STM8 BASIC

Embedded

for the

STM8S103F3 Microcontroller

User's Manual

Copyright © 2019, 2020 by Roger DiPaolo
All Rights Reserved

1.0 Welcome

Thank you for acquiring your new STM8S103F3 Minimum System Board preloaded with the STM8S103F3 version of **STM8 BASIC Embedded**.

STM8 BASIC Embedded is a version of the BASIC language designed specifically for small 8-bit microcontrollers with very limited memory – meaning it will run on very inexpensive microcontrollers. It was inspired by early versions of BASIC (Beginner's All purpose Symbolic Instruction Code) such as the original version, Dartmouth BASIC and BASIC versions such as used on vintage 8-bit Personal Computers – in this sense it is also honoring the early days of programming in BASIC when 8-bit computers began a revolution in the computing world. If you've ever programmed using BASIC on a 1980's or earlier personal computer this style of BASIC will seem very familiar to you – and even if you haven't you will still find STM8 BASIC Embedded very easy to learn and use.

STM8 BASIC Embedded's goal is to allow you to program a microcontroller to do anything it supports, within limited memory constraints of course, controlling it at the register level using a very simple and easy to learn interpreted language.

STM8 BASIC Embedded provides features not known to most varieties of the BASIC language, features specifically designed to allow you do things required when register level programming a microcontroller such as hexadecimal math, hexadecimal user I/O, and bitwise logical operators common to languages such as C and C++ (although the syntax of course differs).

STM8 BASIC Embedded gives you maximum visibility and reach into the Microcontroller and it's memory. With the exception of the code memory for the actual BASIC environment and interpreter by using the PEEK and POKE, along with the very powerful "M-Command" function, you can access almost any memory location and all of the peripheral device registers with both read and write capabilities – if you really want to get tricky you can, if you are very very careful, even modify the BASIC code memory itself which holds your BASIC program that exists in non-volatile EEPROM. Yes, (partially) self-modifying code

is possible with this system – and of course messing things up if you don’t do it exactly right is very possible as well. ;)

For debugging your BASIC programs a debug tool is provided which both traces line execution and outputs the values of all variables at that point in time.

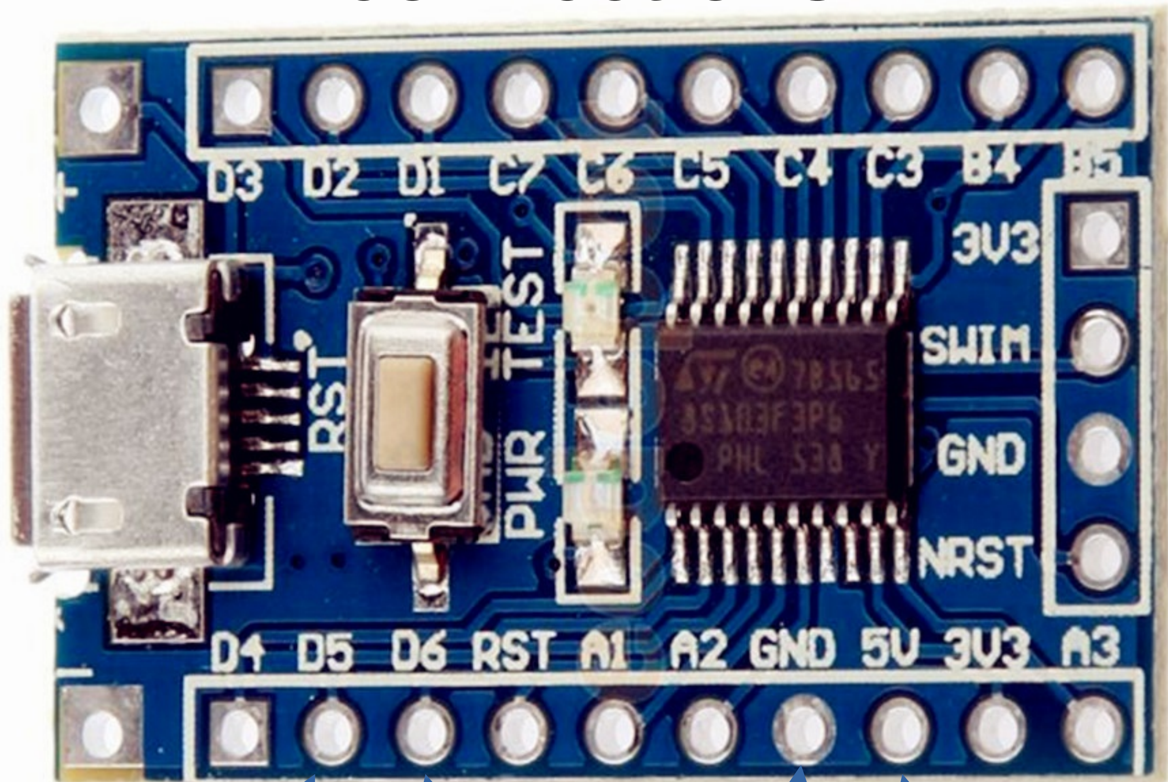
There are some things that STM8 BASIC Embedded wasn’t designed for that other varieties of BASIC do well. All of them would have been included if there was enough code space on the microcontroller to support them, but there isn’t so it was decided that what would be left out of this version of BASIC was anything not absolutely necessary for the use of the microcontroller’s peripheral devices. This BASIC is intended primarily for controlling hardware devices such as I/O (Input/Output) ports, A/D (Analog to Digital) converter inputs, and PWM (Pulse Width Modulated) outputs on microcontrollers so small that they don’t have the memory to support other versions of BASIC besides STM8 BASIC Embedded.

