerIn function is used to read data from the serial port. Use the SerialOut command to write data to the serial port. See the Open SerialPort command for a description of serial I/O. The serial port must be opened before the SerIn function can be used. Serial data is received as an 8 bit unsigned byte (range is 0 to 255). Both UARTs have 8 level deep FIFO (First-In-First-Out) buffers which hold the received data. One byte of data is returned from the buffer each time the SerIn function is called. When the FIFO is empty the SerIn function will return -1. At 19200 baud and 8/N/1 data format the FIFO will fill up in approximately 4ms when data is streaming into the FIFO since each character takes 520us (10/19200). Therefore, the application program must check for data in the FIFO at least every 4ms or the buffer will overflow. The SerIn function returns -2 when the buffer overflows. If the buffer overflows all data in the buffer will be lost (the buffer is cleared). If the UART receives a byte of data but the stop bit is not where it is expected to be the SerIn function will return -3 (frame error). If the serial port is enabled with even or odd parity and the received data byte does not match the parity bit, the function will return -4.

When the serial port hardware uses logic level serial I/O or RS-232 level serial I/O (check the hardware data sheet) the port is considered "full duplex" meaning data can both be transmitted and received at the same time. However, when the port is RS-485 compatible the data is "half duplex" meaning the UART can transmit or receive data but not at the same time. Therefore, when using RS-485 serial I/O, after data is received using the SerIn function, the application program should wait at least one bit time before transmitting data to the port using the SerialOut command. The extra time is needed to allow the RS-485 half duplex hardware time to "turn around" for transmitting. For the example of 19200 baud above, one bit time = 52us. Note, this applies only to RS-485 I/O when using the serial I/O mode. In Modbus master or Modbus slave modes the kernel firmware automatically handles the RS-485 receive and transmit timing.

```
;-------------------------------------------------------------------------------
;read a "carriage return" terminated string

 NewString = ""
 NewChar = serin(1)
 While NewChar <> CR
    If NewChar > -1 Then NewString = NewString + chr(NewChar)
    NewChar = serin(1)
 Wend
;-------------------------------------------------------------------------------
;pause the program for debugging until any character is received then transmit the
;variable data to the terminal window using RS-485 and continue the program

 J = -1
 While J < 0
    J = serin(1)
 Wend
 Delay 1
 PinOut      U1TE:On
 SerialOut1  "MyVar = " + cis(MyVar) + chr(CR)
 PinOut      U1TE:Off
;-------------------------------------------------------------------------------
```

Result = **Sin**(Integer Expression)

Integer Expression	The angle in degrees x 10. Range is 0 to 3599 (i.e. 0 to 359.9 degrees).
Result	Integer equal to the sine of the angle x 10000.

Description

The Sin function returns the sine of the angle x 10,000. The Sin function uses integer math. The result can easily be converted into floating point using the CIF function (see example below). The argument is the angle in degrees x 10.

```
;-------------------------------------------------------------------------------
  Ø = 225                ;angle = 22.5 degrees
  J = sin(Ø)             ;integer result = 3827
  P = cif(J) / 1.0e+4    ;float result = 0.3827
;-------------------------------------------------------------------------------
```

Sleep

Description

The Sleep command puts the CPU into low power "sleep" mode for a period of 32ms. The CPU, crystal oscillator and all hardware modules, interrupts and timers will be turned off. The CPU power drops to less than 1% of the normal operating power. This does not mean the overall PCB power will drop by 99% since there are other hardware devices on the PCB that still draw the normal amount of power. CPU peripheral hardware modules (such as the serial I/O ports) that were on before the Sleep command was executed will turn on automatically when the CPU exits sleep mode. Interrupts are turned off during the Sleep command and must be re-enabled after the Sleep command. For example, serial port 1 would need to be re-opened if it was previously in Modbus slave mode. The output pin states will remain in the state they were in before the command was executed, i.e. if the pin was driving an LED on, the LED will remain on the entire time the CPU is in sleep mode. The Sleep command relies on the CPUs internal WatchDog timer in order to wake up. This timer runs independently of the normal system clocks whose frequencies are derived from the highly accurate crystal oscillator. The WatchDog timer accuracy is only +/-15% over the operating temperature range, i.e. the wake up time is nominally 32ms but can range from about 27ms to 37ms. Firmware version 5.2 does not support the Sleep command.

```
;-------------------------------------------------------------------------------
;check input, remain in Sleep mode until input goes low

 PinIn  5:J
 While J = 1
   Sleep
   PinIn  5:J
 Wend
;-------------------------------------------------------------------------------
```

Result = **SPI**(Integer Expression)

Integer Expression	The data to write to the SPI slave device.
Result	Integer which is the data read from the SPI device.

Description

The SPI function is used to read and write data to a SPI compatible device. The SPI I/O port must be opened using the Open SpiPort command before the SPI function can be used. The slave must be enabled using SS/ in the application program before the SPI function can be used. The SPI integer data is both read and written at the same time. When the data size is 8 bits (which is selected in the Open SpiPort command) the lower 8 bits from the integer expression data will be written to the MOSI pin. At the same time 8 bits will be read from the MISO pin. When the data size is 16 bits, the lower 16 bits will be read/written. When the data size is 32 bits, all 32 bits of the integer will be read/written. The data is read/written in the order of most significant bit first. For example, for 16 bit data bit 15 is the first bit read/written and bit 0 is the last.

```
;-------------------------------------------------------------------------------
;write to MCP4822 D-to-A converter IC (channel 1 or channel 2)

 PinOut  DacEnb_:0
 If DacChannel = 1 Then J = spi(0x3000 + DacData) Else J = spi(0xB000 + DacData)
 PinOut  DacEnb_:1
;-------------------------------------------------------------------------------
;read 20 bytes from FRAM

 PinOut  FramEnb_:0
 J = spi(0xF3)
 J = spi(FramAddr)
 For J = 0 to 19
    FramData[J] = spi(0xFF)
 Next J
 PinOut  FramEnb_:1
;-------------------------------------------------------------------------------
```

Result = **Sqr**(Integer Expression)

Integer Expression	Integer expression which will be converted to its square root. Must be greater than 0.
Result	Integer equal to the square root of integer expression.

Description

The SQR function returns the square root of an integer expression.

```
;-------------------------------------------------------------------------------
  J = 98763
  K = sqr(J)       ;K = 314
;-------------------------------------------------------------------------------
```

String Name, NV Flag, Modbus Address

Name	The name of the string variable. The string name must include the size of the string. The size range is 8 to 256 and must be evenly divisible by 4.
NV Flag (optional)	Nonvolatile (NV) variables retain their value when the power is off. Check the device data sheet to see if NV vars are supported. The Program command must include the +n option in order to use the NV flag option. 0 = variable is volatile 1 = variable is nonvolatile
Modbus Address (optional)	An integer constant which is the Modbus slave address of the variable. The address is only used when serial port 1 (UART1) is set to Modbus slave mode. The Modbus addresses are incremented for each element in the array. Modbus slave addresses must be unique. Range is 1 to 4095.

