User Manual

Version 07/08 E BA001 ProgRef
Rev 1.21

CE

Programmer´s Reference

M 2.0 / Station 2.0
M ADV / Station ADV

Control Units

Content

INTRODUCTION

ABOUT THE M 2.0 / M ADV CONTROL UNITS

Small microcontrollers can everywhere be found. They mostly are programmed in assembly language which requires a deep knowledge in this language and processor architectures. The manual for the controller counts most times some hundred pages and is hard to understand. To learn the assembly language you additionally have to study books about. BASIC++ is the language to program these microcontrollers without special knowledge. Around the core of standard BASIC instructions are few additional instructions to use the controller standard environment hardware for applications (such as a LCD). For experienced programmers BASIC++ offers features (such as import code from a library) which are a must for a modern programming language.

Memory
The M 2.0 / M ADV Control Computer have a operating system on board and can be programmed in BASIC. BASIC ist not only easy to learn, it is very efficient concerning the memory usage too. A BASIC program needs about 1/5 of memory that would be needed if the same application is programmed in assembly language.

Speed:
The statement, BASIC is slow is from a time when the BASIC program is put to memory as source code which which must be read and interpreted at runtime. At the M 2.0 / M ADV Control Units the BASIC source code is converted to Tokens and loaded into the memory, representing the program. This Tokens are executed very fast resulting in a speed of app. 23000 instructions per second on a M 2.0 / M ADV Control Unit.

ABOUT BASIC++

Code Development
If you know that a single BASIC instruction sometime consists of many hundred assembly instructions it is easy to believe that BASIC programming saves a lot of time. I guess you need 1/10 of time programming in BASIC instead in assembly language. The BASIC++ programmable Control Units are a powerfull tool for rapid development even for experienced engineers. While others are still debugging their assembly or C++ program, your application is just running

Programming Skill
BASIC++ is even for unexperienced users simple to use (simple BASIC core functions). Beside of this professional programmers will find all the features required for a modern programming language. A bunch on programs, available in the library, makes it easy to solve even complex problems.

Please look here for addons and product specifications www.Spiketronics.com

ABOUT THIS DOCUMENT

This document is applicable on the Control Units M 2.0 / M ADV
This Document is also applicable for the RAIL versions of this Control Units.,
It describes the BASIC++ Instruction Set and Controll Unit properties. As far as it is useful, a general syntax is given.

Syntax: *Variable = value1 MOD value2*

Variable: Variable of Byte oder Word type
value1 Variable, value oder constant of Byte oder Word type
value2 Variable, value oder constant of Byte oder Word type

The values (here *value1* and *value2*) may also be complex terms closed in braces.

Variable= *(SQR(value*value))* MOD *value2*

This is legal when the result of the term matches the mentioned type
Additional to the Syntax description an example is shown. Please note that the required definitions are not mentioned on samples trougout the document.

Definiton:

```
DEFINE MyWord as word
DEFINE MyByte1 as byte
DEFINE  MyByte2 as byte
DEFINE MyBitVar as bit
```

Example:

```
MyWord=MyByte1 MOD MyByte2
```

DECLARATIONS AND DEFINITIONS

Variables and Port registers are memory locations in the controller. While User Variables are freely usable the Port Registers can be considered as System Variables with defined functions and names.
The content of this memory locations can be changed or requested during program runtime. They are a very important component in all programming languages. Depending on the variable type it may contain different values. BYTE type variables contain values from 0 to 255, and they occupy one byte of memory inside the controller. Once a memory location or port is defined its content can be changed by assignments of other variables content, constants or terms.

```
DEFINE MyBitport1  as PORT[1]
DEFINE MyByte as 1
```

```
MyBitport1 = ON
MyByte1 = MyByte1*10
```

PROGRAM MEMORY

The M 2.0 has almost 10kB of program memory. Memory that is not occupied by a program can be used can be used as non volatile data memory. The Control Unit ADVANCED has an extended program memory. 22kB are available for program memory or data saving. No declarations concerning the memory have to be done

USER VARIABLES - COMPILER DEFINED

The M 2.0 supports 140 bytes of variables memory.The M ADVANCED memory for variables has been extended, but with some small restrictions. Unrestricted usable are 140 bytes, up to 240 bytes are usable if the File Function PRINT# is not used, or is used that way, (e.g for temporary variables) that it does not matter if variables content in the range from 140 to 240 will be changed when INPUT# is used.
At BASIC++ you have to consider special cases. Local and global variables define the valid area of this variables within the entire program or within single functions. The valid area has to be declared with DEFINE.

Any declaration of a variable within a function causes the variable to be a local type. Any other declaration will result in global variables. More can be found in the chapter „Functions“.

Prior to use a variable has do be declared otherwise an error message will occur at compilation. Usually the compiler cares the distribution of variables to the memory. In special cases the user is able to define certain variables to be located at certain memory locations.

OPTION FLOAT

This compiler instruction is importand if you use the M ADV Control Unit. It reserves 16 bytes of variable memory for the FP accumulators and some help registers. Examles for usage u will find in the chapter FLOATING POINT MODULE

BIT Variables
Eight defied Bit Varaibles(value ON or OFF) occupy one Byte of User Memory

```
define MyBit as bit
```

BYTE Variables
Byte (value 0 ... 255) is the smallest numeric data type, ocupies 1 Byte = 8 Bit

```
define MyByte as byte
```

WORD Variables
Word (value -32768 ... 32767) this data type occupies 2 Byte = 16 Bit

```
define MyWord as word
```

USER VARIABLES - USER DEFINED

In special cases it may be required to have certain varibales at certain memory locations within the memory. The user has to care that sharing memory locations by BIT,BYTE and WORD does not alter memory location in an unwanted way, i.e. BIT[8] is shared with the memory locations of BYTE[1] and WORD[1]

{BYTEOFFSET}

This compiler instruction defines the starting point of compiler controlled variable memory managing.
The example shows how certain variable locations can be reserved. Here the locations Byte[1] and Byte[3] are reserved, for e.g a system driver.

```
define MyByte1   as Byte[1]
define MyByte2   as Byte[2]
{BYTEOFFSET 3}
define MyByte3 as Byte
```

BIT Variables
The memory locations may be the numbered BIT locations from the memory bytes 1 to 140. Because the number of BIT variables is restricted to 256, not all Memory bytes can be used for BIT Variables. The Last BIT variable
number 256 is located at the most upper bit and has it's location at byte 32 in memory
Eight defied Bit Varaibles(value ON or OFF) occupy one Byte of User Memory

```
define MyBit as bit[1]
```

Valid is BIT[1] to BIT[256]

BYTE Variables
Byte (value 0 ... 255) is the smallest numeric data type, ocupies 1 Byte = 8 Bit

```
define MyByte as byte[1]
```

Valid is BYTE[1] to BYTE[140] (240 for M ADV UNITS)

WORD Variables
Word (value -32768 ... 32767) this data type occupies 2 Byte = 16 Bit

```
define MyWord as word[1]
```

Valid is WORD[1] to WORD[70] (120 for M ADV UNITS)

EXTERNAL USER VARIABLES

The Control Units are able to adress BYTE sized memory locations at an external I²C-Bus EEPORM directly. The EEPROM always has to be set to address 160 in this case.

Define MyByte as CHIPRAM	User Variable external memory, compiler controlled
Define MyByte as CHIPRAM[1]	User Variable external memory, user controlled

FLOAT Variables
While the M 2.0 has just BYTE and WORD Types, even for non professionals easy to handle, the M ADV has a new Data Type Float. It consists on one byte exponent and 3 byte mantissa with sign.
Therefore a floating point value occupies 4 bytes (32 bit).
Please see the chapter FLOATING POINT MODULE for details.

CONSTANTS

In programming, a constant is a value that never changes. The other type of values that programs use is variables, symbols that can represent different values throughout the course of a program. A constant can be
- a number, like 25 or 3.6
- a character, like a or $
- a character string, like "this is a string"

```
const MyConstant = 122
```

Constants never change at runtime, they are constant.
Numeric constants are generally decimal system based but BASIC++ supports the binary,octal, hexadecimal system also. See the next chapter for details.

POINTER

INTERRUPT VECTOR

There is one Interrupt Vector INTERRUPT available that serves for the immediate reaction either on an external request (negative edge at IRQ) or an internal request (e.g Timer Interrupt). See Chapter CONFIG REGISTER for further information. Any user interrupt is inhibited if no interrupt vector is defined.

```
INTERRUPT MyInterruptService
```

DATA TYPES

Except the Floating Point Module (contained only in Control Unit M ADV) there are only 3 Data Types used Bit, Byte and word.
A bit status is alwas defined by boolean ON (true=logic high) and OFF(false=logic low). A byte or word content is always defined as a value which can be expressed in different number systems.The applied system has to be identified if other than decimal system based constants are used:

Examples for different number systems

01011101b	binary system
123o	oktal system
1AFh	hexadecimal system
1000	decimal system
ON	boolean true (numerical 255 at byte values and -1 at word values)
OFF	boolean false(numerical 0 for byte and word values)

EXTERNAL FILES

IMPORT

The Import keyword causes the compiler to insert external files (Basic Code, Tables, ...) during compilation. This is great for complex programs and makes your source code better readable. Any File with any extension is insertet, but must of course be a BASIC Source code or table, readable by the compiler.

Syntax: Import [File]

```
IMPORT "lib\MYFILE.BAS"
```

SYSCODE

The Control Units are able to load dedicated driver or system extensions. Two pages each 256 bytes in size are located at the end of proram memory. The files always have the ectension .S19

Syntax: SYSCODE [File]

```
SYSCODE "lib\MYFILE.S19"
```

SPECIAL PORTS

Special digital ports
All special digital ports of the unit are connected to pull up resistors as far as they are inputs. Special Ports are fixed concerning their usage and their Function.

START and RESET

This ports may be connected to push buttons if a manual start or stop of user programs is required. Push the START button to run a BASIC program, downloaded previously to the unit's memory. Press RESET to stop a running program and entering the download mode.
To ensure proper function, the AUTOSTART jumper at the unit must be removed. (JP2)

The START port status can be requested using the CONFIG REGISTER.