NOTE: It is strongly recommended to carefully review all of the example programs in this manual as a great deal of detailed information about the system can be gained from the listings and their descriptions.

2.0 Getting Started

You will first need to connect your board to power and a serial terminal. If you are using a standard Serial to USB converter cable then refer to the diagram below.

USB to Serial converter connections



The key connection points are:

D5 RS232 serial TX (transmit) out of your board (3.3 volt levels).

D6 RS232 serial RX (receive) into your board (3.3 volt levels).

GND Ground (from power supply, wire usually black)

5V 4.5 to 15 volts into the voltage regulator (wire usually red)

3V3 3.3 volts out from the voltage regulator (powers the micro & LEDs)

The on-board USB connector only provides power to the board, there is no actual USB connection. Power is usually provided through a Serial to USB cable as shown in the connection diagram above. Note that the power input is +4.5 to +15 VDC, whereas the microcontroller runs on +3.3VDC. There is a voltage regulator on the underside of the board that converts the +4.5 to +15 VDC to +3.3VDC to power the microcontroller. All Board I/O expects inputs to be 0 to +3.3VDC, and outputs from 0 to +3.3VDC. If you are interfacing to any +5VDC level inputs or outputs you will need to use 5V<=>3.3V level converters, or “level shifters” (easily and inexpensively available on the internet), or else you risk damaging your microcontroller and/or connected devices.

The 3V3, SWIM, GND, and NRST connection points on the opposite end of the board from the USB connector are exclusively for development and debugging of the processor’s native code (The BASIC interpreter and environment) and therefore are not used.

The “RST” connection point can be used to reset the microcontroller by pulling the line to ground and releasing it, doing exactly the same function as if the on board reset button had been pressed. All other connection points not in use can be programmed to be inputs or outputs of the microcontroller’s internal devices.

The RS-232 Serial interface consists of the TX (transmit) line out of the board at connection point D5, and RX (receive) at connection point D6. The connection diagram shows the wire colors for connecting the 4 wires of a standard serial to USB cable with wires in place of a standard DB-9 serial port connector.

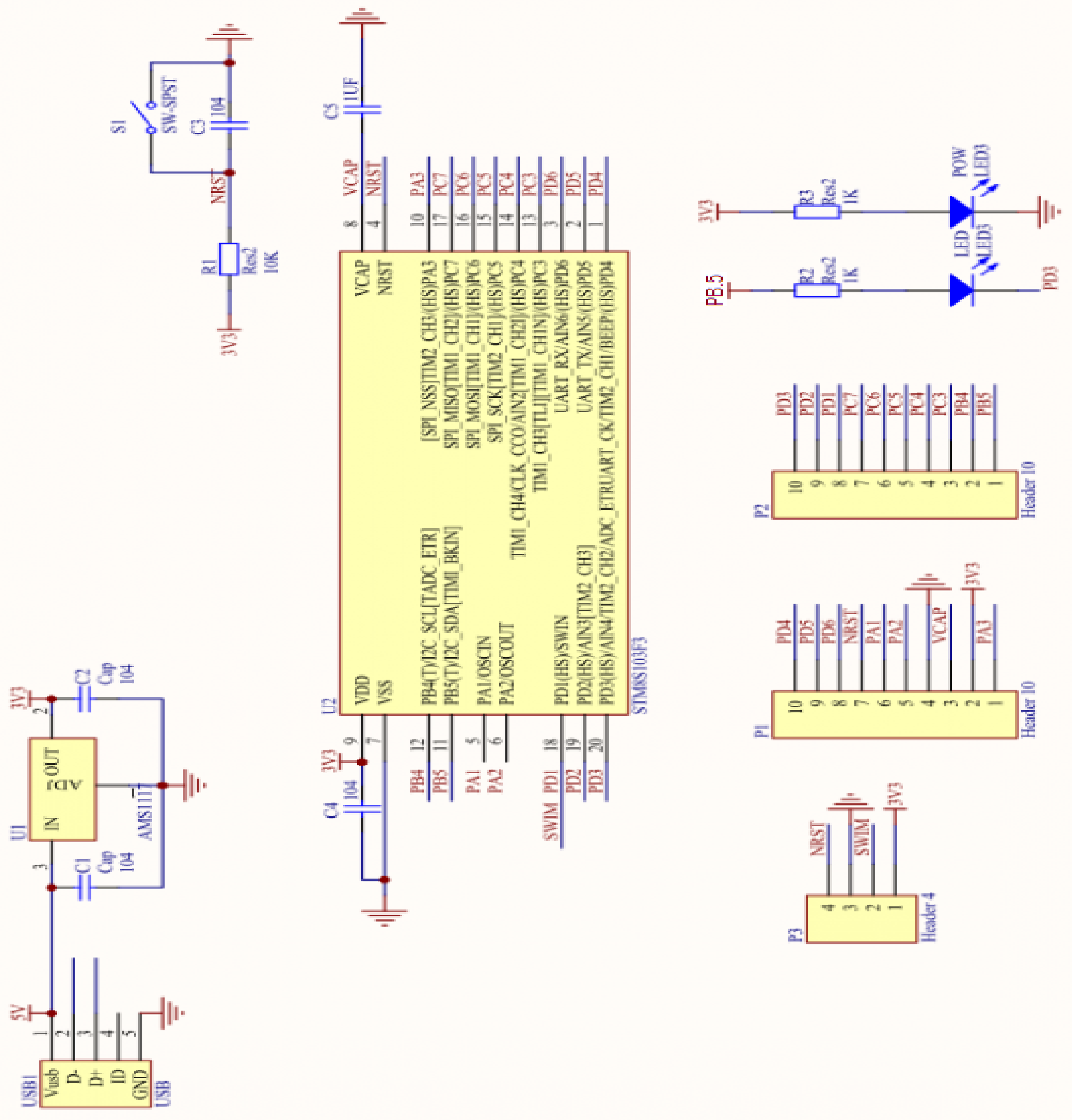
The serial interface works with the following parameters:

Data rate: 115200 Baud
Parity: None
Data bits: 8
Stop bits: 1

This board is intended to be used with a serial terminal as the user interface. If you are connecting to a Windows host PC, the “Tera Term” freeware terminal program is recommended for use, as it has been found to work better than the others tested for use with this board although most serial terminal programs can be used.

A direct serial connection, instead of going through a serial to USB cable, can be used as well as long as the TX (connection point D5) and RX (connection point D6) serial lines are level shifted into and out of the board between 3.3VDC on board and 5VDC externally to interface directly to PC serial ports.

Below is a schematic of your board:



IMMEDIATE COMMANDS FOR STM8 BASIC EMBEDDED

Command	Description
LIST	List the current program, with a pause every 10 lines. This is the only command that also works in Edit mode.
NEW	Clears all BASIC program memory.
RUN	Executes the program currently in BASIC program memory, if any exists.
EDIT	Unlike traditional BASIC, STM8 BASIC employs an "Edit mode" entered by this command. You will notice that in Edit mode the prompt will change from 'OK' to ':' NOTE: USE AN "ESCAPE" KEYPRESS TO EXIT EDIT MODE AND RETURN TO THE COMMAND ENVIRONMENT.
AR	Reports current autorun state, on or off
ARON	Enables automatic execution of current program on power-on or reset (not recommended until you have a fully stable program).
AROFF	Disables automatic execution of program on poweron or reset - DEFAULT MODE. This is a non-volatile setting.
CLR	Performs an ANSI Clear Screen on the terminal display.

MEM	Report BASIC program memory status. Shows current number of bytes free.
RST	Cause a hard reset of the CPU
DBGON	Turn on execution debug (DeBuGON). This debugging tool shows the line number and full text of each line just after the line executes, along with a display showing the current value of all variables at that moment.
DBGOFF	Turn off execution debug (DeBuGOFF).
<'ESC' Key>	Exit to the command environment from either an executing program or Edit mode.

STM8 BASIC EMBEDDED PROGRAM FUNCTIONS & STATEMENTS

CLR

Performs an ANSI Clear Screen on the terminal display.

END

When encountered END will halt program execution and return to the command environment. Not required if your program loops “forever” and never ends, or simply executes a linear sequence of statements with no “forever” looping. No line following the last valid execution line of a BASIC program will imply an “END” without actually needing one. Excellent for looping until a program exit condition is met, then branching to an “END” statement.

FOR var = expression TO expression

Loop between this line and the line containing “NEXT <var>” by first assigning your chosen variable to the initial value after which it will be automatically incremented by one each time through the loop until it reaches the terminal count specified in the “TO” expression. After terminal count is reached execution will continue at the line following the “NEXT <var>” line.

GOSUB (subroutine line number expression)

Go to the line number indicated by the expression (start of the subroutine), then

return to the line following this one upon encountering the first occurrence of a RETURN statement in the subroutine.

GOTO (destination line number expression)

Continue program execution at the destination line number indicated by the expression.

IF (comparison expression) THEN (“GOTO” expression)

If the comparison expression evaluates to TRUE, then continue program execution at the line number indicated by the ‘GOTO’ expression. Note that the explicit use of the keyword “GOTO” <u>is not part of the IF statement syntax</u>, but it is implied.