Description

The String and Dim commands are used to define string variables. A Microva BASIC string is a byte array of ANSI characters. The first byte in the array defines the length of the string. The maximum size of the string variable is fixed when the variable is created. The array size should be set equal to the maximum number of characters allowed in the string + 1. The size of the string variable must be evenly divisible by 4. The minimum size is 8 and the maximum size is 256, therefore, the maximum string length is 255 characters. The actual string stored in the array can be smaller than the array size but not larger. When a string longer than the array size is written to the array it is truncated to fit. This example shows how the string data is stored in memory:

```
                                  RAM Address      Data
                                 ---------------   ----
String  MyString(8)              addr(MyString)+0    5
MyString = "Hello"               addr(MyString)+1   "H"
                                 addr(MyString)+2   "e"
                                 addr(MyString)+3   "l"
                                 addr(MyString)+4   "l"
                                 addr(MyString)+5   "o"
                                 addr(MyString)+6   unknown
                                 addr(MyString)+7   unknown
```

See the appendix for a complete table of ANSI character codes. In order to enter ANSI characters into the string which are not on the computer keyboard, hold the <ALT> key and type "0" + the decimal value of the character using the numeric keypad (not the regular number keys). At reset or power on the kernel firmware sets all string variables to the "null string" (a string with length = 0). To set a string variable to the null string in the application program use empty quote marks like this: `MyString = ""`

Nonvolatile (NV) Variables

Nonvolatile variables retain their value when the power is shut off. The NV Flag option can be used for devices that include the EERAM memory for nonvolatile variables. There is no difference between volatile and nonvolatile variables during normal program execution, i.e. there is no speed penalty for using nonvolatile variables. If the Program command includes the +n option then the NV flags must be included for each String variable defined. If the Program command does not include the +n option then the NV flags should not be included. The full string array is loaded or saved to EERAM when the NV flag option is set to 1. The number of EERAM addresses = string size / 4. So for the example above, if MyString was designated as nonvolatile, then it would use up 2 EERAM memory addresses. See the NvVarsLoad command page for more information on nonvolatile variables.

Modbus Addresses

The String variable Modbus address is optional regardless of whether the NV option described above is used. The address is only used in Modbus slave mode. The number of Modbus addresses used by the string variable = string array size / 2. All Modbus addresses must be unique. See the Integer command page for a more detailed description of Modbus addresses.

```
;-------------------------------------------------------------------------------
  String  MyString(12), 5            ;MyString = 11 chars max, uses Modbus addrs 5 to 10

  MyString = "MyVar=" + cis(MyVar)   ;assign a new string to MyString
;-------------------------------------------------------------------------------
```

Sub Name(VarName1, VarName2, VarName3, ..., VarNameN)

Name	The name of the subroutine.
VarNames (optional)	Subroutine parameters must be simple string, integer or float variable names or array element names with a constant or variable index such as MyVar[3] or MyVar[J]. Parameters cannot use expressions (ex. MyVar[J * 5] is not allowed).

Description

The Sub command is used to define a new subroutine. Subroutines must be defined after the End Program statement. Subroutines can be called from within the main application program or from within another subroutine. A subroutine cannot call itself (subroutines are not recursive). Subroutines can be called using the standard Microva BASIC command format or the subroutine can be called like a function. For example, these two statements are both valid ways to call subroutine "MySub".

```
MySub  K*2, abs(Z/10)          J = MySub(K*2, abs(Z/10))
```

In the second case variable J will be set to the return value from the subroutine. When the subroutine is defined parameters must be enclosed in parenthesis. Use empty parenthesis if there are no parameters. When the subroutine is called like a function, as in the second example above, the subroutine can return integer, float or string data. See the Return command. The returned value can be assigned to a direct or indirect variable. When the subroutine is called using the function format and there is no Return statement in the subroutine, the returned value will be unknown. The parenthesis around the parameters must be included when calling the subroutine like a function. Use empty parenthesis if there is no parameters. All subroutines must end with the End Sub statement. When calling the subroutine using the command form the parenthesis around parameters should not be included. When defining a subroutine using the Sub command, all parameters must be simple direct or indirect variable names. Expressions cannot be used as parameters in the subroutine definition. However, when calling a subroutine the subroutines' parameters can be expressions. Also, subroutines cannot be used as arguments for other subroutines, functions or commands or used within an expression.

All variables in Microva BASIC are "public", meaning they can be read and written from within any subroutine. Therefore, when a parameter is passed to a subroutine, that parameter's variable is written with the new data. For example:

```
Dim MyData, MyArray(20) As Integer

Sub   MySub( MyData,  MyArray[J])
    <subroutine goes here>
End Sub
```

The statements:

```
For J = 0 To 10
    MySub   J * 5,  SumAvg[M + 2] / 100
Next  J
```

are equivalent to:

```
For J = 0 To 10
   MyData = J * 5
   MyArray[J] = SumAvg[M + 2] / 100
   MySub   MyData,  MyArray[J]
Next  J
```

As mentioned above, expressions can be used as parameters when calling a subroutine, however, array names cannot be passed as parameters. Instead of using the array name as a parameter use the array address. The example on the next page shows a subroutine which returns the average of the values in an array using the array address as a parameter.

Sub Reset()

There is one special subroutine, the Reset subroutine. The Reset subroutine is optional. To include it just name the subroutine Reset (with no parameters). The kernel firmware executes the Reset subroutine immediately after the device is reset (when the power is turned on) before the main program software has begun. The Reset subroutine is used to initialize variables and configure the CPU hardware such as the pin directions and the serial ports.

Sub Name(VarName1, VarName2, VarName3, ..., VarNameN)

Description (continued)

```
;-------------------------------------------------------------------------------
;use linear interpolation to calibrate an Analog-to-Digital value

  J = Interpolate(ain(PD2), CalData[0], CalData[1], CalData[2], CalData[3])

  Dim X, X0, Y0, X1, Y1 As Integer

  Sub Interpolate(X, X0, Y0, X1, Y1)
     Return  (X - X0) * (Y1 - Y0) / (X1 - X0) + Y0
  End Sub
;-------------------------------------------------------------------------------
 Sub Reset()
  ;init I/O pins data, type (0=digital, 1=analog), direction (0=out, 1=in)
    PinOut   48:1, 49:1, 50:1, 51:1, 53:1, 54:1, 55:1, 56:1
    PinType  48:0, 49:0, 50:0, 51:0, 53:0, 54:0, 55:0, 56:0
    PinDir   48:0, 49:1, 50:1, 51:1, 53:0, 54:0, 55:0, 56:0
 End Sub
;-------------------------------------------------------------------------------
;find the average of "MyArray" (assumes variables were previously defined)

  M = Avg(addr(MyArray), ubound(MyArray))

  Sub  Avg(ArrayAddr, ArrayBound)
     K = 0
     For J = 0 To ArrayBound
        K = K + rdi(ArrayAddr + J * 4)
     Next J
     Return K / (ArrayBound + 1)
  End Sub
;-------------------------------------------------------------------------------
```

Result = **Sys**(Integer Expression)

Integer Expression	Integer expression determines which system data is returned.
Result	Various integer data which depends on the value of "Integer Expression".

Description

The SYS (**Sys**tem) function returns various system (i.e. kernel firmware) related integer data depending on the value of the argument. The returned values are described below. The integer argument must evaluate to one of the values below or the returned value will be unknown.