FREQ1 and FREQ2

This ports are always and exclusively inputs. The primary operation is frequency counting in the range from 0 to 32kHz. FREQ 1 supports the feature to synchronize the system clock if a DCF77 receiver module is connected to this port. The synchronization is done automatically in the background, i.e. no user action is required. The receiver module has to provide a open collector output to switch this port lo. Use shielded cables to connect the receiver module to the unit. This ports may be used alternatively in Event counting mode.See chapter CONFIG REGISTER for details. The trigger is negative edge level sensitive for all modes.

FREQ1 and FREQ2 are predefined. No user definition has to be done

IRQ

The IRQ port is always input and serves for the immediate reaction on an external request i.e the current operation is interrupted an the external event is serviced by a appropriate Interrupt Routine. The IRQ input is negative edge triggered.
The interrupt source can be canged e.g to an automatically generated timer interrupt. Moreover the IRQ line can be used as input port when the IRQ is not used as interrupt source.

IRQ is predefined as INTERRUPT. The user has to define the Interrupt vector INTERRUPT

BEEP

The BEEP port is always an output an usually connected to a piezo buzzer. It serves for program or alert status indications during normal program operation. During debugging your program this buzzer may be useful also. The audio frequency ranges from 10kHz to 100Hz.

The BEEP Port is predefined. No user definition has to be done

RXD/TXD

The serial interface (RXD=input TXD=output) is designed for 5V digital logic level, You never must connect a RS232 interface directly to this ports. This would immediately result in permanent damage to the Control Unit

RXD and TXD are predefined. No user definition has to be done

STANDARD DIGITAL I/O PORTS

The Control Units provide two standard byteports (16 bitports, P1 to P16). Each bitport can be used as input or output port. Both byteports are provided with software switchable pull up resistors (30k). Each of the eight analog ports (A/D-converter) can be alternatively used as a standart digital I/O port if this function is enabled in the configuration register (see Chapter CONFIG REGISTER) Each of this ports can then be considered as standard bitport (P17 to 24) but has no switchable pull up resistor. After reset all ports are usually Inputs. Please note that some options my change this (See Chapter CONFIG REGISTER). A DEACT instruction will make the port an INPUT, a write instruction will make the port an output . See Chapter INSTRUCTIONS AN KEYWORDS for informations on read and write a port.

```
define MyBitPort1        as PORT[1]
define MyBytePort2   as BYTEPORT[2]
```

Valid are Port[1] to Port[16]
Valid are Byteport[1] to Byteport[2]

DIGITAL INPUT PORT

Digital input ports are used to request an external switch status. A digital input is on undefined logical level if nothing is connected (eg. an external switch is open) Therefore it is recommended to tie the port to a defined level eg. conncting a pull up resistor to the port. In this case a closed switch causes the port reading as "false" (logical lo level) and an open switch will be read as "true" (logical hi level) After applying the operation voltage or after entering the reset state all ports are switched to inputs.
Internal pullup resistors can be activated by the CONFIG 1 REGISTER and may save you some componets in your project. As well as for other often needed procedures, the library already contains the program modules for switching the pullups.

DIGITAL OUTPUT PORT

If a digital port is switched to output you can connect circuits, transistors or LEDs with current limiting resistors. The maximum load (output) current of each port must not exeed 10mA. In all cases a current limit (e.g by connecting a resistor) has to be ensured. Otherwise a immediate and permanent damage of ports can result. The port function (if a port is input or output) is controlled during program execution. After applying the operation voltage or after entering the reset state all ports are switched to inputs.

DEDICATED DIGITAL PORTS

The operating system offers some special functions which occupy dedicated ports. Other than the special ports, the dedicated ports can be used as Standard digital I/O Ports. They become dedicated if special functions (eg. read/write to the Extended Ports or r/w to the Infra Red Interface)

LCD INTERFACE (PORT 9 TO 16)

As soon as the LCD Interface is initialised with LCD.INIT the Ports 9 to 16 are occupied and are not usable freely any more. The LCD driver located in the operating system lets u comfortable output data and text to any HD44100 compatible LCD.

I²C-BUS INTERFACE (PORT 9 and 10)

Any usage of the I²C Functions (see chapter I²C OBJECT) will occupy this ports to work as I²C Bus Interface and the ports can not be used freely any more. System Functions (e.g CHIPRAM or CHIP CARD BOOT) will also need the appropriate hardware connected to this ports.

RC5 IR INTERFACE (PORT 2 and 3)

Any usage of the RC5 IR Functions (see chapter IR OBJECT) will occupy this ports to work as RC5 coded R Interface and the ports can not be used freely any more.

RF INTERFACE (PORT 2 and 3)

Any usage of the RF Functions (see chapter RF OBJECT) will occupy this ports to work as special coded RF Interface and the ports can not be used freely any more.

EXTENDED PORTS

The Extended Ports are not really property of the Control Units but they are treated this way by the Operating System. Therefore a connected IIC-Bus PCF8574 Digital Ports circuit can be accessed in the same simple way as the standard ports. See Chapter INSTRUCTION AND KEYWORDS for informations on read and write a port.

```
define MyBitPort17          as PORT[17]
define MyBytePort4          as BYTEPORT[4]
```

Valid are Port[17] to Port [144]
Valid are Byteport[3] to Byteport[18]

You can use the Extended Ports if a PCF8574 is connected to the IIC-BUS. For this case you may not use the IIC-BUS Object, because the PCF8574 is fully supported by the Operating System This integrated circuit provides you 8 I/O Ports. The port index (or port number) depends on the address given to the PCF8574 The address consists of a 4 bit fixed and 3 bit user selectable address space. The LSB defines as for I²C-Bus devices usual the read/write action. Removing a jumper will cause the corrsponding address bit to be HI.

```
0  1  0  0  x  x  x  1
|-------------|-----------|----|
FIX ADR       ADR    R/W
```

```
0  1  0  0  x  x  x  0
|-------------|-----------|----|
FIX ADR       ADR    R/W
```

READ- Operation, LSB HI **WRITE- Operation, LSB LO**

The table shows the relation between address and port assignment

PCF 8574	**ADR 0**	**Ports 17 - 24**	**BYTEPORT 3**
PCF 8574	**ADR 1**	**Ports 25 - 32**	**BYTEPORT 4**
PCF 8574	**ADR 2**	**Ports 33 - 40**	**BYTEPORT 5**
PCF 8574	**ADR 3**	**Ports 41 - 48**	**BYTEPORT 6**
PCF 8574	**ADR 4**	**Ports 49 - 56**	**BYTEPORT 7**
PCF 8574	**ADR 5**	**Ports 57 - 64**	**BYTEPORT 8**
PCF 8574	**ADR 6**	**Ports 65 - 72**	**BYTEPORT 9**
PCF 8574	**ADR 7**	**Ports 73 - 80**	**BYTEPORT 10**
PCF 8574A	**ADR 0**	**Ports 81 - 88**	**BYTEPORT 11**
PCF 8574A	**ADR 1**	**Ports 89 - 96**	**BYTEPORT 12**
PCF 8574A	**ADR 2**	**Ports 97 - 104**	**BYTEPORT 13**
PCF 8574A	**ADR 3**	**Ports 105 - 112**	**BYTEPORT 14**
PCF 8574A	**ADR 4**	**Ports 113 - 120**	**BYTEPORT 15**
PCF 8574A	**ADR 5**	**Ports 121 - 128**	**BYTEPORT 16**
PCF 8574A	**ADR 6**	**Ports 129 - 136**	**BYTEPORT 17**
PCF 8574A	**ADR 7**	**Ports 137 - 144**	**BYTEPORT 18**

Using the IIC-BUS the LCD always has to be initialised even it is not used:
LCD.INIT
LCD.OFF

EXTERN PORT 17-24 (BYTEPORT 3) is disabled when the AD-ports are used in digital mode.

ANALOGPORTS

The Control Units offer eight A/D ports and two D/A-ports (witch servo drive capabilities as alternate function). The maximum input conversion range of the A/D-converters is fixed by the applied reference voltage. The maximum output voltage of the D/A-converters is independent of the reference voltage and always as high (and accurate) as the operating voltage.

Reference Voltage:
Before using the A/D-converters, the reference voltage has to be connected with the reverence voltage input of the Control Units. This voltage defines the maximum input voltage applied to the A/D converters and will cause a A/D conversion result of 255. For the most applications the 5V operating voltage is sufficient accurate and can directly be used as reference voltage. (Jumper REF onboard the Unit) If more precission is required, an external reference voltage can be applied to the Uref input. All measurements of the A/D converters are related to GND

A/D CONVERTER PORTS

All kind of sensors may be connected to the A/D-ports, if they match the maximum A/D input voltage. The A/D converters have 8 bit resolution i.e one digit corresponds to 19.6mV. Protect the A/D-ports with a 10k serial resistor if the input voltage applied to the ports can exeed voltages above 5V. This resistor will not affect the conversion accuracy and provides a over voltage protection up to 12V.See chapter CONFIG REGISTER for informations on use the AD ports in digital mode and chapter INSTRUCTIONS AND KEYWORDS for informations on reading an analog port

```
define MyAnalogIn          as AD[1]
```

Valid are AD[1] to AD[8]

D/A CONVERTER PORTS

The two 8 bit D/A converter are PWM (Pulse Width Modulated) converters. The output pulse consissts of 256 seperate sections switched to logic LO or HI related to value of the D/A conversion output. If a conversion output of 128 is required, 128 sections are set to HI and the remaining 127 sections are hold LO. This waveform is repeated at a rate of 1930Hz, each single section is of 2us width. To convert this PWM signal into a true analog value a simple RC low pass filter is working fine. Attention has to be paid to the remaining ripple (deviations from ac onstant output voltage, varying with tim) which depend on the load, connected to the RC filter. For more precision an active circuit is recommended. Driving lamps or LEDs with this PWM do not require a filter because the repetition rate is to fast to be realised as flickering of light. Please note that a filtered PWM output is not exactly at zero volts if the D/A converter output is set to zero. The reason is that the port output lo voltage is approx. 50mV and a 2us pulse is remaining at the D/A output even the converter is programmed to a zero output. In this case the filter output will be around 70mV for a D/A converter output programmed to zero. The D/A-ports can be programmed for an alternate servo drive function. See chapter CONFIG REGISTER for informations on use in Servo Mode and chapter INSTRUCTIONS AND KEYWORDS for informations on writing to an analog port

```
define MyAnalogOut         as DA[1]
```

Valid is DA[1] and DA[2]

SYSTEM PROPERTIES

System Properties are predefined Symbols and can be usually be read or write

BEEP

The "Beep" Instruction generates a square wave output at the Beep port. The BEEP port is always an output and usually connected to a piezo buzzer. He serves for program or alert status indications during normal program operation. During debugging your program this buzzer may be useful also. The audio frequency ranges from 10kHz to 100Hz. The Tone parameter may range from 1 to app 100. For larger values the change in frequency is almost not audible.