INPUT variable

Waits for the user to enter either a positive or negative integer value from -32767 to 32767 if the variable is A through F. Variables G through L, and Z, are unsigned 16-bit variables with a range of 0 to 65535 ($FFFF). Variables N through Y have a range of from 0 to 255. Hexadecimal input is also supported, the Hexadecimal number must be preceded by a ‘$’ character (such as: $FF for 255).

NOTE: Hitting the escape (ESC) key instead of entering a numeric value will skip that INPUT statement without altering the value of the specified variable.

LON (v2.3 or later only)

Illuminate the on-board user LED. Note that this is destructive to all Port B bits. For register safe LED usage see the example programs.

LOFF (v2.3 or later only)

Extinguish the on-board user LED. Note that this is destructive to all Port B bits. For register safe LED usage see the example programs.

M (4-byte hex address)(hex data bytes)

This can be considered “POKE on steroids”. The “M” command (for ‘M’emory insert) takes the first 4 hex digits following the character ‘M’ as the starting address to write the subsequent data bytes to. Each data byte is entered as 2 hex digits to be written to subsequent memory locations. Note that as in the example there is no space between the “M” and the first digit of the address.

Example:

```
10 M02001122334455667788
```

This writes the 8 bytes, $11, $22, $33, $44, $55, $66, $77, and $88 to 8 subsequent memory locations beginning at address $0200.

NOTE: 256 bytes of volatile User RAM is available from $0200 to $02FF, and is reserved exclusively for your use.

NEXT variable

(see FOR statement description)

var = NOT variable

NOT Performs a unary bitwise negation (inversion) on the variable (ex: $FF becomes $00, and $55 becomes $AA, etc.).

var = PEEK (address expression)

Assign to variable the 8-bit byte contents of the STM8 memory address indicated by expression. PEEK of a non-existent memory location will cause no harm - your program will keep right on running normally, but it will return a garbage value.

NOTE: 256 bytes of volatile User RAM is available from $0200 to $02FF, and is reserved exclusively for your use.

POKE (address expression),(data expression)

Set the value of the STM8 memory location indicated by the address expression to the value indicated by the 8-bit byte data expression. If the data expression evaluates to 16 bits (such as variables A through L, and Z) then the value POKE'd is truncated to the value of the least

significant 8-bits of the variable. POKE to a non-existent memory location will cause no harm - your program will keep right on running normally.

NOTE: 256 bytes of volatile User RAM is available from $0200 to $02FF, and is reserved exclusively for your use.

PRINT string

Print to the user console the literal string contained between double-quotes (“”).

Nothing following the closing quote tells PRINT to automatically add a carriage return and line feed, a semi-colon (;) tells PRINT to leave the cursor at the end of the printed string, a comma (,) tells PRINT to TAB 4 spaces after printing the string, and lastly an underscore character (_) will cause a carriage return on the current line with no linefeed after printing the string (good for overwriting something already existing on the same line).

PRINT “”

Prints nothing, only outputs a carriage return followed by a line feed.

PRINT expression

Prints the numeric value of expression to the user console, followed by a carriage return and a line feed.

PRINT variable

Prints the numeric value of the variable, in either decimal or hexadecimal. The default is decimal, and to print a hexadecimal value simply precede the variable with a "$" character (ex: 20 PRINT $A).
Nothing following the variable tells PRINT to automatically add a carriage return and line feed, a semi-colon (;) tells PRINT to leave the cursor at the end of the printed number, a comma (,) tells PRINT to TAB 4 spaces after printing the number, and lastly an underscore character (_) will cause a carriage return on the current line with no linefeed after printing the variable value (good for overwriting something already existing on the same line).

PRINT <nothing following>

You've heard about them, but until now never actually seen one - a real bug that became a legitimate feature!

This version of PRINT, which is only a "PRINT" statement followed by nothing, will print out the line number of the line following this one (if any exists) and do nothing else. The next line will execute normally.

REM any text

BASIC's answer to code commenting. Due to the fact that each byte of a REM takes up

precious BASIC program memory space, it is recommended to use REM only as absolutely needed. The BASIC interpreter does absolutely nothing with a REM statement but to pass over it to the next line.

RETURN

(see description of GOSUB)

RST

Cause a hard reset of the CPU, equivalent to a power-on restart of the STM8 microcontroller.

SLEEP expression

Pauses program execution for the number of seconds indicated by the expression. (Note: due to use of the internal microcontroller oscillator instead of an on-board external oscillator this timing is approximate and may vary slightly from board to board).

USR expression

Causes CPU execution to jump to a machine code routine that begins at the address indicated by expression.
It is the user's responsibility to insure that the user routine performs a proper return – whereupon the BASIC program will continue execution as normal at the line following this line.

NOTE: 256 bytes of volatile User RAM is available from $0200 to $02FF, and is reserved exclusively for your use.