Function(Argument)	Function Return Value Description
Sys(0)	Returns the number of pages used in the application program. Even if only 1 byte on the page is used by the application program, that page is considered used. For example, if the page size is 4096 bytes and the program size is 8200 bytes then Sys(0) will return 3 (pages 0, 1 and 2 are used in the application). The page size depends on the firmware version (see FlashErase command).
Sys(1)	Returns the number of unused pages in the application program. All bytes on the page must be erased to be considered unused. The page size depends on the firmware version (see the FlashErase command).
Sys(2)	Returns the memory address of the start of the application program in Flash ROM.
Sys(3)	Returns the communication settings. See the SaveComSettings command for a description of the communication settings.
Sys(4)	Returns a unique, random number "fingerprint" generated by the Microva BASIC Compiler. Each time the application program is compiled a new random number is saved in the program ROM by the compiler. Even the exact same program re-compiled will generate a new random number. Once the program is compiled this number never changes. This number can be read using Sys(4). Note, this number is unrelated to the value returned from the random function RND.
Sys(5)	Returns the ROM address of the compiler version string, ex. "2.8".
Sys(6)	Returns the ROM address of the application program name string, ex. "My App Program version 1.0".
Sys(20)	Returns the ROM address of the firmware version string, ex. "4.6.1".
Sys(21)	Returns the ROM address of the Microva kernel firmware copyright string.

```
;-------------------------------------------------------------------------------
;print the application program name

   Print  linein#(sys(6))
;-------------------------------------------------------------------------------
;read the value at the last address in Flash ROM (assumes the page size = 4096 bytes)

  J = rdi(sys(2) + 4096 * (sys(0) + sys(1)) - 4)
;-------------------------------------------------------------------------------
```

Table Name
Table# Name

Name	The name of the ROM table.

Description

A Microva BASIC table is just an array that exists in ROM instead of RAM like a normal variable array. Tables are a convenient way to store non-changing data. Tables can store Integer, Float or String data, however, each table in the application program can only store one type of data (i.e. all integer data or all float data or all string data). Tables can be any size and there is no limit to the number of tables allowed in the application program. The Table command defines the table name. The first non-comment line following the Table keyword must be the start of the table data. Variable names cannot be included in the table, however, constant names can be included. All data entries in the table must be of the same type. The first data entry defines the table type. For example, if the first data value was -1234 then the entire table must consist of only Integer values. If the first value was -1234.0 then the table must include only Float data. If the first entry was "-1234" then the table is all String data.

Table data is accessed in the same way as variable array data, i.e. by referring to individual elements in the table using square brackets and with an expression which equates to the array index (ex. K = MyTable[2 * J + 1]). The first element in the table is index 0. The Ubound function can be used to determine the maximum index number. The End Table command indicates the end of the table. The Table command can only be used after the Program statement but before the first program line in the program or between the subroutine definitions. All table data is "public" meaning it can be read from within any subroutine.

String Tables

A String table is really an array of strings. Each entry in the table (each index) is one string. The strings do not need to be the same length. The strings can be separated by a comma or by placing each string on a separate line in the table. String table entries must be string constants. Only the CHR string function with a constant argument can be used in the string constant. See the table of chemicals on the next page for an example. Also, see the example which uses the Ubound function to print all the strings from a string table when there is an unknown number of strings in the table. The compiler stores all ROM strings in the program at one central location in the ROM (one giant array of strings). Therefore, each string table is actually just an array of memory addresses (pointers) where each entry in the table is an integer equal to the address of the string in ROM.

State Tables

Adding the number symbol suffix (#) to the Table# command creates a special type of Integer table called a state transition table (which is also just called a state table). A state table is just an ordinary Integer table except the first value in each row in the table is referred to as the "present state" and each column in the table represents an input condition that changes the state from the present state to the "next state". Normally, any names used in a table must be previously defined constants. However, for a state table the actual integer value of each state is not important. It is only important that each state has a unique value. The Table# command automatically assigns a unique value to each "present state" name in the table starting with 1 and then incrementing for each additional new state name. This saves the extra work of separately assigning a value to each state name in the table using the Constant command. Since state tables often have hundreds of states, the Table# command is a convenience. Only the state names in the first column in the table will be assigned constant values and there is no limit to the number of rows or columns in the state table.

State tables are one of the fundamental ways that automation control systems are implemented. Some example finite state control systems are: elevator controllers, vending machines, traffic lights, combination locks, etc. For Microva BASIC applications state tables are commonly used to hold the cursor positions for navigating around menus. For example, when the operator presses the DOWN key the next menu to display (the next state) depends on the menu currently displayed (the present state).

As a simple example, consider an elevator controller in a building with just four floors. Assume the only inputs to the controller are an UP button and a DOWN button. The state table is shown on the next page for the elevator's present state and the elevator's next state which depends on the input. Also shown is the state table data entered using the Table# command. The constant values assigned by the Table# command are: Floor_Gnd = 1, Floor_1 = 2, Floor_2 = 3, Floor_3 = 4. The application software must be written such that an entry of 0 in the table means to increment the state if the input is UP or decrement the state if the input is DOWN. Therefore, only the exceptions in the state table need to be defined which makes it easy to read. In this example the only exceptions are the elevator can't move up if its already on Floor_3 and it can't move down if its already on the ground floor.

Table Name
Table# Name

Table Examples

```
;-------------------------------------------------------------------------------
;state table example for elevator with four floors
```

States Of Elevator With Four Floors

Present State	Input UP	Input DOWN
Floor_GND	Floor_1	Floor_GND
Floor_1	Floor_2	Floor_GND
Floor_2	Floor_3	Floor_1
Floor_3	Floor_3	Floor_2

```
 Table#  ElevatorStates
;  present state  Input UP  Input DOWN
;  -------------  --------  ----------
     Floor_GND,   0,        Floor_GND
     Floor_1,     0,        0
     Floor_2,     0,        0
     Floor_3,     Floor_3,  0
 End Table
;-------------------------------------------------------------------------------
;table of chemical names (each prints in different color)

 Table Chemicals
    chr(NC) + chr(yellow) + "SO" + chr(183)                  ;sulpher dioxide
    chr(NC) + chr(blue)   + "H"  + chr(183) + "O"            ;water
    chr(NC) + chr(green)  + "NH" + chr(184)                  ;ammonia
    chr(NC) + chr(orange) + "C"  + chr(187) + "H" + chr(187) ;benzene
    chr(NC) + chr(purple) + "CH" + chr(184) + "OH"           ;methanol
 End Table
;-------------------------------------------------------------------------------
;print all strings from a table with an unknown number of strings

  For J = 0 To Ubound(Chemicals)
     Print Chemicals[J]
     Delay 2000
  Next J
;-------------------------------------------------------------------------------
;print day of week

 Table DayOfWeek
   "Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"
 End Table

 Day = 3
 Print DayOfWeek[Day]          ;prints "Wednesday"
;-------------------------------------------------------------------------------
;search table "TempData", print the index number of temperature = 75

 Table  TempData
    122, 76, 93, 107, 88
    189, 195, 23, 37, 75, 57, 34
    28, 119, 67, 66, 52, 45
 End Table

 For J = 0 To Ubound(TempData)
    If TempData[J] = 75 Then Exit
 Next J
 If J > Ubound(TempData) Then Print "not found" Else Print "index=" + cis(J)
;-------------------------------------------------------------------------------
```

Result = Tan(Integer Expression)

Integer Expression	The angle in degrees x 10. Range is 0 to 3599 (i.e. 0 to 359.9 degrees).
Result	Integer equal to the tangent of the angle x 10000.

Description

The Tan function returns the tangent of the angle x 10,000. The tangent function uses integer math. The result can easily be converted into floating point using the CIF function (see example below). The argument is the angle in degrees x 10. For angles approaching 90 and 270 degrees the result will quickly loose accuracy.

```
;-------------------------------------------------------------------------------
  Ø = 3168                ;angle = -43.2 degrees (316.8 degrees)
  J = tan(Ø)              ;integer result = -9391
  P = cif(J) / 1.0e+4     ;float result = -0.9391
;-------------------------------------------------------------------------------
```

Result = **Timer**(Integer Expression)

Integer Expression	The timer channel number (0 or 1).
Result	Integer equal to the timer count. Timer 0 range is 0 to 0xFFFF FFFF. Timer 1 range is 0 to (Timer1Max - 1). Timer1Max is a predefined constant: **`Timer1Max = 0x7FFF FD78`**

Description

The Timer function returns the count from the CPU instruction clock timer or from a virtual millisecond timer. The Timer function does not use any of the CPU hardware timer modules. Therefore, all of the CPU hardware timer modules can be used in the application program independently of the Timer function.