Syntax: *Beep Tone, duration, pause*

```
BEEP 10,100,20
```

Tone
The Tone audio frequency has to be defined. The value may be inthe range from 1 to app. 60. Values higher than 100 are possible but don't cause much difference in frequency.

Duration
Duration in 20ms steps duration in ms * 20

Pause
Pause after the tone in 20ms steps

RTC

Real Time Clock (RTC) provides the user with the system time. The RTC can be manually set or, if a DCF77 receiver is connected to FREQ1 Port it will be set to current time automatically. All values are byte values.
The "YEAR" property requests/sets the year from the realtime clock
The "MONTH" property requests/sets the month from the realtime clock
The "DAY" property requests/sets the day of the month from the realtime clock
The "DOW" property requests/sets the day of the week from the realtime clock (value 1 to 7, 1 is monday)
The "HOUR" property requests/sets the hour from the realtime clock
The "MINUTE" property requests/sets the minute from the realtime clock
The "SECOND" property requests/sets the second from the realtime clock

Example for request second

```
IF SECOND = 10 THEN GOTO X
```

Example for setting time seconds

```
SECOND = 0
```

TIMER

The Timer can be considered as system variable, containing the 20 ms timer value. The timer runs up to 32767 and stops if this value is reached. The timer can be written i.e. it can be preset to a certain value or cleared. The timer also can be used to generate 20ms interrupts. The timer content is a word value.

Example for request timer

```
IF TIMER = 10000 THEN GOTO X
MyWord=TIMER/100
```

Example for set timer

```
TIMER = 0
```

FREQ1 and FREQ2

The primary operation is frequency counting in the range from 0 to 32kHz. FREQ 1 supports the feature to synchronize the system clock if a DCF77 receiver module is connected to this port. This ports may be used alternatively in Event counting mode. FREQ values are word values.
The trigger is negative edge level sensitive for all modes.

FREQUENCY COUNTING

In mode frequency counting FREQ is a read only word value, containing the measured Frequency in 1Hz units. Counting is done at 1s gate time. For lower frequencies it may be suitable to measure events within a certain time instead

Example for request counter

```
IF FREQ1 = 10000 THEN GOTO X
MyWord=FREQ/100
```

EVENT COUNTING

In mode event counting FREQ is a read/write value. For instructions to switch the mode please see the chapter CONFIG REGISTER

Example for request FREQ1

```
IF FREQ1 = 10000 THEN GOTO X
MyWord=FREQ/100
```

Example for set FREQ1

```
FREQ1 = 0
```

INSTRUCTIONS AND KEYWORDS

INPUT/OUTPUT

Some of the available Input/output instructions can be redirected form the (default) serial interface to the Objects LCD,CONFIG,IIC,RF or IR. See Chaper OBJECTS for reference.
For Floating Point Input/Output (Unit M ADVonly) see Chapter FOATING POINT MODULE

BAUD

The Baud instruction defines the serial interface speed at program runtime. A value or constant is expected as parameter. Variables are not supported. The parameter does not corresponds directly to the baudrate, it is rather a symbolic constant for the true speed. The default speed is 9600 Baud, 8N1

Syntax: Baud Rate

```
BAUD R1200
```

Rate is a constant or a numeric value. Some baudrates can be selected with predefined constants: R1200 (1200 Baud), R2400 (2400 Baud), R4800 (4800 Baud) oder R9600 (9600 Baud)

RXD

The RXD instruction retruns TRUE or FALSE depending on the content of the serial interface buffer. The buffer is 16 bytes long.

INPUT

The Input instruction expects one or more bytes provided by the serial interface A variable is expected as parameter. The received bytes are expected to be ASCII characters. The reception is terminated if Carriage Return (&H0D) is received. The value is stored into the variable and then program execution is continued. Program execution is on hold until the input is terminated.

Syntax: Input Variable

```
INPUT Myword
```

GET

The Get instruction waits for a single byte provided by the serial interface or provided (through redirection) from one of this objects: CONFIG,IIC,IR or RF The received byte is stored into a variable. If an IR or RF Object is affected, two bytes of data (address and data byte) must be read. See Chapter OBJECTS for reference

Syntax: Get Variable

```
GET MyByte
```

PRINT

The PRINT instruction transmits strings,variables or constants of to the serial interface. A semicolon at the end of a "Print" instruction causes a Line Feed suppression. (ASCII 13, 10). For transmission of more arguments with a single"Print" instruction you may use the concatenation operator &
The Print instruction is also applicable to the LCD Objekt and the IIC Object. See the Chapter OBJECTS for reference.

Printing the character # is illegal because this character is internal used to precede a control sequence

Syntax: Print Argument [& Argument [& ...]] [;]

```
PRINT "WERT :" &  Myword & " V"
```

PUT

The PUT instruction transmits a single byte to the serial interface or (with redirection) to one of this objects: CONFIG,IIC,IR or RF. See Chapter OBJECTS for reference

Syntax: Put Variable

```
PUT MyByte
```

variable byte variable or constant

INTERNAL/EXTERNAL DATA STORAGE

In any case you want to save important system data (e.g. system configuration or aquired data) in non volatile manner you may use the EEPROM emulation. The data is stored straight sequential, and a certain location can not be read directly but all previous locations must be read before. The data stored is always in word format. Even you only store bytes a full word of Memory is used always. For Floating Point memory operations (Unit M ADV only) see Chapter FOATING POINT MODULE

OPEN# FOR READ - OPEN# FOR WRITE

Resets the data pointer to the the memory start location, and is always required prior to the first read/write operation of a sequence. This is also needed to initialise the FILEFREE value.

CLOSE#

This will save the current data pointer to be able to restore it, if data have to be append to the aready existing data. The data pointer remains saved even after reset or power off/on,

OPEN# FOR APPEND

The data pointer is set to the last writen location, ready to append data.

INPUT#

This will read the location addressed by the data pointer. The content of the data pointer is incremented after read.

Syntax: INPUT# Variable

```
INPUT# MyWord
```

variable: word/byte variable

PRINT#

This will write to the location addressed by the data pointer. The content of the data pointer is incremented after write

Syntax: PRINT# Variable

```
PRINT# MyWord
```

variable: word/byte or constant

FILEFREE

FILEFREE contains the remaining memory space in number of words. FILEFREE is not initialised before the first OPEN# is executed.

Example for request FILEFREE

```
IF FILEFREE = 0 THEN GOTO X
```

EOF

EOF contains the boolean value if the last position of a previously created data file is reached when reading. EOF is true (-1) if end of file is reached or false (0) if not
In principle it is a compare of the current datapointer and the the saved pointer (done with CLOSE#)

Example for request EOF

```
IF  EOF THEN GOTO X
```

EEPROM.WRITE / APPEND / READ

Moving just few bytes to the EEPROM there is an shorter option available, which manages file close and open automatically in background.

Syntax: EEPROM.Instruction [parameter,]

```
EEPROM.WRITE value1,value2
EEPROM.APPEND value3
EEPROM.READ value1, value2, value3
```

Parameter: word/byte variable or constant

CAUTION:
Writing to the Flash Memory is lifetime limited. The Flash has a lifetime of 100.000 write cycles. Excessive writing to Flash beyond 100.000 cycles may permanently the Controller memory within shortest times

LOOKTAB - TABLE

A Data File sometimes may be requested as read only (e.q converting sensor values by a table). The required set of bytes can be allocated to a table. The Table is embedded in the compiler generated token code and therefore read only. The internal data storage format for table values is always word format The LOOKTAB instruction reads a word from a table. Looktab may either be used as function or as instruction.

Syntax: Looktab(Table, Index, Variable)
Variable = Looktab(Table, Index)

Index: word/byte variable or constant
Variable: word/byte variable

```
LOOKTAB   MyTab,MyIndex,MyWord

TABLE myTab
0 1 2 3 4 5 6 7
END TABLE
```

CHIPRAM - DIRECT INDEXED ACCESS

Chipram offers easy direct addressed access to Microchip I²C-Bus EEPROMs connected external to the Ports 9 and 10. Please see Chapter DEDICATED DIGITAL PORTS. Applicable are EEPROMS up to 64 kByte capacity. The values transfered from or to the external EEPROM has to be a Byte Value. Please notice, that CHIPRAM uses no OBJECT, so External I²C Memory can be accessed without closing other OBJECTS prior. Also this access need no declaration of memory and variables.

Syntax: CHIPRAM(address)

Address: word/byte variable or constant

Example for reading a value

```
MvBvte=CHIPRAM(MvAddress)
```

Example for writing a value

```
CHIPRAM(MvAddress)=MvBvte
```

CHIPRAM - SYSTEM ACCESS

CHIPRAM also supports access to external memory in the same manner as for System variables, if your variables have been defined before. See Chapter DEFINITION and here EXTERNAL USER VARIABLES In thiks examples MyByte has been previously declared:

```
define MyByte as chipram
```

Example for reading a value

```
NewValue=MvBvte
```

Example for writing a value

```
MvBvte=NewValue
```

Even though it is possible to use variables, located in external EEPROM memory, it is not recommended, because the access to EEPROM Memory slows down system speed. An external memory access takes typ. 600us for read and 10ms for write.

STANDARD DIGITAL PORTS

After reset all ports are usually Inputs. please note that some options my change this (See Chapter CONFIG REGISTER). A DEACT instruction will make the port an INPUT, a write instruction will make the port an OUTPUT (ecept the ports which are read or write only)

PORT READ/WRITE
Prior to any operation of ports, a port has to be defined (see Chapter DECLARATIONS and DEFINITIONS) Reading a Bitport returns a boolean value which is ON (true) for a port that is logic high level and OFF for a port that is logic low level. Bitports may also accessed by read/write with the numerical value 0(false) and -1(true) instead of ON and OFF
MyBitPort = -1 would also work to set the port to ON but is not recommended. See Chapter DATA TYPES also.