WAIT expression

Pauses program execution for the number of tenth-milliseconds (0.1 ms) indicated by the expression. (Note: due to use of the internal microcontroller oscillator instead of an on-board external oscillator this timing is approximate and may vary slightly from board to board).

variable=expression

A simple variable assignment.

```
STANDARD OPERATORS (grouped by precedence, high to low)
------------------

>                  Greater than
<                  Less than
<=                 Less than or equal to
>=                 Greater than or equal to

*                  Multiply
/                  Divide
%                  Modulus  (Remainder of a Divide)

+                  Add
-                  Subtract

=                  Equate (comparison)
<>                 Not equals


BITWISE OPERATORS (grouped by precedence, high to low)
----------------- (the precedence of all “bitwise”
operators is lower than the precedence of all “standard”
operators)

>>                 (shift right)
<<                 (shift left)

&                  (AND)

^                  (XOR)

|                  (OR)

NOT                (unary operator, ex: “40 Y=NOT X”)
                   (NOTE: Precedence does not apply to “NOT”)
```

```
ASSIGNMENT OPERATOR (lowest of all precedence)
-------------------

=                  Assignment operator
```

VARIABLES

STM8 BASIC Embedded uses system predefined single letter variables. Note that all variables are capital, not lower case, letters.

Variables A through F are signed 16-bit integers (range -32768 to 32767).

Variables G through L, and non-volatile variable Z, are unsigned 16-bit integers (range 0 to 65535 which is $FFFF hexadecimal).

Variables N through W are unsigned 8-bit bytes (range 0 to 255 or $00 to $FF hexadecimal), as are the non-volatile byte variables X and Y.

Z is an unsigned 16-bit integer (range 0 to 65535, or $0000 to $FFFF in hexadecimal) and actually *shares memory space with X and Y such that X = the high byte of Z, and Y = the low byte of Z.* Changes to Z can affect X and/or Y. Any changes to X and Y of course also affect Z.

Variables are not initialized, so be sure to initialize any variables you use before using their contents as they will contain random values after a power-on - with the exception of X, Y, and Z which are located in non-volatile memory.

EDITING

- Edit mode is entered from command mode by using the “EDIT” command. Edit mode is exited by using the Escape key (ESC).

- The Line Editor uses line re-entry to overwrite an existing line.

- Entering a line number with nothing following it deletes the line.

- Line numbers can be in the range of 1 to 32767

- Maximum number of program lines allowed vary per processor and STM8 BASIC version.

BASIC program length is limited by the condition encountered first: either the maximum number of program lines for the given processor or all of program memory used.

Note that each program line includes a termination "NULL" character (a value of zero) that the programmer cannot alter, this NULL is counted as a part of the program for purposes of determining a line's length. Line numbers and all spaces are also counted as part of a program line's length.

For the STM8S103F3 the maximum number of BASIC lines is 100 and the maximum program line length is 32 characters, line numbers and spaces included.

NOTE: REGARDLESS OF THE NUMBER OF PROGRAM LINES, ONLY A MAXIMUM NUMBER OF 630 CHARACTERS (Bytes) IN TOTAL IS ALLOWED for a BASIC program on the STM8S103F3.

REQUIRED INPUT FORMAT

The reasons for requiring this formatting is to keep the code small enough to fit in an STM8 microcontroller with room left to run BASIC. If the BASIC interpreter code expects something to be in a certain place it's alot less code required than for doing a search.

- ALL INPUT WILL BE ATTEMPTED TO BE CONVERTED BY THE BASIC ENVIRONMENT TO UPPERCASE, *"Except within strings in your BASIC code".* It is still the programmer's responsibility to insure that all BASIC code (excluding text inside of quoted strings) must be UPPERCASE.

- NO SPACE IS ALLOWED BEFORE THE LINE NUMBER WHICH MUST BEGIN EACH LINE OF CODE.

- ONE SPACE AND ONLY ONE SPACE must follow the line number (unless deleting the line in Edit Mode)

- Outside of strings ***DO NOT ADD SPACES*** to expressions except as shown below!! Here are some valid examples:

```
50 A=B
210 X=$FF
45 Y=PEEK A
20 POKE A,N (or POKE A N)
30 L=((32*B)/R)+1)
200 FOR I=0 TO T
300 IF X>Y THEN (3000+(D*100))
```

Note that there is never a space on either side of an equals sign or operator, nor inside of an expression.

IMPORTANT MEMORY REGIONS

Hexadecimal Address	Memory region	Notes
$0000	STM8 System RAM	Start
$01FF	STM8 System RAM	End
$0200	User RAM start	User RAM is 256
$02FF	User RAM end	bytes.
$0300	STM8 System RAM	Start
$03FF	STM8 System RAM	End
$4000 (non-volatile)	BASIC prog mem	Start
$4276 (non-volatile)	BASIC prog mem	End
$4277 (non-volatile)	BASIC ENV NV data	Start
$427F (non-volatile)	BASIC ENV NV data	End
$4800	STM8 registers	Start
$7F92	STM8 registers	End
$8000 (PROTECTED)	BASIC ENV/INTRP	Start
$9FFF (PROTECTED)	BASIC ENV/INTRP	End

NON-VOLATILE PROGRAM MEMORY

BASIC program memory for STM8 BASIC is non-volatile, meaning that your BASIC program will not disappear when you power off or reset the processor. To erase your program you must use the “NEW” command or completely overwrite it in Edit mode.