Timer(0)

Timer(0) is a 32 bit hardware timer which runs continuously with a fixed clock frequency = SysClk / 2, i.e. it increments every other CPU clock. The SysClk frequency depends on the firmware version as shown in Table 1 at the beginning of this manual. Timer(0) is highly accurate since the CPU clock (SysClk) is generated from a crystal oscillator. Timer(0) runs independently of the application software and is only ever stopped in Sleep mode (see the Sleep command). Timer(0) counts from 0 to 0xFFFF FFFF and then "rolls over" (back to 0) and continues counting. Timer(0) is reset to count = 0 when the device is reset. For example, with SysClk = 100MHz, Timer(0) will roll over approximately every 86 seconds. Timer(0) can be used to measure time intervals. For example, the following statements measure the time (in SysClks/2) to execute a "program loop":

```
OldTime = NewTime
NewTime = timer(0)
LoopTime = NewTime - OldTime
```

The statements above work properly even when Timer(0) rolls over as long as the program loop time is less than half of the Timer(0) max count, i.e. the program loop time must be under 43 seconds with SysClk = 100MHz.

Timer(1)

Timer(1) is a 32 bit "soft" timer with a fixed clock frequency = 1kHz , i.e. it increments every millisecond. Timer(1) counts continuously from 0 to (Timer1Max - 1) and then "rolls over" (back to 0) and continues counting. Therefore, if the power is never turned off, Timer(1) will roll over every 24.86 days. Timer(1) is a "soft" timer meaning it is derived from Timer(0) in the kernel firmware and therefore is also highly accurate. If Timer(1) is used in the application program it must be refreshed at least once every 0x7FFF FFFF Timer(0) clocks or it will lose time, i.e. at least once every 43 seconds with SysClk = 100MHz. Timer(1) is refreshed each time the function is called. If Timer(1) is not used in the application program then it does not need to be refreshed. Timer(1) is reset to count = 0 when the device is reset. Timer(1) can be used in the application to derive any timers needed. For example, the statements:

```
Timer1ms    = timer(1)
Timer50ms   = Timer1ms \ 50
Timer125ms = Timer1ms \ 125
```

creates 3 timers in the program = 1ms, 50ms and 125ms. Timers make it easy to program events that need to occur periodically. For example, an easy way to "debounce" a switch input is to read it periodically like this:

```
If Timer50ms Changes Then PinIn  ~5:Switch
```

To make the red LED blink rapidly (4 times per second) the statement would be:

```
PinOut  RLed:Timer125ms
```

That statement works because the PinOut command uses the least significant bit (bit 0) to write the pin and bit 0 of Timer125ms toggles every 125 ms.

Timer(1) rolls over every "Timer1Max" clocks. Timer1Max is a magic number which is evenly divisible by many other numbers such as 4, 5, 8, 10, 20, 25, 40, 50, 65, 100, 125, 200, 250, 260, 500, 1000, etc. Any timers that are derived from Timer(1) with values that are evenly divisible into Timer1Max (such as Timer50ms and Timer125ms above) will run correctly even when Timer(1) rolls

Result = **Timer**(Integer Expression)

Description (continued)

over. However, a timer such as Timer37ms = Timer1ms \ 37 will have a "partial pulse" once every 25 days when Timer(1) rolls over. The application program can correct for this error if necessary, however, in most cases this tiny error which occurs about once a month, and only if the power is never turned off, will not cause a problem. Timer(1) is derived from Timer(0) and is therefore highly accurate, however, when executing a command such as:

```
PinOut  RLed:Timer125ms
```

the output pulse will have a tolerance which is equal to the program loop time. That's because the statement above only executes once every program loop. Therefore, if the program loop takes 200us to execute, then the RLed output pulse can occur at 125ms +/- 200us (a tolerance of 0.16%). This loop time tolerance does not effect the accuracy of the timer. The timer is still as accurate as the CPU crystal oscillator (which is accurate to around 25ppm over the full operating temperature range). That means that averaged over time, the RLed will change every 125ms exactly, however, an individual pulse that changes the RLed might vary from the 125ms ideal by as much as the program loop time (200us in this example). When driving outputs such as LEDs or stepper motors a tolerance of a few percent on the output is acceptable. However, when the output pulse is short or when the program loop time is long, the tolerance may be unacceptable. In that case it is better to use a hardware generated output like from the CPU's PWM output hardware module. That is why it is generally best to avoid using the Delay command in the application program because the Delay command halts the CPU until the delay is completed which means the program loop is halted.

For example, suppose there is the need to turn on an LED for 5 seconds each time a switch is activated. One solution is:

```
If Switch ChangesTo On Then
    PinOut  Led:On
    Delay 5000
    PinOut  Led:Off
End If
```

If that's all the program does then the program is done. However, in most cases an application program is doing many things. The problem with the example above is that the program is blocked while waiting for the 5 second delay timeout. A better way is to keep track of the LED switch activation time and turn off the LED after 5 seconds has elapsed such as:

```
If Switch ChangesTo On Then
    PinOut  Led:On
    LedStartTime = timer(1)
End If
LedTimer = timer(1) - LedStartTime
If LedTimer < 0 Then LedTimer = LedTimer + Timer1Max
If LedTimer >= 5000 Then PinOut  Led:Off
```

The statements above execute in just a few microseconds so the program is not blocked and yet the LED is still turned on for 5 seconds each time the switch is activated. These statements work correctly even if the switch is activated just before Timer(1) rolls over (25 days after reset) because the statement If LedTimer < 0 Then LedTimer = LedTimer + Timer1Max corrects for the rollover.

```
;-------------------------------------------------------------------------------
;measure time in SysClks/2 to execute subroutine DrawScreen
   StartTime = timer(0)
   DrawScreen
   DrawTime = timer(0) - StartTime
;-------------------------------------------------------------------------------
;print time in milliseconds that switch is held on (this runs in the main program loop)
   PinIn ~12:Switch
   If Switch ChangesTo On  Then StartTime = timer(1)
   If Switch ChangesTo Off Then EndTime   = timer(1)
   TotalTime = EndTime - StartTime
   If TotalTime < 0 Then TotalTime = TotalTime + Timer1Max
   Print cfs(cif(TotalTime) * 0.001)
;-------------------------------------------------------------------------------
```

Triangle X0, Y0, X1, Y1, X2, Y2, Color, Line Width

X0, Y0	X0 and Y0 are integer expressions which equate to the position in the display buffer of one of the triangle vertices.
X1, Y1	X1 and Y1 are integer expressions which equate to the position in the display buffer of one of the triangle vertices.
X2, Y2	X2 and Y2 are integer expressions which equate to the position in the display buffer of one of the triangle vertices.
Color	An integer expression which equates to the color index of the triangle.
Line Width (optional)	An integer expression which equates to the triangle outline width. Range is 0 to 33.