Example for request Bitport input status

```
IF  MyBitPort1=OFF THEN GOTO X
IF MyBitPort1=ON THEN GOTO X
```

Example for request Byteport input status

```
IF  MyBytePort1=123 THEN GOTO X
```

Example for setting a Bitport output

```
MyBitPort1=OFF
MyBitPort1=ON
```

Example for setting a Byteport output

```
MyBytePort1=123
```

TOG
TOG (toggle) is an instruction which changes the current port status to its opposite. I.e a port becomes ON if it was previously OFF and reverse. Prior to use TOG, the affected port has to be switched to OUTPUT (e.g MyPort=OFF) otherwise TOG will not work. TOG is not working with Byteports and Extended Ports

Syntax: TOG(bitport)
Example for toggle a bitport

```
TOG MyBitPort
```

PULSE
PULSE is an instruction which changes the current port status for 5us and then returns to its previous status I.e a port becomes ON for a very short time if it was previously OFF and reverse. Prior to use PULSE, the affected port has to be switched to OUTPUT (e.g MyPort=OFF) otherwise PULSE will not work. PULSE is not working with Byteports and Extended Ports

Syntax: PULSE(bitport)
Example for pulse a bitport

```
TOG MyBitPort
```

DEACT
DEACT will cause an output port to return to operate as input

Syntax: DEACT(bitport)
DEACT(byteport)

Example for setting a port to input mode:

```
DEACT MyBitPort1
DEACT MyBytePort1
```

EXTENDED PORTS

Extended Ports are handled in the same manner as the standard digital ports, except the fact that PULSE and TOG are not applicable. See chapter DECLARATION and DEFINITIONS for information on external port assignment

AD PORTS

Prior to any operation of ports, a port has to be defined (see Chapter DECLARATIONS and DEFINITIONS)

Each of the eight analog ports (A/D-converter) can be alternatively used as a standart digital I/O port if this function is enabled in the configuration register (see Chapter CONFIG REGISTER) Each of this ports can then be considered as standard bitport (P17 to 24) but has no switchable pull up resistor. If the AD Ports are used in digital mode they are handled in the same manner as the standard digital ports, except the fact that PULSE and TOG are not applicable.
Reading the AD Ports in analog mode (default) will return a Byte value representing the applied voltage to the AD input.

Example for read an AD Port

```
IF MyAnalogIn = 100 THEN GOTO X
MyByte=MyAnalogIn / 10
```

DA PORTS

Prior to any operation of ports, a port has to be defined (see Chapter DECLARATIONS and DEFINITIONS)
The D/A-ports can be programmed for an alternate servo drive function. See chapter CONFIG REGISTER for informations on use in Servo Mode Writing to the the DA Ports in analog mode (default) will require a Byte value representing the average output voltage to the DA port.

Example for set a DA Port

```
MyAnalogOut =10
```

MATH FUNCTIONS

For Floating Point Math Functions (Unit M ADV only) see chapter FOATING POINT MODULE

MAX

The MAX function returns the larger one of two byte or word size values.

Syntax: *Variable = Max(value1, value2)*

```
MyWord=MAX(MyByte1,MyByte2)
```

Variable: Variable of Byte oder Word type
value1 Variable, value oder constant of Byte oder Word type
value2 Variable, value oder constant of Byte oder Word type

MIN

The MIN function returns the smaller one of two byte or word size values.

Syntax: *Variable = MIN(value1, value2)*

```
MyWord=MIN(MyByte1,MyByte2)
```

Variable: Variable of Byte oder Word type
value1: Variable, value or constant of Byte or Word type
value2: Variable, value or constant of Byte or Word type

ABS

The ABS function returns absolute value of a word or byte size value. It works for bytes but usually it makes no sense because byte values are unsigned

```
MyWord=ABS(MyWord)
```

Syntax: *Variable = ABS(value1)*

Variable: Variable of Byte or Word type
value1: Variable, value or constant of Byte or Word type

SGN

The SGN function returns 1 if the value is >0 and -1 if the value is <0. It works for bytes also but usually it makes no sense because byte values are unsigned

Syntax: *Variable = SGN(value1)*

```
MyWord=SGN(MyWord)
```

Variable: Variable of Byte or Word type
value1: Variable, value or constant of Byte or Word type

RAND - RANDOMIZE

The RAND function returns a random word value. The values will be random, but the sequence of random values will always be the same unless you use RANDOMIZE X for different initialisations.
RANDOMIZE TIMER will load the running system timer as initialisation

Syntax: RANDOMIZE *init*
Variable = RAND

```
RANDOMIZE 2000
MyWord=RAND
```

Variable: Variable of Byte or Word type
init: Variable or constant of Byte or Word type

SQR

The SQRT function returns an aproximated value of the squareroot.

Syntax: *Variable* = Sqrt(*value*)

```
MyWord=SQRT(MyWord)
```

Variable: Variable of Byte or Word type
Value: Variable, value or constant of Byte or Word type.

MOD

The MOD (Modulo) function returns the remainder of an integer division.

Syntax: *Variable = value1 MOD value2*

```
MyWord=MyByte1 MOD MyByte2)
```

Variable: Variable of Byte or Word type
value1 Variable, value or constant of Byte or Word type
value2 Variable, value or constant of Byte or Word type

MATH AND BOOLEAN OPERATORS

For Floatingpoint Operators (Unit M ADV only) see Chapter FOATING POINT MODULE
When computing terms with operators and functions the rank in hirarchy is most important. The rank is as follows

RANK	OPERATOR
9	( )
8	MATH FUNCTIONS
7	NEGATIVE SIGN
6	MULTIPLY DIVISION MOD SHL SHR
5	PLUS MINUS
4	COMPARES
3	NOT
2	AND
1	OR

+ - / * Basic Arithmetic operations

COMPARES

> (larger) <(smaller) >=(larger or equal) <=(smaller or equal) =(equal) <>(not equal)

Compares are legal for byte, word and constant values, as well as terms and functions

```
IF MyByte1 <= MyByte2 THEN GOTO X
IF  MyByte1*10>SQRT(MyByte2) THEN GOTO X
```

SHL

Bitwise logical shift left
SHL (logical shift left)is legal for byte, word and constant values. The result is equal to a multiply by 2 for every single shift.

Syntax: *Variable = value1 SHL value2*

```
MyByte1 = MyByte2 SHL 8
```

Variable: Variable of Byte or Word type
value1 Variable, value or constant of Byte or Word type
value2 Variable, value or constant of Byte or Word type

SHR

Bitwise logical shift right
SHL (logical shift right) is legal for byte, word and constant values. The result is equal to a division by 2 for every single shift.

Syntax: *Variable = value1 SHR value2*

```
MyByte1 = MyByte2 SHR 8
```

Variable: Variable of Byte or Word type
value1 Variable, value or constant of Byte or Word type
value2 Variable, value or constant of Byte or Word type

AND, OR XOR NOT

Are boolean operators used for bitwise manipulation of variables or constants.

Syntax: *Variable = value1 AND value2*

```
MyWord=MyByte1 AND MyByte2
```

Variable: Variable of Byte oder Word type
value1 Variable, value oder constant of Byte oder Word type
value2 Variable, value oder constant of Byte oder Word type

Using ths operators with IF then, they behave like a function call that accepts two parametres and returns either True or False.

```
IF (MyByte=1) OR (MyByte=2) THEN GOTO X
```

PROGRAM FLOW CONTROL

PAUSE

Holds a program for a specified time
PAUSE is the simplest instruction for flow control. It stets the program for a cerain time on hold. While a pause is active, system interrupts (e.g. System Timer) are still executed, but the user interrupt INTERRUPT is executed after pause has ended. The Pause value is 20ms units, i.e PAUSE 50 will cause a 1 second pause.

Syntax: PAUSE = value1
V*alue1:* Variable, value or constant of Byte or Word type.

```
PAUSE 25
```

FOR, TO, NEXT, STEP, EXIT FOR

FOR TO NEXT repeats a group of statements a specified number of times.

Once the loop starts and all statements in the loop have executed, step is added to counter. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the NEXT statement. The step argument can be either positive or negative. Other as usual the loop is exited at a exact match of *Counter* and *End* only. EXIT FOR will cause an early exit.
You can nest FOR NEXT loops by placing one FOR NEXT loop within another. Give each loop a unique variable name as its counter.

```
FOR MyWord=MyStart TO MyEND STEP  MyStep
PRINT MyWord
IF MyPort=OFF THEN EXIT FOR
NEXT MyWord
```

Syntax: For [Counter] = [Start] To [End] [Step [Incr]]
[Exit For]
Next

Counter: Variable used as counter.
Start: Variable, value or constant which is moved to the counter variable as start condition.
End: Variable, value or constant, the counter variable is compared with.
Incr: Required if STEP is used. Term, variable, value or constant defining the increment.

DO, LOOP UNTIL, EXIT DO

Repeats a block of statements while a condition is True or until a condition becomes True.

To create a simple program loop, you should use DO LOOP UNTIL. The loop is executed at least one time and may be quit early with Exit Do. The Until instruction at the end of the loop checks the expression to be True or False. True will cause to exit the loop.
The Exit Do can only be used within a Do...Loop control structure to provide an alternate way to exit a Do...Loop. Any number of Exit Do statements may be placed anywhere in the Do...Loop. Often used with the evaluation of some condition (for example, If...Then), Exit Do transfers control to the statement immediately following the Loop.
When used within nested Do...Loop statements, Exit Do transfers control to the loop that is one nested level above the loop where it occurs.

Syntax: Do
[instruction]
[Exit Do]
Loop (Until [expression])

```
DO
Mybyte=Mybyte+1
IF MyPort=OFF THEN EXIT DO
LOOP  UNTIL MyByte=5
```

IF, THEN, ELSE END IF

Conditionally executes a group of statements, depending on the value of an expression.

You can use the single-line form (first syntax) for short, simple tests. However, the block form (second syntax) provides morestructure and flexibility than the single-line form and is usually easier to read, maintain, and debug. A block IF statement must be the first statement on a line. The block IF must end with an END IF statement.