The variables X, Y, and Z are also contained in the Non-volatile (NV) memory area along with a small area reserved for use by the BASIC environment for non-volatile system flags.

STM8 PERIPHERAL BASE REGISTER ADDRESSES

NOTE: *It is highly recommended to refer to documents “stm8s103f3.pdf” and “cd00190271.pdf”, the microcontroller datasheet and reference manual which contains the information needed to program this microcontroller’s registers, available for download at:*

https://www.st.com/resource/en/datasheet/stm8s103f3.pdf

https://www.st.com/resource/en/reference_manual/cd00190271.pdf

```
OPT_BaseAddress         $4800
GPIOA_BaseAddress       $5000
GPIOB_BaseAddress       $5005
GPIOC_BaseAddress       $500A
GPIOD_BaseAddress       $500F
GPIOE_BaseAddress       $5014
GPIOF_BaseAddress       $5019
GPIOG_BaseAddress       $501E
GPIOH_BaseAddress       $5023
GPIOI_BaseAddress       $5028
FLASH_BaseAddress       $505A
EXTI_BaseAddress        $50A0
RST_BaseAddress         $50B3
CLK_BaseAddress         $50C0
WWDG_BaseAddress        $50D1
IWDG_BaseAddress        $50E0
AWU_BaseAddress         $50F0
BEEP_BaseAddress        $50F3
SPI_BaseAddress         $5200
I2C_BaseAddress         $5210
UART1_BaseAddress       $5230
UART2_BaseAddress       $5240
UART3_BaseAddress       $5240
UART4_BaseAddress       $5230
```

```
TIM1_BaseAddress        $5250
TIM2_BaseAddress        $5300
TIM3_BaseAddress        $5320
TIM4_BaseAddress        $5340
TIM5_BaseAddress        $5300
TIM6_BaseAddress        $5340
ADC1_BaseAddress        $53E0
ADC2_BaseAddress        $5400
CAN_BaseAddress         $5420
CFG_BaseAddress         $7F60
ITC_BaseAddress         $7F70
DM_BaseAddress          $7F90
```

USER CALLABLE ‘C’ ROUTINES

The user callable ‘C’ routines are “bonus” features in that they were written for the use of the BASIC Environment and Interpreter, but I am providing you access to the ones that are simple to call and do something potentially useful for you without incurring any harmful side-effects.

All of these are called with the “USR” command, <u>or can be called from your custom machine code routine in RAM</u>. The entry point addresses are provided in a table below the following descriptions of the routines:

List

Executes a “LIST” command, exactly as if issued from the BASIC Environment, including pauses every 10 lines.

ClearScreen

Sends an ANSI Clear Screen command to the terminal (of course, you could use “CLR” to do the same - making this best to be used in your STM8 machine code routines).

Debug

This allows you to perform a single invocation of the same Debug as is available with the immediate command DBGON, except it only runs for the one line you make this call from. Good for checking the variable values at a point of interest in your code.

Reset

Triggers the microcontroller’s internal watchdog circuit, then waits in a forever loop for the microcontroller reset to occur (to guarantee no unwanted behavior), just like an “RST” command (which uses up fewer bytes in your BASIC

code, making this best for use in your own STM8 machine code routines).

ReScanCode

Causes the BASIC Environment to re-scan your BASIC code and create the necessary internal data structures required to execute your code. Where this could be useful is if you somehow modify your own code during it's execution – after making the modifications (the self-modifying code is up to you) before continuing execution you can do 2 things to allow the new code to run correctly, either call this routine or reset the processor with AutoRun on.

And speaking of AutoRun, to enable or disable AutoRun from your code you simply need to set bit 1 at location $427F to 1 to enable, or to 0 to disable. This is a non-volatile setting.

UserSendByte

Refer to the example program "DISPLAY ALL ANSI CHARACTERS". IMPORTANT: You must write the character to be transmitted to RAM location $02FF prior to making the call to UserSendByte.

ENTRY POINT ADDRESSES FOR C HELPER ROUTINES BY VERSION

STM8 BASIC EMBEDDED VERSION 2.3

List	$8644
ClearScreen	$863D
Debug	$8C13
ReScanCode	$85CC
Reset	$8C0D
UserSendByte	$82A1

STM8 BASIC EMBEDDED VERSION 2.2

List	$8623
ClearScreen	$861C
Debug	$8BF2
ReScanCode	$85AB
Reset	$8BEC
UserSendByte	$8280

STM8 BASIC EMBEDDED VERSION 2.1

List	$8638
ClearScreen	$8631
Debug	$8c07
ReScanCode	$85c0
Reset	$8c01
UserSendByte	$8295

EXAMPLE PROGRAMS

"Hello World"

```
10 PRINT "Hello World"
20 END
```

NOTE: The last statement, line ‘20 END’, is not necessary and can be eliminated because when BASIC reaches a line which has no following lines an “END” is automatically assumed.