Description

The Triangle command draws a solid (filled) triangle or a triangle outline to the display buffer. If the Line Width parameter is not included or if the line width value is 0 the triangle will be drawn solid. When the line width parameter is included and is not 0 only the triangle outline will be drawn. The application software must make sure the triangle does not draw past the display buffer RAM. Both the solid triangle and the triangle outline are drawn using the color index value. The three triangle vertices can be defined in any order but they must be unique, i.e. no two vertices can be the same point. However, the three vertices can all lie on the same line in which case only that line will be drawn.

```
;-------------------------------------------------------------------------
;draw a triangle outline to the display buffer using color index = blue

 Triangle  250, 180, 125, 40, 30, 150, Blue, 5
 Refresh
;-------------------------------------------------------------------------
```

String Result = Trim(String Name)

String Name	The name of a string variable (not a string expression).
String Result	A string which is a copy of the string in "String Name" except with leading and trailing "white space" removed.

Description

The Trim function trims (i.e. removes) all leading and trailing "white space" from the source string. White space is defined as any ANSI character with a character code less than 0x21. The Trim function does not remove white space from the middle of the source string. See the ANSI character code table in the appendix.

```
;-------------------------------------------------------------------------------
;remove leading/trailing white space from MyString

  MyString = "   Hello    World!    " + chr(CR) + chr(LF)
  MyString = trim(MyString)                          ;MyString = "Hello    World!"
;-------------------------------------------------------------------------------
```

Result = **UBound**(Name)

Name	The name of an integer, float or string array or table.
Result	Integer equal to the upper bound of the array.

Description

The UBound (**U**pper **Bound**) function returns the upper bound of an integer, float or string array or table. UBound is useful when passing an array argument to a subroutine as shown in the example below.

```
;-------------------------------------------------------------------------------
;print all strings from "StringTable"

  For J = 0 To ubound(StringTable)
     Print  StringTable[J]
     Delay  2000
  Next J

  Table  StringTable
     "The horse jumped", "The cow mooed", "The dog barked",
     "The cricket chirped", "The frog croaked", "The bird sang"
  End Table
;-------------------------------------------------------------------------------
;multiply even index values in array by 2, odd index values by 3

  DigFilter  addr(MyArray), ubound(MyArray)

  Dim  ArrayIndex, ArrayAddr, ArrayBound As Integer

  Sub  DigFilter(ArrayAddr, ArrayBound)
     For ArrayIndex = 0 To ArrayBound
        If ArrayIndex & b0 = 0 Then
           WriteInt  rdi(ArrayAddr) * 2, ArrayAddr
        Else
           WriteInt  rdi(ArrayAddr) * 3, ArrayAddr
        End If
        ArrayAddr = ArrayAddr + 4
     Next ArrayIndex
  End Sub
;-------------------------------------------------------------------------------
```

String Result = **UCase**(String Name)

String Name	The name of a string variable (not a string expression).
String Result	A string which is a copy of the string in "String Name" except with all upper case letters.

Description

The UCase (**U**pper **Case**) function converts all lower case letters in the source string to upper case. See the ANSI character code table in the appendix for upper and lower case letters. Use the LCase function to convert upper case letters to lower case.

```
;-------------------------------------------------------------------------------
;convert MyString to all upper case

  MyString = "Hello World!"
  MyString = ucase(MyString)     ;MyString = "HELLO WORLD!"
;-------------------------------------------------------------------------------
```

While Comparison
<one or more statements>
Wend

Comparison	A comparison between two integer expressions or two float expressions or a string name and a string expression. The statements will execute repeatedly as long as the comparison is true.

Description

The While...Wend statement executes one or more program statements repeatedly while the comparison condition is true. When the comparison condition is false program execution will continue with the first program statement after the Wend statement. The comparison must be made between integer expressions or between float expressions or between a string name and a string expression. Several comparisons can be combined using the And and OR keywords and the "change bit" compares are allowed. See the If...Then...Else command page for a detailed description of the comparison operators.

```
;-----------------------------------------------------------------------------
;display LED bargraph using Level (range is 0 to 10)

  LedBits = 0
  While Level > 0
     LedBits = (LedBits { 1) + 1
     Level = Level - 1
  Wend
  WriteHalf  LedBits, PortD
;-----------------------------------------------------------------------------
;Boolean And/Or comparison logic is evaluated from left-to-right

  While 5 * J > K + MyArray[M * 2] Or MyVar Changes And MyString = "Hello" + chr(CR)
    Delay  100
    PinOut 7:N - 1
    MyVar = MyVar
    J = J - 1
  Wend
;-----------------------------------------------------------------------------
;pause the program until switch on

  While Switch = Off
    Delay 50
    PinIn ~5:Switch
  Wend
;-----------------------------------------------------------------------------
;freeze program

  While 1 = 1
    J = 0
  Wend
;-----------------------------------------------------------------------------
```

WriteByte Data, Address
WriteFlt Data, Address
WriteHalf Data, Address
WriteInt Data, Address

Data	An integer or float expression which equates to the data to write. WriteByte = write a byte from integer data (bits 0 to 7) WriteFlt = write all 32 bits of float data WriteHalf = write a "half" integer from data (bits 0 to 15) WriteInt = write all 32 bits of integer data
Address	An integer expression which equates to an address in the CPU address space. For WriteByte Address can be any value. For WriteFlt Address must be evenly divisible by 4. For WriteHalf Address must be even. For WriteInt Address must be evenly divisible by 4.

Description

The WriteByte (write byte), WriteFlt (write float), WriteHalf (write half integer) and WriteInt (write integer) commands write data to any memory address anywhere in the CPU address space. Therefore, these commands can be used to write data to RAM or to the CPU hardware registers such as the I/O port registers. The destination memory type (integer, float or string) is not checked. These commands cannot write to ROM memory addresses. Use the FlashWrite command to write ROM memory. Use functions RDB, RDF, RDH and RDI to read data from any address in the CPU address space.

```
;-------------------------------------------------------------------------------
;add 10 to MyData[3} using memory address

  MyData[0] = 125
  MyData[1] = 225
  MyData[2] = 325
  MyData[3] = 425
  MyData[4] = 525

  J = addr(MyData) + 12

  WriteInt  rdi(J)+10, J
;-------------------------------------------------------------------------------
;write to PortD

  WriteInt  0x00FE, PortD
;-------------------------------------------------------------------------------
```

Appendix A ANSI (ISO-8859-1) Character Codes

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	000	001	002	003	004	005	006	NL 007	008	009	LF 010	NW 011	NK 012	CR 013	NC 014	NP 015
1	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
2	SP 032	! 033	" 034	# 035	$ 036	% 037	& 038	' 039	(040	) 041	* 042	+ 043	, 044	- 045	. 046	/ 047
3	0 048	1 049	2 050	3 051	4 052	5 053	6 054	7 055	8 056	9 057	: 058	; 059	< 060	= 061	> 062	? 063
4	@ 064	A 065	B 066	C 067	D 068	E 069	F 070	G 071	H 072	I 073	J 074	K 075	L 076	M 077	N 078	O 079
5	P 080	Q 081	R 082	S 083	T 084	U 085	V 086	W 087	X 088	Y 089	Z 090	[091	\ 092	] 093	^ 094	_ 095
6	` 096	a 097	b 098	c 099	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	€ 160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	166	167	168	© 169	170	171	Š 172	Œ 173	Ž 174	Ÿ 175
B	° 176	± 177	sup 2 178	sup 3 179	Ω 180	µ 181	π 182	sub 2 183	sub 3 184	sub 4 185	sub 5 186	sub 6 187	š 188	œ 189	ž 190	¿ 191
C	À 192	Á 193	Â 194	Ã 195	Ä 196	Å 197	Æ 198	Ç 199	È 200	É 201	Ê 202	Ë 203	Ì 204	Í 205	Î 206	Ï 207
D	Ð 208	Ñ 209	Ò 210	Ó 211	Ô 212	Õ 213	Ö 214	215	Ø 216	Ù 217	Ú 218	Û 219	Ü 220	Ý 221	Þ 222	ß 223
E	à 224	á 225	â 226	ã 227	ä 228	å 229	æ 230	ç 231	è 232	é 233	ê 234	ë 235	ì 236	í 237	î 238	ï 239
F	ð 240	ñ 241	ò 242	ó 243	ô 244	õ 245	ö 246	247	ø 248	ù 249	ú 250	û 251	ü 252	ý 253	þ 254	ÿ 255