Syntax1: *If [expression] Then [instruction] [Else] [instruction]*

```
IF MyByte=10 THEN GOTO X ELSE GOTO Y
```

Syntax2: *If [expression] Then*
[instructions]
[Else]
[elseinstructions]
End If

```
IF MyWord=10 THEN
    PRINT "This is "
    PRINT MyWord
ELSE
    PRINT "No Match"
    PRINT "found"
END IF
```

SELECT CASE, CASE, CASE ELSE

Executes one of several groups of statements, depending on the value of an expression.

Select Case matchexpression
[Case expression]
[instructions]
[Case Else expression]
[elseinstructions]

End Select

```
SELECT CASE i
   CASE 1
      PRINT "1"
   CASE 2
      PRINT "2"
   CASE ELSE
      PRINT "No Match"
END SELECT
```

If *matchexpression* matches any *Case expression*, the statements following that CASE clause are executed up to the next CASE clause, or for the last clause, up to END SELECT. Control then passes to the statement following END SELECT If *matchexpression* matches an *expression* in more than one CASE clause, only the statements following the first match are executed.

The CASE ELSE clause is used to indicate the *elseinstructions* to be executed if no match is found between the *matchexpression* and an *Case expression* in any of the other Case selections. Although not required, it is a good idea to have a CASE ELSE statement in your SELECT CASE block to handle unforeseen match*expression* values. If no *Case expression* matches *matchexpression* and there is no CASE ELSE statement, execution continues at the statement following END SELECTSELECT CASE statements can be nested. Each nested SELECT CASE statement must have a matching END SELECT statement.

WAIT

The Wait instructionis used as level-sensitive control.
The processor waits when the expressions is False. When the expression is True, the statement is executed.
The expression (or it's result) is treated as boolean value, therefore wait responds to True and False only

Syntax: WAIT [expression]

```
WAIT (Mybitport = ON)
```

Because a bitport is always boolean
it may be written:

```
WAIT (MyBitPort)
```

GOTO

The GOTO Instruction causes a jump to a label in source code. No return address is saved

Syntax: GOTO Label

```
MyLabel:      IF (MyBitPort=OFF  THEN  GOTO MyLabel
```

The example is equal in functionas: WAIT MyBitPort.

FUNCTION

The Function keyword initialises a new function. A function may request parameter when it is called. To save memory you may use the REF keyword to create a reference for variables i.e. the parameters passed to the functions occupies the same bytes of memory as the reference. Using this reference variables you may save a lot of memory because always only the set of reference variables is used for passing the parameters to a function. You may also return a value when the function is exited.

Syntax: Function [Name] ([[Parameter] As [Data type] | [Parameter] Ref [Variable], ...])
[instructions]
Return [expression]
End Function

```
FUNCTION MyFunction(X as byte Y as byte)
      X=2*Y
      RETURN 2*X
END FUNCTION
------------------------------------------------------------
PRINT MyFunction(1,2)
```

Name: Valid identifier
Parameter: Optional. Valid variable identifier
Data type: Required if parameter variable is not a REF Type variable .
Variable: Required if parameter variable is a REF Type variable Identifier of a declared variable.
Expression Optional. Term, variable, value or constant as return value

OBJECTS

The Objects are parts of the Extended Functions. For any access to Objects belonging to the Extended Functions it is required to activate the Object prior to any read/write action. Because only one Object may be active at the same time it is required to cose the object before activating another Object. While an Object is accessed, the serial interface can not be accessed with operations like PUT,GET,INPUT,PRINT. Close an Object before an access to the serial interface.

CONFIG REGISTER
Two internal registers may be written to set up a special system configuration e.g alternate port functions or can be read to aquire a certain system status flag e.g. radio controlled clock synchronisation.

CONFIG REGISTER 1

Bit 0 - Switch both PWM-DACs to SERVO-Mode
Bit 1 - Switch frequency counter 1 to event counter mode
Bit 2 - Switch frequency counter 2 to event counter mode
Bit 3 - switch on PULLUP- resistors for PORT 1 to 8
Bit 4 - switch on PULLUP- resistors for PORT 9 to 15
Bit 5 - radio controlled synchronising of RTC. Clock is Sync
Bit 6 - IIC-BUS Communication Error
Bit 7 - Mirrors logic level at Start button input port

The mentioned action is performed on setting the corresponding bit to HI

CONFIG REGISTER 2

Bit 0 - IRQ disabled and replaced by 20 ms timer interrrupt
Bit 1 - IRQ line logic level
Bit 2 - IRQ disabled, replaced by RF-Module interrupt.
Bit 3 - IRQ disabled, replaced by IR-Module interrupt.
. . . . . .
Bit 7 - AD-Ports now acting as digital BYTEPORT 3, external IICBUS byteport 3 is disabled

The mentioned action is performed on setting the corresponding bit to HI

CONFIG.INIT
Activate the Object to get access to its registers for read/write operation. Close other Objects prior to open.

Syntax: OBJECT.instruction

CONFIG.PUT
CONFIG.GET
Access the Config register for read/write a binary value

CONFIG.OFF
Close the Object prior to open any other Object.
Syntax: OBJECT.instruction

The Example shows how to read/write a special bit to set up a configuration (here: PWM DACs to SERVO MODE)

```
CONFIG.INIT
CONFIG.GET MyByte
MyByte = MyByte OR 00000001b
CONFIG.PUT MyByte
CONFIG.OFF
```

I²C-BUS OBJECT

The IIC Object supports all basic functions to design I²C-Bus driver programs in an very easy manner. The I²C-Bus is connected to PORT 9 (SDA) und PORT 10 (SCL). An I²C-BUS error flag can be accessed at the CONFIG 1 register. Even Word sized values are accepted, the bus will transfer a single byte (Lo-Byte in case of a word) only

Beside of the general INIT und OFF procedure the I²C-Bus Object supports some more I²C-Bus Object special Instructions required for Bus control.

IIC.INIT
Activate the Object to get access to its registers for read/write operation. Close other Objects prior to open.

Syntax: OBJECT.instruction

IIC.START
IIC.STOP
Send a START CONDITION prior to any bus access. Send a STOP CONDITION to terminate bus access

Syntax: OBJECT.instruction

IIC.GET
IIC.SEND
Access the I²C bus to read/write a binary value.

Syntax: OBJECT.instruction Variable

variable: Variable, value or constant which is moved to the Bus at write acces.
Variable the value is moved to at bus read access.
Valid for word and byte size.

```
IIC.INIT
IIC.START
IIC.SEND MyAddress
IIC.SEND MyByte
IIC.STOP
IIC.OFF
```

The Example shows writing the value *Mybyte* to a I²C Bus device at address *MyAddress*

IIC.OFF
Close the Object prior to open any other Object.

Syntax: OBJECT.instruction

IR OBJECT

The IR Object supports all basic functions for infrared communication (based on the RC5 format) in an very easy manner. The IR Receiver/Transmitter connected to PORT 2 (RX) und PORT 3 (TX).
The RC5 data format consists of a device address and a data byte reveived from (or transmitted to) a IR remote device. Even Word sized values are accepted, IR.SEND/GET will transfer 6 bytes for command and 6 bytes for address and Toggle.

```
 RC5 FORMAT:
    13-12-11-10-09-08-07-06-05-04-03-02-01-00   DATA BIT
     S  S  T a4 a3 a2 a1 a0 c5 c4 c3 c2 c1 c0   RC5 Format

S = Start Bit (auto set/remove by the IR OBject)
T = Toggle
a = Address
c = Command

The IR MODULE expects a receiver connected at port 2 and the
transmitter connected to port 3
```

IR.INIT
Activate the Object to get access to its registers for read/write operation. Close other Objects prior to open.

Syntax: OBJECT.instruction

```
IR.INIT
IR.SEND MyAddress , MyByte
IR.OFF
```

IR.GET
IR.SEND
Access the I²C bus to read/write a binary value.

Syntax: OBJECT.instruction Address,Data

Address: Variable or constant which is send as address at write access (SEND)
Variable, the received address is moved to at read access. Valid for word and byte size.

Data: Variable or constant which is send as data at write access (SEND)
Variable, the received data is moved to at read access. Valid for word and byte size.

```
IR.INIT
IR.GET MyAddress , MyByte
IR.OFF
```

Both values (address and data) will be read as 255 if no valid data frame was received i.e if the receive buffer is empty The receive buffer is set to 255 (address and data) if it has been read.

IR.OFF
Close the Object prior to open any other Object.

Syntax: OBJECT.instruction

RF OBJECT

The RF Object supports all basic functions for RF communication (based on the HT12 format) in an very easy manner. The RF Receiver/Transmitter connected to PORT 2 (RX) und PORT 3 (TX).
The HT12 data format consists of a device address and a data byte reveived from (or transmitted to) a FT12 remote device. Even Word sized values are accepted, RF.SEND/GET will transfer 4 bytes for command and 8 bytes for address.

```
    FORMAT:
          11-10-09-08-07-06-05-04-03-02-01-00   DATA BIT
          c3 c2 c1 c0 a7 a6 a5 a4 a3 a2 a1 a0   HT12 Format

a = Address
c = Command

The RF MODULE expects a receiver connected at port 2 and the
transmitter connected to port 3
```

RF.INIT
Activate the Object to get access to its registers for read/write operation. Close other Objects prior to open.

Syntax: OBJECT.instruction

```
RF.INIT
RF.SEND MyAddress , MyByte
RF.OFF
```

RF.GET
RF.SEND
Access the I²C bus to read/write a binary value.

Syntax: OBJECT.instruction Address,Data

Address: Variable or constant which is send as address at write access (SEND)
Variable, the received address is moved to at read access. Valid for word and byte size.

Data: Variable or constant which is send as data at write access (SEND)
Variable, the received data is moved to at read access. Valid for word and byte size.

```
RF.INIT
RF.GET MyAddress , MyByte
RF.OFF
```

Both values (address and data) will be read as 255 if no valid data frame was received i.e if the receive buffer is empty. The receive buffer is set to 255 (address and data) if it has been read.

The Data Buffer may contain random Values if the receiver put out noise data (i.e. is working with open Squelch / noise suppression)

RF.OFF
Close the Object prior to open any other Object.

Syntax: OBJECT.instruction

LCD OBJECT

One of the mostly used features is the direct print to LCD function.