"Hello World Forever"

```
10 PRINT "Hello World"
20 GOTO 10
30 END
```

NOTE: The last statement, line ‘30 END’, is not necessary and can be eliminated in this kind of situation where it will never be reached.

"Blinky"

The following program will blink the on-board LED one second on and one second off in a “forever loop” while printing out messages in sync with the LED switching on and off.

On your board the User LED is a digital output on Port B, bit 5. Setting bit 5 to 0 in the Port B Output Data register turns on the LED, and of course setting it to 1

turns the LED off. It would be simple to just write a hexadecimal $20 or $00 to the Port B output data register (hexadecimal address $5005) to turn the LED on and off, but we would be overwriting all of the other bits in the register as well. This might be OK in some cases but in other cases you may want to retain any existing bit values in the other 7 bits.

The following code uses a “READ-MODIFY-WRITE” technique to insure that the value of all bits except bit 5 - which controls the on-board user LED - retain their original values as they were before the program was run.

```
10 REM REGISTER SAFE BLINKY
20 PRINT "LED ON"
30 V=PEEK $5005
40 POKE $5005,V&$DF
50 SLEEP 1
60 PRINT "LED OFF"
70 V=PEEK $5005
80 POKE $5005,V|$20
90 SLEEP 1
100 GOTO 20
```

And for the sake of completeness, as well as saving code memory, here is the “QUICK AND DIRTY” method (which still doesn’t toggle any other bits, it just sets them all to zero), which works fine to blink the LED but is not register safe. All we had to do was to remove 4 lines of code and renumber.

```
10 REM QUICK & DIRTY BLINKY
20 PRINT "LED ON"
30 POKE $5005,0
40 SLEEP 1
50 PRINT "LED OFF"
```

```
60 POKE $5005,$20
70 SLEEP 1
80 GOTO 20
```

USR Function example

Both example programs below write a very small STM8 machine code routine consisting of 4 bytes into User RAM starting at address $0200. This routine first pops a single byte off of the internal system “C stack”, discarding it at address $0300 (assuming that location is unimportant, if not you can change it to another), then executing a return instruction to return to the line immediately following the line where the USR function called the machine code routine.

```
10 REM USR WITH POKE
20 POKE $0200,$32
30 POKE $0201,$03
40 POKE $0202,$00
50 POKE $0203,$81
60 REM $0200 = $32 = POP byte to
70 REM $0201,$0202 =  ADDR $0300
80 REM $0203 = $81 = RET
90 PRINT "Making USR $200 call";
100 USR $0200
110 PRINT "..back from USR call"
```

Or a version using the “M” command instead of POKE:

```
10 REM USR WITH M-COMMAND
20 REM $0200 = $32 = POP byte to
30 REM $0201,$0202 = ADDR $0300
40 REM $0203 = $81 = RET
50 REM WRITE STM8 MACHINE CODE:
60 M020032030081
```

```
70 PRINT "Call code @ $0200";
80 PRINT "...";
90 USR $0200
100 PRINT "RETURNED!"
```

USER RAM DUMP

```
4 PRINT ""
5 PRINT "User RAM dump"
6 PRINT "-------------"
7 PRINT ""
10 FOR A=$0200 TO $02FF
20 PRINT "Addr: $";
30 PRINT $A;
40 PRINT " Data: $";
50 V=PEEK A
60 PRINT $V
70 NEXT A
```

PEEK 8 CONSECUTIVE BYTES

```
50 REM PEEK 8 BYTES @ ANY ADDR
100 PRINT "Addr: ";
200 INPUT A
210 PRINT $A;
220 PRINT " :";
230 B=A+7
300 FOR L=A TO B
600 N=PEEK L
650 PRINT " ";
700 PRINT $N;
800 NEXT L
850 PRINT ""
999 END
```

SOFTWARE PWM (PULSE WIDTH MODULATION)

```
1 REM 'BREATHING' LED
2 REM USING SOFTWARE PWM
3 REM (PULSE WIDTH MODULATION)
4 REM TO VARY THE LED INTENSITY
5 A=0
7 B=150
10 POKE $5005,0
15 WAIT A
20 A=A+1
25 IF A=150 THEN 2000
30 POKE $5005,$20
40 WAIT B
42 B=B-1
50 GOTO 10
60 A=150
70 B=0
80 POKE $5005,0
90 WAIT A
100 A=A-1
110 POKE $5005,$20
120 WAIT B
130 B=B+1
135 IF B=150 THEN 1000
150 GOTO 80
1000 WAIT 1000
1010 GOTO 5
2000 WAIT 1000
2010 GOTO 60
```

LUNAR LANDER GAME

This is a re-creation of one of the earliest BASIC computer games, the famous “Lunar Lander” program.

The goal is to land on the moon in your lunar lander at a downward velocity of less than 5. You begin at an altitude of 1000 with a downward velocity of 70, and with 500 units of fuel in your tank.

At each turn you can burn from 0 to 250 units of fuel. Make too big of a burn and you can achieve a negative downward velocity and actually begin climbing higher in altitude! Burn too little and you risk descending so fast that even burning the maximum fuel won’t slow you down enough to land softly.