Red characters are the print control codes
Blue characters are non-standard ANSI characters

Appendix B Firmware 5.2 Graphic Display Controller Module

Introduction

Microva BASIC supports color graphic display using commands like Print and Image. The color graphic display and the color palette are explained at the beginning of this manual. Firmware version 5.2 uses an advanced CPU with a **G**raphic **D**isplay **C**ontroller (GDC) hardware module on chip. Firmware 5.2 supports display on a VGA type video monitor. The GDC generates the display timing synchronization signals and writes the RGB color data values from RAM memory to the display independently of the CPU operation. The GDC resolution can be set to many different values up to a maximum resolution of 800 x 600. At reset the VGA display resolution is set to 840 x 480, however, only 736 x 430 of this area is "live", i.e. able to be written by the application program. Therefore, there is a 52 x 25 pixel "dead" border around the live area on screen. The border color is set to black at reset, however, the border color can be changed in the application program. The display is double buffered. The memory map at reset is shown below. The GDC output timing and resolution can be changed at anytime in the application program as explained below.

Display RAM vs Display Buffer

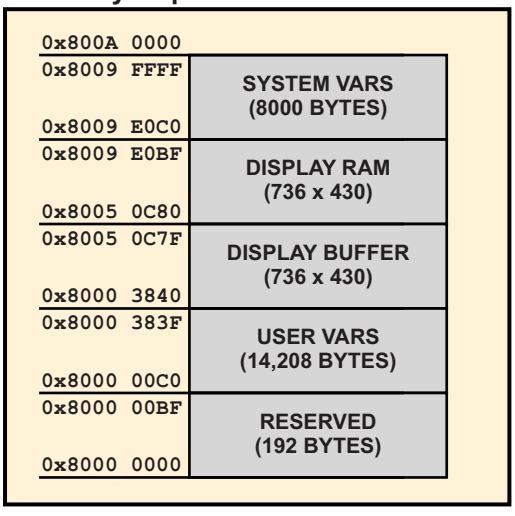
The firmware version 5.2 CPU has 640K bytes of RAM total. This RAM holds the kernel system variables, the user variables (application program variables), the display RAM and the display RAM buffer. The display RAM is a block of memory located at any address in the 640K RAM space which holds the RGB color data which the GDC constantly and automatically writes to the VGA output. As explained at the beginning of this manual the display uses a color palette which selects any 256 colors from the total color space of 65,536 colors. So there is one byte needed for each pixel displayed. Therefore, a display resolution of 640 x 480 requires 300K bytes of display RAM. The display buffer is a separate block of memory which also holds RGB data, however, it is not used by the GDC. The display buffer can be the same size or larger than the display RAM. The Microva BASIC graphic commands write to the display buffer not to the display RAM. The Refresh command then copies the display buffer to the display RAM. That's why the display is "double buffered". The application software takes a finite amount of time to write graphic data to the display buffer, however, the Refresh command executes almost instantly. Without the double buffered display the screen would appear to flicker while the graphic commands were writing data. Also, in some cases the display buffer is made larger than the display RAM so that the display buffer can hold sprites or other graphic information "off screen" until it is needed.

The Memory Map

The amount of RAM memory used in the application program = user variables + system variables + display RAM + display buffer. This total cannot exceed 640K bytes. The size of the system variables, i.e. the memory reserved for use by the firmware, is 8K bytes. Therefore, for a 640 x 480 double buffered display, the maximum user variables = 640K - 8K - 300K - 300K = 32K bytes (8,192 integer or float variables). The Microva BASIC Compiler shows the number of variables used in the application program, however, the GDC resolution can be changed in the application at run time, therefore, the compiler does not know how much RAM memory is used for the display. The compiler just assumes that the entire RAM minus the system variables is available for user variables which is 632K bytes (640K - 8K). Therefore, for the 640 x 480 display resolution, the compiler will not flag an error even if the number of user variables exceeds 8,192. ***Therefore, It is up to the application programmer to verify that the user variables will not write into RAM areas reserved for the display.***

In order to know how much RAM is available for user vars it is necessary to understand the RAM memory map. The RAM memory starts at address 0x8000_0000. The total RAM size is 640K bytes, therefore, the top of RAM is at address 0x800A_0000. The first 192 bytes of RAM and the last 8000 bytes of RAM (8K bytes total) are reserved for use by the firmware. The display RAM is located in memory just below the system vars. Therefore, the display RAM start address is always equal to: 0x800A_0000 - 8000 - X * Y. So, for the default GDC resolution of 736 x 430, the starting address of the display RAM is 0x8005_0C80. The display buffer can be located anywhere in memory, however, usually the display buffer is the same size as the display RAM and is located just below the display RAM in memory. Therefore, for the 736 x 430 resolution, the display buffer address is 0x8000_3840. That leaves the remaining RAM available for user vars which is 14,208 bytes (640K - 8K - 2 x 736 x 430) or 3,552 vars as shown in the diagram.

Memory Map For 736x430 Resolution



If more user vars are needed in the application program then the display resolution can be reduced or the display can be changed to "single buffered" which means the display buffer and the display RAM are the same block of memory. That's done by writing the display RAM address to the DisplayBufPntr system var. When the display is single buffered the graphic drawing commands (Print, Triangle, Rectangle, etc) will draw directly to the display RAM and, depending on how often the display is drawn, there may be a noticeable screen flicker. Also, when the display is single buffered the Refresh command is not used, since it is only used to copy the display buffer to the display RAM. For some resolutions the display must be single buffered since there is not enough memory for the display RAM and a separate display buffer (for example, the 800 x 600 resolution). Its also possible to make the display buffer larger than the display RAM. That might be done to hold sprites in the display buffer which are copied to the display RAM using the Blit command when an event occurs. The example on the next page shows how to "side scroll" the display by making the display buffer over six times wider than the display.

Appendix B (page 2/4)

The GDC Color Palette

As explained in the section on the color palette at the beginning of this manual, all colors use the 16 bit R5/G6/B5 format but only 256 colors can be displayed at the same time, i.e. the color of each pixel in the display RAM is represented with one byte. This is accomplished by using a color palette. Firmware version 5.2 adds the restriction that color index 0 in the palette must be set to black. That's because the GDC outputs color 0 during the horizontal and vertical blanking intervals. The Image command automatically changes color 0 to black in the image color palette which is explained in detail on the Image command page.

The GDC Border Color

As explained on the previous page, it is possible to configure the GDC such that there is a passive border around the live screen pixels. The application program can change the border color by writing to the predefined constant GdcBorderColor which is the GDC memory address of an integer value which holds the display border color. Normally, the border color format is very simple as shown in the figure below. However, due to an error in the GDC hardware module, the border color format changes depending on the colors loaded into the color palette. For the default color palette SysColors (shown at the beginning of this manual) the border color uses the R3/G3/B2 color format (i.e. red = 0 to 7, green = 0 to 7 and blue = 0 to 3) as shown in the figure below. So to set the border color to 100% yellow the command is: `WriteInt 0xE01C0000, GdcBorderColor`

However, when other color palettes are loaded, for example when an image is loaded, the border color format changes in a manner that is not well defined. In all cases byte N + 3 holds the red data, byte N + 2 holds the green data and byte N + 1 holds the blue data. But which bits are most/least significant is not clear. Therefore, for color palettes other than SysColors there will need to be some trial and error in order to determine the correct GdcBorderColor value to write for the color desired.