The LCD must be initialised with LCD.INIT once prior tu use. As this is a true initialisation of the LCD onboard controller it takes app. 20ms. Once the LCD is ititialised, LCD INIT switchonly is recommended I Due to technical reasons you must not use the "#" Character when writing to the LCD.

LCD.INIT
The LCD must be initialised with LCD.INIT once prior tu use. As this is a true initialisation of the LCD onboard controller it takes app. 20ms. The LCD is cleared and Cursor i set to Line1 Position 1

LCD.INIT switchonly
Once the LCD is ititialised, LCD INIT switchonly is recommended, because it only reopens the Object LCD.

Syntax: OBJECT.instruction

LCD.CLEAR
Clears LCD and sets cursor to line 1 position 1. Takes up to 2ms execution time

Syntax: OBJECT.instruction
LCD.POS
Set cursor to line and position

Syntax: OBJECT.instruction Line,Column

Line: fixed values 1 and 2 are valid, only
Column: fixed values 1 to 16 are valid, only

LCD.SR
LCD.SL
Rotate display content Right/Left one column.

PRINT
The PRINT instruction transmits strings,variables or constants of to the LCD. For transmission of more arguments with a single "Print" instruction you may use the concatenation operator &
Printing the character # is illegal because this character is internal used to precede a Control Sequence

Syntax: Print Argument [& Argument [& ...]] [;]

LCD.PRINT "WERT :" & Myword & " V"

LCD.OFF
Close the Object prior to open any other Object.

Syntax: OBJECT.instruction

START OPTION REGISTER

Both Units M 2.0 / M ADV feature a Start Option. Here a Hardware Option can be set at a non volatile register. This may not be confused with the CONFIG REGISTER what sets an option at runtime and keeps this options only until next reset. One the Start Option is activated it is maintained (even after reset) until the corresponding bit in the register has been cleared.

The bitset is done with a special BASIC++ Instruction that is not mentioned in this manual because it is not used in general.

```
ADDTOKEN   39   This Instruction will cause to write the next value to the Start Option Reg.
ADDTOKEN   3    This value 0000 0011 will be written to the Start Option and setting
                bit 0 and 1
```

A Program to set the options will only contain just these two lines, and has to run only one time.
For your conveniance the programs to set and reset options are supplied at the samples in the folder START OPTION

M ADV

The Start Option Register has following functions, switched to ON if the corresponding bit is set.

```
Bit 0       EEPROM BOOT OPTION
Bit 1       AUTOSTART AFTER PROGRAMM DOWNLOAD
```

If you want to activate both options, both bits ave to be set accordingly. The bitset is done with a special BASIC++ Instruction that is not mentioned in this manual because it is not used in general.

```
ADDTOKEN   39   This Instruction will cause to write the next value to the Start Option Reg.
ADDTOKEN   3    This value 0000 0011 will be written to the Start Option and setting
                bit 0 and 1
```

A Program to set the options will only contain just these two lines, and has to run only one time.
For your conveniance the programs to set and reset options are supplied at the samples in the folder START OPTION

M 2.0

The AUTO START Option is not available at this unit.
To activate EEPROM BOOT OPTION the START OPTION REGISTER has to contain the value 1
To deactivate EEPROM BOOT OPTION the START OPTION REGISTER has to contain the value 255

OPTION AUTOSTART

If the corresponding bit is set, it will cause an immediate Program Start after program download.
The Unit M 2.0 features no Auto Start Option.

EEPROM BOOT OPTION

The Operating System Versions above 2.05 offer the option to load a user program from a EEPROM e.g. in form of a „Chip-Card“. This is very comfortable if the user program has to be updated in the field without a PC available or if qualified personnel for a manual program download is not present. The EEPROM Boot Option is set off als default but can be switched on. The procedure is similar to a usual downlaod. A RESET causes the Unit to enter the Download Mode and the Operarting System tries to identify a EEPROM containing a valid program file. If no EEPROM is connected or the loaded file is not valid, the usual procedure (download from serial interface) is applicable. If a EEPROM containing a valid program file is connected then the program is copied into the FLASH Memory.
A program file on EEPROM is defined to have the first byte in memory loaded with the value $55. The memory beyond the program bytes can be used as data memory. The boot option is set on/off with a

dedicated small BASIC program. The boot file located on EEPROM is created with a special program running on a Control Unit.

To make a Boot File on a chipcard, load the Program MAKE_BOOTFILE.BAS (also in the folder START OPTION) into the unit and run the program.

Insert the memory card as requested by the message on the LCD.
When the card is found by the program u are requested to start the download as usual. The running program on your Control Unit simulates a regular download to flash but actually writes the program to the chipcard.

While downloading the program size and progress of written bytes is shown at the LCD.

The download is finished when the corresponding message is shown at the LCD. The Chipcard is ready to be used as boot device. To aktivate/deactivate the Boot Option run the programs run the dedicated programs, located in the folder START OPTION

Please note that the Unit M 2.0 and M ADV need different files to deactivate the boot option.

FLAOTING POINT MODULE (FPM)

The Floating Point Module is only available for the Unit M ADV

INTRODUCTION

The Control Unit M ADV is based on the Operating System of a standard Unit M 2.0, but has an additional 32 bit Floating Point SoftwareModule and an extended program memory (22k). The memory for variables has also been extended, but with some small restrictions. Unrestricted usable are 140 bytes, up to 240 bytes are usable if the File Function PRINT# is not used, or is used that way, (e.g for temporary variables) that it does not matter if variables content in the range from 140 to 240 will be changed when INPUT# is used.

Floating Point Extension Library
The basic floating point operations are done on system level i.e. they are implemented as machine code executables. Beside this, other functions e.g LN (natural logarithm) are executed on BASIC level, because here it is useful to trade between accuracy and computing speed. Similar can be said for tools. e.g a program module that allows to enter floating point values, using the Keyboards. Here also the user has the chance to change this modules to his specific requirements.

Limitations:
Implementing a Floating Point Module in a small machine like the Control Units does not affect the accuracy of course, but has some other disadvantages. This means that the stack of the Control Unit is sufficient to do all calculations, but requires some careful considerations of the user, programming his applications.
Same can be said for type convesion. Often this conversions are done in background, and the user may directly move a integer value to a floating point variable. In such a small machine there is not enough memory for the operating system do do things like this. Here the user has to take care himself to make proper convertations when mixing variable Types.

Mismatching variable types will often cause wrong results in calculations and may crash the program because of stack errors.

DETAILS OF THE DATA TYPE FLOAT

Internal organisation
While the ControlUnits usually have just BYTE and WORD Types, even for non professionals easy to handle, now there is the new Data Type Float. It consists on one byte exponent and 3 byte mantissa with sign. Therefore a floating point value occupies 4 bytes (32 bit).

Value size
The flaoting point Module is able to compute values in the range of +-1 × 10±38.

Accuracy
You will find that even simple values can not be expressed 100% accurate with a 32bit Floating Point System. The value 1234 will be shown as 1233.999, but this is not a computing error. It is based on the limited accuracy of *every* 32 bit Floating Point System.

INTERNAL HANDLING OF FP OPERATIONS

For simple calculations it is not required to know this details. But very complex computations may lead to stack overflows if some basics are not considered.

The Control Unit Stack
The stack is a memory used for temporary storage of values. This can be values needed for calculations or addresses, needed for a return from a subroutine.

For a brief information of stack usage see this example:

MyFloat=FloatVar1*Floatvar2

This expression is compiled to a token code that causes the operating system to do the following:

```
Put content of  FloatVar1 to Stack
Put content of Floatvar2 to Stack
Pull last 2 stack entries from stack and push them  into FP Accumulators (FPACC)
Multiply the FPACCs content
Push the result to stack
Pull the last stack entry from stack and move it to  MyFloat
```

If the term is more complex, many values will be pushed to stack before any calculation is done and they are removed from stack.

Example:
MyFloat=Floatvar3-FloatVar1*(Floatvar2+Floatvar4)

```
Content of FloatVar3 to Stack
Content of Floatvar1 to Stack
Content of FloatVar2 to Stack
Content of Floatvar4 to Stack
Pull last 2 stack entries from stack, load them in to FPACCs
Add FPACCs
Push result to Stack
Pull last 2 stack entries from stack, load them in to FPACCs
Multiply FPACCs
Push result to Stack
Pull last 2 stack entries from stack, load them in to FPACCs
Subtract FPACCs
Push result to Stack
Pull last stack entry and move to  MyFloat
```

You can see that even this simple term pushes 4 float values to stack, before calculation is done and values are removed from stack. Here 16 Bytes Stack are used for this simple term. And remember that a subroutine call also adds two bytes return address to stack.

To avoid annoing wrong results or even crashes, split your computation into small terms, what makes the code better readable too.

Stack overflow will cause wrong results and even a crash or other unpredictable behaviour of the Control Unit.

FLOATING POINT INPUT/OUTPUT

It is important to instruct the compiler that float values will have to be compiled. This is done with the compiler instruction "OPTION FLOAT" , located the beginning of your program file. Remember that a float takes 4 bytes, and be carefull on use of too much variables of this type. The internal assigning of variables to the memory is done by the compiler automatically. The compiler also reserves further 16 bytes for internal use for the FP System.

```
option float
define FV1 as float
define FV2 as float
```

ASSIGNMENTS

Now u can start to write a small program. This usually is done with assignment of variables used in the program. The example shows the the different ways to do this.
The FPM is capable to handle 7 digits, more digits are ignored. The exponent can range +-38

```
FV1=3
FV1=2.3
FV1=0.023
FV1=2.12345678
FV1=exp(1.602,-19)    corresponds to  1.602E-19
```

INPUT

The FPM has an INPUT function as it is usual for WORDs and BYTEs

INPUT MyFloat ;
Terminal input of a FP Value and move to MyFloat
Because a FP value can be expressed in many different ways there are some regulations to consider:

1) The leading ZERO in front of the decimal point must be written 0.001 (not: .001)
2) The exponent must have two digits: 0.234E03 (not: E3)
3) After the decimal point one digit has to follow as a minimum 1234.0 (not:1234.)