Border Color Data Formats

Border Color Format From the CPU Datasheet

BYTE N+3								BYTE N+2								BYTE N+1								BYTE N+0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	00	00	00	00	00	00	00	00

Border Color Format With the SysColors Color Palette

BYTE N+3								BYTE N+2								BYTE N+1								BYTE N+0							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
R2	R1	R0	00	00	00	00	00	00	00	00	G2	G1	G0	00	00	00	00	00	00	00	00	B1	B0	00	00	00	00	00	00	00	00

```
Example:  WriteInt  (Red{29) + (Green{18) + (Blue{8), GdcBorderColor
```

```
;-------------------------------------------------------------------------------------------------
;This example shows how to "side scroll" the display. LiveX = 320 and LiveY = 240 with no border.
;The display RAM size = LiveX * ResY = 320 * 480 = 153,600 bytes (since ResY = 2 * LiveY).
;The display buffer size is set to 2000 x 240 = 480,000 bytes (120,000 integers) which is 6.25
;times the display width. So the amount of free RAM = 640K - 8K - 480,000 - 153,600 = 13,568 bytes.

    Dim J, DisplayBuf(120000) As Integer
  ;change the display mode
    NewGdcMode  0, 320, 240
    WriteInt  addr(DisplayBuf), DisplayBufPntr
    WriteInt  2000, DisplayBufWidth
    WriteInt  240, DisplayBufHeight
  ;draw the screen
    Cls  blue
    Triangle 0, 0, 0, 239, 1999, 239, ltblue
    Print   400, 80, white, "THIS SHOWS HOW TO SIDE SCROLL THE DISPLAY"
    Print  1200, 80, white, "THIS SHOWS HOW TO SIDE SCROLL THE DISPLAY"
  ;scroll forever
    While 1 = 1
       For J = 0 to 420
          Refresh  J * 4, 0
          Delay 30
       Next J
       Delay 3000
    Wend
;-------------------------------------------------------------------------------------------------
```

Appendix B (page 3/4)

How To Change The Display Resolution

Subroutine NewGdcMode can be used to change the GDC module to any of several different display timings and resolutions. The source code is shown on the next page. The format for the command is: NewGdcMode Mode, LiveX, LiveY Each line in the table GdcTimingData holds the display resolution and the horizontal and vertical synchronization timing values. The X columns in the table are in pixels and the Y columns are in lines. Columns ResX and ResY set the display resolution. The NewGdcMode subroutine requires parameters LiveX and LiveY which define the live area on the screen, i.e. the pixels that can be written in Microva BASIC. ***LiveX must be evenly divisible by four and LiveY must be an even number.*** LiveX and LiveY can be the same as or less than ResX and ResY. LiveX and LiveY determine the display RAM and display buffer memory sizes. Normally, the display is double buffered in which case the display buffer size is equal to the display RAM size which is equal to LiveX * LiveY bytes. However, the application program can make the display buffer larger than the display RAM in order to hide off screen graphic elements. If LiveX or LiveY is less than ResX or ResY then there will be a border around the live screen area. The border color can be set to any value, as described on the previous page, however, pixels within the border region cannot be read or written. For example, the display resolution is set to 840 x 480 at reset with LiveX = 736 and LiveY = 430. Therefore, there will be a 52 x 25 pixel border on screen. For some resolutions there is not enough RAM memory for the double buffered display, such as LiveX = 800 and LiveY = 600. In that case the NewGdcMode subroutine forces the display to be single buffered and the graphic drawing commands will write directly to the display RAM. For single buffered displays the Refresh command is not used.

The Double Height Special Case

There is one special case when ResY = 2 * LiveY. In this case the Refresh command will automatically write each line in the display buffer to the display RAM twice, i.e. the display buffer is doubled in height. For example, there is no GDC mode for a resolution of 320 x 240, however, mode 0 or mode 1 could be used with half the Y resolution like this: NewGdcMode 0, 320, 240 In this case the size of the display buffer will be 320 x 240 but the display RAM size is set to 320 x 480. The "side scroll" example shown on the previous page shows how this is useful. To summarize, normally the display RAM and the display buffer are both set to LiveX * LiveY bytes. However, in the special case where ResY = 2 * LiveY then the display RAM size is set to LiveX * ResY and the display buffer size remains LiveX * LiveY.

Horizontal and Vertical Sync Timing

Some display modes in the GdcTimingData table have two versions with the same resolution but with different sync timing (like mode 0 and 1). That's because some displays require different timing even with the same resolution. Modern VGA display monitors can usually sync to just about any timing values, however, some older monitors require specific timing. Therefore, if a GDC mode does not display properly on the monitor, usually a different mode can be selected which will work properly. The horizontal and vertical timing values in GdcTimingData were taken from the document *"VESA and Industry Standards and Guidelines for Computer Display Monitor Timing"* which can be found on the Internet.

The first column in the table (Divide) sets the pixel clock frequency (the DotClk) for the GDC. The equation is: DotClk = SysClk / Divide, where SysClk is 200MHz. The Divide value must be even and the maximum DotClk frequency is 50MHz. For example, the GDC mode 3 DotClk = 20MHz. It is not always possible to use the exact DotClk frequency specified in the standard. For example, the standard DotClk frequency for display mode six is actually 25.175MHz, however, the GDC sets the DotClk to 25MHz and the timing values are slightly adjusted to compensate. Also, It is desired that when a 100 wide x 100 high rectangle is drawn to the screen that the result on the display is a square, i.e. the pixels are square. Several factors determine whether the display pixels are square including the aspect ratio of the monitor, the DotClk frequency and the X/Y resolutions. Modern display monitors have an aspect ratio of 16:9. Older computer monitors use an aspect ratio of 4:3. For the same mode and resolution, a square drawn on the 16:9 monitor will not appear square on the 4:3 monitor and vice versa. The default GDC resolution for firmware 5.2 is 840 x 480 using timing mode 8. This resolution was chosen so the display pixels are square on a 16:9 monitor. The command to change the display to the default configuration is: NewGdcMode 8, 736, 430

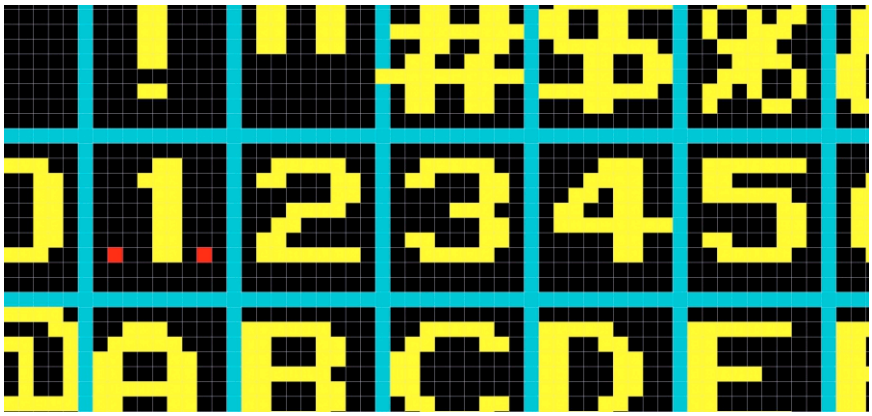
Appendix B (page 4/4)

```
;-------------------------------------------------------------------------------
 Table GdcTimingData
; Divide ResX  ResY FPorchX SyncX BPorchX FPorchY SyncY BPorchY Mode
; ------ ----  ---- -------- ----- -------- -------- ----- -------- ----
   16,   320,  480,    8,    64,     8,    10,     2,    29     ;0
   16,   320,  480,    8,    48,    24,    10,     2,    33     ;1
    8,   400,  600,   28,    60,    32,    37,     6,    23     ;2
   10,   400,  600,   20,    64,    44,     1,     4,    23     ;3
   12,   418,  480,    9,    55,    55,     6,     8,    23     ;4
    8,   640,  480,   16,   128,    16,    10,     2,    29     ;5
    8,   640,  480,   15,    95,    44,    10,     2,    33     ;6
    4,   800,  600,   56,   120,    64,    37,     6,    23     ;7
    6,   840,  480,   15,   111,   109,     6,     8,    23     ;8
 End Table
;-------------------------------------------------------------------------------
; This changes the Graphic Display Controller (GDC) hardware module output sync
; timing and resolution using the "GdcTimingData" in the table above. See appendix
; in the "Microva BASIC Programming Reference Manual" for info on the GDC operation.
; The command format is:  NewGdcMode  ModeNum, LiveX, LiveY
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; Ex. Set the GDC to mode 5 with a 640 x 480 live pixels:  NewGdcMode  5, 640, 480

 Constant  CfgCon2     = 0xBF8000F0
 Constant  RefO5Con    = 0xBF801300
 Constant  GdcMode     = 0xBF8EA000
 Constant  GdcRes      = 0xBF8EA00C
 Constant  GdcFPorch   = 0xBF8EA014
 Constant  GdcSync     = 0xBF8EA018
 Constant  GdcBPorch   = 0xBF8EA01C
 Constant  GdcL0Mode   = 0xBF8EA030
 Constant  GdcL0Start  = 0xBF8EA034
 Constant  GdcL0Size   = 0xBF8EA038
 Constant  GdcL0Addr   = 0xBF8EA03C
 Constant  GdcL0Stride = 0xBF8EA040
 Constant  GdcL0Res    = 0xBF8EA044
 Constant  GdcInt      = 0xBF8EA0F8

 Dim  GdcData, GdcAddr, GdcModeNum, GdcLiveX, GdcLiveY, GdcL0ResY As Integer

 Sub  NewGdcMode(GdcModeNum, GdcLiveX, GdcLiveY)
    WriteInt   0, RefO5Con           ;turn off the GDC dotclock
    WriteInt   0, GdcMode            ;turn off the GDC module
    WriteInt   0, GdcL0Mode          ;turn off the GDC layer 0 overlay
    WriteInt   0xC0000000, CfgCon2   ;enable GDC output pins using R5/G6/B5 mode
    GdcAddr = addr(GdcTimingData) + 36 * GdcModeNum    ;GdcAddr = pntr to GdcTimingData
    GdcData = (rdi(GdcAddr + 4) { 16) + rdi(GdcAddr + 8)
    WriteInt  GdcData, GdcRes
    GdcData = (rdi(GdcAddr + 12) { 16) + rdi(GdcAddr + 24) + GdcData
    WriteInt  GdcData, GdcFPorch
    GdcData = (rdi(GdcAddr + 16) { 16) + rdi(GdcAddr + 28) + GdcData
    WriteInt  GdcData, GdcSync
    GdcData = (rdi(GdcAddr + 20) { 16) + rdi(GdcAddr + 32) + GdcData
    WriteInt  GdcData, GdcBPorch
    WriteInt  GdcLiveX, GdcL0Stride
    WriteInt  GdcLiveX, DisplayBufWidth
    WriteInt  GdcLiveY, DisplayBufHeight
    GdcL0ResY = GdcLiveY    ;GdcL0ResY = GDC layer 0 Y resolution (usually = GdcLiveY)
    WriteInt  0, GdcInt     ;b31 of GdcInt tells Refresh command f/w whether to double LiveY
    If rdi(GdcAddr + 8) = 2 * GdcLiveY Then
       GdcL0ResY = 2 * GdcLiveY
       WriteInt  0x80000000, GdcInt
    End If
    GdcData = (GdcLiveX { 16) + GdcL0ResY
    WriteInt  GdcData, GdcL0Res
    WriteInt  GdcData, GdcL0Size
    GdcData = ((rdi(GdcAddr + 4) - GdcLiveX) { 15) + ((rdi(GdcAddr + 8) - GdcL0ResY) } 1)
    WriteInt  GdcData, GdcL0Start
    GdcData = 0xA0000 - 8000 - GdcLiveX * GdcL0ResY
    WriteInt  GdcData, GdcL0Addr
    GdcData = GdcData - GdcLiveX * GdcLiveY
    If GdcData < 192 Then GdcData = GdcData + GdcLiveX * GdcLiveY
    WriteInt  GdcData | b31, DisplayBufPntr
    WriteInt  (rdi(GdcAddr) { 15) | b15, RefO5Con
    WriteInt  0x98100100, GdcMode
    WriteInt  0x80FFA300, GdcL0Mode
 End Sub
;-------------------------------------------------------------------------------
```

Appendix C How to Make a New Font

The Microva BASIC Compiler IDE includes a utility that makes it easy to make a new font. The font source file is just an ordinary full color (24 bit per pixel) device independent bitmap (.bmp) file. Characters in a font have variable width and height. Any name can be used for the source image file. See the Print command page for the font naming convention. The output file (the font file) will have the same name as the source image bitmap except with the file extension ".mvf". The source bitmap should be divided into 16 columns and 2, 6 or 12 rows. Therefore, there will be 32, 96 or 192 character blocks in the source image. The 32 character blocks correspond to character codes 0x20 thru 0x3F in Appendix A. The 96 character blocks correspond to character codes 0x20 thru 0x7F. And the 192 character blocks correspond to character codes 0x20 thru 0x7F plus 0xA0 thru 0xFF. Any character in the font can be skipped by leaving the character block blank. For example, this is the bitmap source image for font F0809H.mvf:

The utility program only recognizes two colors: yellow and red. Any other color in the source image is ignored. Yellow is used for the text pixels that will be drawn to the display. Red can be used to control the character spacing, but is often not needed. The position and spacing of the character's yellow pixels within the character block does not matter except for the space from the top of the block to the first line in the character. That space is known as the vertical offset and is used by the utility software to align all characters vertically so their baselines match. Note that the number of pixels to the left of the yellow pixels within a character block does not matter and is not used to create the font. Likewise, the pixels below or to the right of the yellow pixels does not matter. Only the pixel spacing above the yellow pixels within a character block, the vertical offset, is used in the font. The character width is determined by the widest line of yellow pixels in the character block. For example, in the image below, the width of the number 3 is seen to be 7 pixels and the width of the number 4 is 8 pixels. In both cases the vertical offset is 2 pixels.



The Print command is used to draw text to the display buffer. After each character is drawn to the display buffer the screen position is moved to the right by the width of the character plus the number of pixels in the font character spacing (kerning) parameter in the font. So if the font kerning was 3, after the number 4 is drawn, the new screen position will have moved 11 pixels right. The compiler IDE includes a utility that shows these spacings for any font selected. The font kerning, word spacing and line spacing are all specified when the font is created using the utility software. Red pixels are used to change the character width without altering the character appearance. For example, the image above shows that the number 1 has been modified using the red pixels. Without the red pixels the number 1 width would be 3. With the red pixels the number 1 width is 7. That changes the character spacing when the character is printed to the screen. One or more red pixels can be drawn anywhere within the character block in order to change the character width. For most fonts the red pixels are not needed.