Beside this the input format is widely flexible as the examples show:

```
0.00000000000000000000000001
1.123456789123456789
12345678912345678912345678
12345678912345678912345678.12345678912345678
12345678912345.7891234E09
-12345678912345.123456789123456789E-12
12.3E12
0.0000000012345E13
```

An Input is terminated if the terminal sends CR or any other non numeric character. If your input device sends CR LF on termination u have to remove the LF (use getMyByte). Beside this all other considerations concerning the use of the serial interface are valid. All inputs may appear different when they will be shown on LCD or printed. This is caused by internal formatting, but of course does not change the value itself.

OUTPUT

Output ot terminal or LCD is controlled by the Object that is active. All examples for the Unit M ADV write to LCD

Outputformat:
Values larger than 1:
For output in decimal there are 7 digits available which are distributed to be located before and after the decimal point.
But always at least one digit remains after the decimal point.
If this is not possible (e.g a value with more than 7 digits before decimal point) the scientific notation is used.
Is a value positive, a blank is written first and a minus otherwise.

Values 0 bis 1:
Very small values are shown decimal as long as there is only a single zero after the decimal point.
Otherwise here the scientific notation is used also.

```
12345.123456        shown as        12345.12
123456.1234567      shown as        123456.1
1234567.1234567     shown as        1.23456E06
-1234.1234567       shown as        -1234.123
1.234566E03         shown as        1234.123
0.0123456           shown as        0.012345
0.001234567         shown as        1.234567E-03
```

FPPRINT
Provides a formatted output to LCD or terminal
Because a FP value may have many digits it is useful to limit the number of digits after decimal point. The decimal point counts as one digits .
FPRINT always means an output to LCD or terminal (depending on the active object) and is not written as LCD.FPRINT if output is done to LCD

```
FPPRINT (Term, Digits)
```

Examples for terms in FPPRINT:

```
FPPRINT(MyFloat*FLOAT(MyWord),5)
FPPRINT(MyFloat*MyFloat,5)
```

Note:
FPPRINT ignors the operator &
FPPRINT (MyVar,3) & "VOLT" is therefore not possible, but easyly can be achieved by adding:
LCD.PRINT "VOLT"

PRINT
Provides a standard unformated output to LCD or terminal

A PRINT instruction will cause an output value with n digits after the decimal point (corresponding to the internal formatting) and is not different to an output of a WORD or BYTE variable. For an output to LCD here therefore it must be written:

```
LCD.PRINT MyVar & "VOLT"
```

Note:
*It is not possible to print a term of float variables like PRINT (MyVar1*MyVar2).*
It alway has to be a float variable like this print example: PRINT(MyVar)

Examples:

```
option float
define FV1 as float
define FV2 as float
LCD.INIT
'------------------------------------------------------------------------ ------------------------
FV1=exp(1.23456789,4)  '1.23456789E04
'------------------------------------------------------------------------ ------------------------
FV2=12345.6789
'----------------------- EXAMPLE FOR OUTPUT ----------------------------------------
LCD.POS 1,1
LCD.PRINT FV1 & " VOLT     "
LCD.POS 2,1
LCD.PRINT FV2 & " VOLT     "
```

Evaluate the sample program FP_INPUT_OUTPUT. Change the format parameters to get familiar with.

TYPE CONVERSION FLOAT<-> INTEGER WORD/BYTE

Depending on the application it may be usefull to convert data types. In all cases the user has to ensure that the values of the variables match the type they should be converted to. A value 70000 e.g can not be converted to a WORD or BYTE type variable.

FLOAT

Conversion of a term or variable from BYTE/WORD Type to FLOAT

```
MyFloat = FLOAT(MyWord)
MyFloat = MyFloat*FLOAT(MyWord)
MyFloat = MyFloat*FLOAT(MyWord*MyWord)
MyFloat = MyFloat*FLOAT(MyWord*MyByte)
```

Mixed Data Types as below shown are not supported:

*MyFloat = MyFloat*FLOAT(MyWord)*MyWord*
Will lead to wrong results

MyFloat = MyWord
Will lead to stackerrrors and related trouble

INT

Conversion of a F:OAT varaiable to INTEGER/BYTE Type

Here it is not possible to convert a Term. Conversion is restricted to variables only.

Conversion of FLOAT valiables to BYTE/WORD size

```
MyWord = INT(MyFloat)
MyWord = 3*INT(MyFloat)/MyWord
MyByte = 3*INT(MyFloat)/MyWord
```

This is a samle of a invalid conversion:
MyWord=INT(MyFloat/MyFloat)

Traps:
It is eminent always to consider the different data type ranges when converting. See the samples for details.

MyFloat1=50000
MyFloat2=30000
MyWord=1000

Case1:
INT_RESULT=INT(MyFloat)
The value may overflow a WORD on conversion.

Case2:
INT_RESULT=INT(MyFloat)/MyWord
Even though the result (5000) is not too large for a WORD, the value will overflow the WORD before division

Case 3:
INT_RESULT=10*INT(MyFloat2)/MyWord
Here an overflow will occur at the multiplication 10*INT(30000)

INT_RESULT=INT(MyFloat2)/MyWord*10
This will show the right result

```
option float
define LIGHT as port[16]
define WV1 as word
define WV2 as word
define FV1 as float
define FRESULT as float
define IRESULT as word
LIGHT=off
LCD.INIT


FV1=exp(1.23456789,4)  '1.23456789E03
WV1=1000
'*************************************************************************************************************
'***            FLOAT CONVERSION WITHIN A TERM                                                              ***
'*************************************************************************************************************
'------- TERM WITH FP-VARIABLE AND CONSTANT -------------------------
FRESULT=FV1*1000
LCD.POS 1,1
LCD.PRINT FRESULT & " VOLT      "
'------- TERM WITH FP-VARIABLE AND WORD-Variable ----------------------
WV1=1000
FRESULT=FV1*FLOAT(WV1*10)                          'Conversion WORD-> FLOAT !!
LCD.POS 2,1
LCD.PRINT FRESULT & " VOLT      "
PAUSE 100
'*************************************************************************************************************
'***                INTEGER CONVERSION  OF A TERMS                                                          ***
'*************************************************************************************************************
' Is not allowed and nust be done my conversion of variables

'------- TERM WITH FP-VARIABLE AND CONSTANTE -------------------------
FV1=5678.234
IRESULT=INT(FV1)/1000
LCD.POS 1,1
LCD.PRINT IRESULT & " VOLT         "
'------- TERM WITH FP-VARIABLE AND WORD-Variable ----------------------
WV1=1000
IRESULT=3*INT(FV1)/WV1                             'Conversion FLOAT-> WORD
LCD.POS 2,1
LCD.PRINT IRESULT & " VOLT        "
PAUSE 100
```

Review the sample program 2_FP_CONVERSION

LOOPS WITH FLOAT VARIABLES

FOR TO NEXT loops will not work together with FP variables. This does not matter because it is possible to use DO LOOP UNTIL and here it is possible to create loops working with all valid floating point values.

```
DO
. . . . .
. . . . .

LOOP UNTIL (COUNTER>ENDVALUE)
```

The end condition may not consist of terms or constants
LOOP UNTIL (COUNTER>123456)   is not allowed

Example:

```
'************************************************************************
'***         LOOP WITH MIT FLOAT VARIABLES                ***
'************************************************************************
COUNTER=123.4567
ENDVALUE=999.567

'---- FOR COUNTER=123.4567 TO 999.567 STEP 0.123 ----

DO
COUNTER=COUNTER+0.123
LCD.POS 1,1
LCD.PRINT COUNTER & "         "
LOOP UNTIL COUNTER>ENDVALUE
```

FUNCTIONS WITH PARAMETERS

The function calls are equally done with word or bytes. Parameters may also be mixed types.

```
FUNCTION MyFunction(FVAL1 as FLOAT,FVAL2 as FLOAT)
. . . . . .
. . . . . .

END FUNCTION
```

Because the memory usage is very intesive, when using float variables it is recommended to use referenced float values as local variables. This means different variables as here in the example INPUT and WERT share one memory location (FVALUE in this example).

```
define FVALUE as FLOAT

FUNCTION ABC(INPUT ref FVALUE)
RESULT=RESULT*INPUT
END FUNCTION

FUNCTION XYZ(WERT ref FVALUE)
RESULT=RESULT*WERT
END FUNCTION
```

FUNCTIONS WITH PARAMETER RETURN

A function also can return a float value if the return value is assigned to a help variable (in this example MyFunction) that is named equal to the function itself (here MyFunction) The help variable must not be declared seperately.

```
FUNCTION MyFunction(FVAL1 as FLOAT,FVAL2 as FLOAT)
. . . . . .
MyFunction=FVAL1*FVAL2
. . . . . .
END FUNCTION


'------------ FUNCTION AUFRUF MIT RÜCKGABE -----------------------
FLOATVAR=MyFunction(10.9, 33.0)
```

Review the example 4_FUNCTION_1 and 2 for details.

MATH OPERATIONS

The overview shows the operations that are integrated to the operating system, and therefore executed with maximum speed. Other operations are available as extension, but are executed as BASIC Code.
Because the BASIC code is just used to repeatedly call system located float operations, the loss in speed is negligible, and the user has the chance to change certain algorithms to better accuray if it is needed.
The functions are provided at the FLOTMATH.BLIB and described her in this document it the capter FLOATING POINTMATH BASIC++ LIBRARY

Overview:

```
MULTIPLY
DIVIDE
ADD
SUBTRACT
SQRT
SIN
COS
ABS
<, >, =, <=, >=
```

Please review the example program 5_STANDARD_FLOAT_MATH.BAS

MULTIPLY, DIVIDE, ADD, SUBTRACT

For a multiply demo a conversion of a analog value (given by an ADC) to the true physical value is well suited. The result is shown at the LCD, limited to 4 digits after decimal point. The secons sample is a compare that show how much more lines a program has, doing the equivalent conversion with integer calculations.

```
DO
VOLTS=FLOAT(ADC8)*0.0196
LCD.POS 1,1
LCD.PRINT VOLTS & "V      "
LOOP
```

```
DO
MILLIVOLT=98*ADC8/5
VOLT=MILLIVOLT/1000
NACHKOMMA=MILLIVOLT MOD 1000
LCD.POS 1,1
LCD.PRINT "ADC8: "& VOLT & "."
if NACHKOMMA<100 then LCD.PRINT "0"
LCD.PRINT NACHKOMMA & " V     "
LOOP
```

Samples of some complexTerms

```
MyFloat=MyFloat/FLOAT(ADC8*MyByte)*0.0196+MyFloat*MyFloat
MyFloat= FLOAT(MyWord-MyByte*MyWord)/(MyFloat+MyFloat/5)
```

SIN, COS

Accuracy
The Sinus is calculated with a Taylor Function and is considered to be a very good approximation. The calculated values for angles 0 to 90 degrees have more than 5 digit accuracy. But beyond 90 deg. the accuracy drops to 3 digits at 130° and 2 digits at 179°. The accuracy can be improved if the term **x^11/11!** and further terms are added.

Domain
The calculation is limited to angles from -180° to +180°. Angles beyond this range have to be mapped to this range. This is also valid vor COS because it is calculated by the equation COS(x)=1-SIN(x).

SIN can not calculate a term. A constant angle or a variable containing an angle is required therefore **BPPDEGREES** is the input in degrees, **BPPRADIANS** is input as radians (-3.14.....+3.14).

```
MyFloat=SIN(ANGLE,bppdegrees)
MyFloat=SIN(30.33,bppdegrees)
```

```
MyFloat=COS(ANGLE,bppdegrees)
MyFloat=COS(30.33,bppdegrees)
```

SQRT

accuracy
The Sqare Root calculation is done also by an approximation with an accuracy of better than 5 digits. SQRT can not calculate a term. A single constant or variable is required for calculation.

```
MyFloat=SQRT(VALUE)
MyFloat=SQRT(12.345)
```

ABS

ABS calculates the absolute value of a terms or a variable

```
MyFloat=ABS(FLOAT(ADC8*MyByte)*0.0196+MyFloat*MyFloat)
MyFloat= ABS(FLOAT(MyWord-MyByte*MyWord))/ABS(MyFloat+MyFloat/5)
```

COMPARES

A compare is always related to two variables. Compares of constants or terms are not allowed.
Valid compare operators are: **<, >, =, <=, >=**

```
IF MyFloat1>MyFloat2 THEN.....
IF FLOAT(MyWord) > MyFloat1 THEN.....
LOOP UNTIL  MyFloat1>MyFloat2
LOOP UNTIL FLOAT(MyWord) > MyFloat1
IF NOT(MyFloat1=MyFloat2) THEN.....
```

The operator <> is missing. An equivalent compare is done with:
IF NOT(MyFloat1=MyFloat2) THEN.....

FLOATING POINT ERRORS

Though calculations have to be done in a way that no illegal operations are processed (such as division by zero) it may nevertheless happen that e.g a sensor malfunction leads to that. Because this can lead to plausible results by all means, even the results are totally wrong, you can check your results for this cases of errors.

ON ERROR GOTO MyErrorHandler

Adds an error request to the compiled program code and jumps to the error handler. The variable **ERR.NUMBER** contains the error code. ERR.NUMBER is cleared before any floating point operation.

```
0  No error
1  Overflow error
2  Underflow error
3  Division by Zero error
4  Square root of negative number
5  Number too large/small to convert
to integer
```

This example at the right demonstrates the usage. Continued multiplication leads to an overflow, the errorhandler is called and the error code is shown.

```
FUNCTION FPE()
ON ERROR GOTO ER
#x
RESULT=1
DO
RESULT=RESULT*100
LCD.POS 1,1
LCD.PRINT RESULT & "            "
pause 20
LOOP
'-------- DISPLAY AN ERROR ----------
#ER
LCD.POS 2,1
LCD.PRINT "ERROR:" & ERR.NUMBER
BEEP 5,5,50
LCD.POS 2,1
LCD.PRINT "                "
goto x
END FUNCTION
```

THE FLOATING POINT MATH BASIC++ LIBRARY

Not all floating point operations are done on operating system level. Some special operations are available as extension, and are executed as BASIC Code. The functions are provided at the **FLOTMATH.BLIB** and described here. The FLOATMATH.BLIB contains the declaration of all used variables:

```
define BVALUE1 as byte
define BVALUE2 as word
define FVALUE1 as float
define FVALUE2 as float
define FVALUE3 as float
define FVALUE4 as float
define FVALUE5 as float
define FVALUE6 as float
define RESULT ref FVALUE2
```

The variables must not be declared a second time, but can be referenced.
The Library contains the following functions:

```
LN(x)
LOG(x)
TAN(x)  (x in degrees)
EXPO(x,y)
POWER(x,y)
E(x)
Nth_ROOT(x,y)
ARCSIN(x)
ARCCOS(x)
```

All Functions are called with one or more parameter. The result is contained in the variable RESULT, always.
e.g. **ARCCOS(0.5)**

Moreover the functions can be called to return a value e.g. MyFloat=LN(MyFloat)
assignments like this: **MyFloat=LN(MyFloat)**
It is also legal to form complex terms in this way: **FV1=LOG(POWER(3,nth_ROOT(27,3)))**

Please review 6_EXTENDED_FLOAT_MATH.BAS for examples

LN(MyFloat)

Is calculated with a precision of better than 4 digits. To gain precision the corresponding function can be modified by changing the loopcounter from 9 to 13 or any other odd number.

FOR BVALUE1=3 TO 9 STEP 2

```
LN(MyFloat)
LN(12.345)
LN((FLOAT(MyWord))
The variable RESULT contains calculation result
```

LOG(MyFloat)

LOG calculations are based on the LN function and is equal in pecision. Modify LN as described to gain precision for LOG.

```
LOG(MyFloat)
LOG(12.345)
LOG((FLOAT(MyWord))
The variable RESULT contains calculation result
```

EXPO(MyFloatX,MyByteY)

Calculates x power y based on continued multiplication and is full 32 bit accurate. The exponent is signed, but may not exceed byte size.

```
EXPO(MyFloat, MyWord)
EXPO(12.345,3)
EXP=(MyFloat,3)
The variable RESULT contains calculation result
```

EXPO is much faster than the POWER function, but is only working with byte size exponents.

POWER(MyFloatX,MyFLOATY)

Calculates x power y

```
POWER(MyFloatx, MyFloaty)
POWER(12.345,3.33)
POWER(MyFloat,3.45)
The variable RESULT contains calculation result
```

The calculation is based on x^y = e ^ (y * LN x) and therefore has limited accuraca. To improve accuracy change the number of iterations in the library of function E

BVALUE1=30 'ITERATIONS

E(MyFloatX)

calculates e power x

```
E(MyFloatx)
E(12.345)
The variable RESULT contains calculation result
```

Is calculated with a precision of better than 4 digits as long as the exponent is within -5 to 15. To gain precision the corresponding function can be modified by changing the number of iterations

Nth_ROOT(MyFloatX,MYByteY)

Calculates Yth root of X

```
Nth_ROOT(MyFloat, Mybyte)
Nth_ROOT(12.345,3)
Nth_ROOT(MyFloat,3)
The variable RESULT contains calculation result
```

FAK(MyByte)

Calculates faculty of a value

```
FAK(MyByte)
FAK(12)
FAK((INT(MyFLOAT))
The variable RESULT contains calculation result
```

TAN(x) (x=degrees)

TAN is calculated with a SIN function and has an absolute error of 0.5 at 89°. At lower angles the accuracy increases rapidly. At 80° the calculation is accurate up to 3 digits.

```
TAN(MyFloat)
TAN(12.345)
The variable RESULT contains calculation result
```

ARCSIN(MyFloatX) ARCCOS(MyFloatX)

This functions return degrees and the result becomes inaccurate exceeding 0.9.
While ARCSIN(0.9) is accurate to several digits after the decimal point, ARCSIN (0.9999) has errors in the order of 0.1 degrees.

```
ARCSIN(MyFloat)
ARCSIN(0.5)
Das Ergebnis steht dann in der Variablen RESULT
```

ARCTAN(MyFloatX) ARCCOT(MyFloatX)

This functions are calculated by an approximation and return degrees.
For |x|<1 the accuracy is several digits after decimal point.
For |x|>1 the accuracy is better than 1 digit after decimal point.

```
ARCTAN(MyFloat)
ARCCOT(0.5)
Das Ergebnis steht dann in der Variablen RESULT
```

THE FLOATING POINT TOOLS LIBRARY

This library contains tools that are often needed and will make your programming more enjoyable

GET_FPVALUE() **Enter a FP value with keyboard**
The call of this function lets u type a floating point by keyboard value e.g. to set up some parameters or similar.
The input value then is available at the variable RESULT1 for further processing.
The function is dedicated to the keyboard provided with the Application Board or any other based on this circuit. The keys 0 to 9 have to be pressed to enter this numeric values. F2 will create the minus sign, F1 the decimal point. If an decimal point is already entered, F1 will create the character E preceding the exponent value. If a value is entered in scientific notation, the mantissa must have a decimal point therefore.

See the example here:

-123E-02

INPUT: <F2> <1> <2> <3> <F1> <0> <F1> <F2> <0> <1> <E> enters the value **-123.0E-02**

The input format is widely flexible, but some rules have to be considered.
1) The leading ZERO in front of the decimal point must be written 0.001 (not: .001)
2) The exponent must have two digits: 0.234E03 (not: E3)
3) After the decimal point one digit has to follow as a minimum 1234.0 (not:1234.)
4)The total entered number of characters (inclusive E , minus and decimal point) must be 15 characters maximum.

Examples of valid inputs

0.0000000000001
1.1234567891234
123456789123456
123.12345678912
3456.7891234E09
-12345678.1E-12
12.3E12
-1234.56789E-01

The input is terminated when pressing the Button "E" (do not confuse with pressind F1 for character E). Pressing the key "C" will clear all digits. A single digit clear is not possible.

Review the example 8_GET_FPVALUE.BAS

Caution:
This function uses the RS232 serial interface buffer. To use this Function it must be ensured that no characters are received while this function is running.

PUTFLOAT(MyFloat) **Moving Float Values to the Flash Memory (equals to PRINT#)**
Use this function to save float values to non volatile FALSH memory. A flaoting point value will occupy 4 bytes of memory space.

GETFLOAT() **Read Float values from Flash memory(equals to INPUT#)**
The variable RESULT1will contain the value red from Flash after call.

Review the sample 7_FP_DATASAVE.BAS

BASIC PROGRAMMABLE CONTROL UNITS

http://www.Spiketronics.com

D

GB