It’s very possible to make a good landing (I’ve done it with this program), but it’s harder than you think. The first several times you will most likely just be adding more craters to the moon!!! ;)

```
1 PRINT "Land@<5"
3 L=70
4 F=500
5 H=1000
6 GOSUB 33
7 GOSUB 55
8 H=H-L
9 IF H<=0 THEN 21
10 L=(((L+2)*10)-(K*2))/10
11 IF F<=0 THEN 17
12 F=F-K
15 IF H>0 THEN 6
16 GOSUB 33
17 F=0
18 GOTO 6
```

```
20 IF L>5 THEN 24
21 PRINT ""_
22 PRINT "Xlnt!"
23 END
24 PRINT ""_
25 PRINT "URA Crater!"
26 END
33 WAIT 5000-(L*L)
34 PRINT "*"
35 PRINT "Fuel:";
36 PRINT F
37 PRINT "Vel:";
39 PRINT L
41 PRINT "Alt:";
43 IF L>H THEN 20
44 PRINT H
45 RETURN
55 IF F<=0 THEN 82
57 PRINT "?";
59 INPUT K
60 IF K>250 THEN 76
61 IF K<0 THEN 73
63 IF K<=F THEN 81
65 PRINT "LowF"
67 GOTO 57
73 K=0
76 PRINT "Err"
77 GOTO 55
81 RETURN
82 K=0
83 RETURN
```

SOUND FROM THE TERMINAL USING ASCII “BELL” CODE

```
10 POKE $2FF,7
20 USR $8280
```

The above and below programs use a call to a ‘C’ language routine in the BASIC Environment code (called UserSendByte) that you can use to directly output to the serial port any byte value from 0 to $FF (255).

UserSendByte expects the byte to be sent to first be written to RAM address $02FF, the last byte of the User RAM area, before UserSendByte is called with “USR $8280”.

DISPLAY ALL ANSI CHARACTERS

Using this method this will output the code value in hexadecimal, followed by a ‘:’, then followed by the character itself as displayed on the terminal. A partial capture of the output of this program - all of the actual graphics characters excluding normal characters are shown below the program listing.

```
5 CLR
10 FOR T=0 TO $FF
12 PRINT $T;
14 PRINT " : ";
20 POKE $02FF,T
30 USR $8280
40 SLEEP 1
45 PRINT ""
50 NEXT T
60 PRINT ""
70 PRINT "Done"
```

```
A0 : á
A1 : í
A2 : ó
A3 : ú
A4 : ñ
A5 : Ñ
A6 : ª
A7 : º
A8 : ¿
A9 : ⌐
AA : ¬
AB : ½
AC : ¼
AD : ¡
AE : «
AF : »
B0 : ░
B1 : ▒
B2 : ▓
B3 : │
B4 : ┤
B5 : ╡
B6 : ╢
B7 : ╖
B8 : ╕
```

```
B9 : ╣
BA : ║
BB : ╗
BC : ╝
BD : ╜
BE : ╛
BF : ┐
C0 : └
C1 : ┴
C2 : ┬
C3 : ├
C4 : ─
C5 : ┼
C6 : ╞
C7 : ╟
C8 : ╚
C9 : ╔
CA : ╩
CB : ╦
CC : ╠
CD : ═
CE : ╬
CF : ╧
D0 : ╨
D1 : ╤
```

```
D2 : ╥
D3 : ╙
D4 : ╘
D5 : ╒
D6 : ╓
D7 : ╫
D8 : ╪
D9 : ┘
DA : ┌
DB : █
DC : ▄
DD : ▌
DE : ▐
DF : ▀
E0 : α
E1 : ß
E2 : Γ
E3 : π
E4 : Σ
E5 : σ
E6 : µ
E7 : τ
E8 : Φ
E9 : Θ
EA : Ω
```

```
EB : δ
EC : ∞
ED : φ
EE : ε
EF : ∩
F0 : ≡
F1 : ±
F2 : ≥
F3 : ≤
F4 : ⌠
F5 : ⌡
F6 : ÷
F7 : ≈
F8 : °
F9 : ∙
FA : ·
FB : √
FC : ⁿ
FD : ²
FE : ■
FF :
```

ANSI Character Graphics codes (in hex), with output

DUMP BASIC PROGRAM MEMORY

This program will dump the contents of the non-volatile BASIC program memory, which begins at address $4000 and is 630 bytes in size. It displays first the address, then the hexadecimal value at that address, and lastly the ASCII character represented by that value. <u>Note that to get the ASCII character to output we need to use a call to the UserSendByte helper routine</u>.

```
10 FOR A=$4000 TO A+630
20 PRINT "$";
30 PRINT $A;
40 PRINT " : $";
50 T=PEEK A
60 PRINT $T;
70 PRINT " : ";
80 POKE $02FF,T
90 USR $8280
100 PRINT ""
110 NEXT A
```

<u>*A good thing to note*</u> is that the interpreter determines program end when it encounters two NULLs in a row (NULL = 0). A single NULL terminates each line of BASIC code. If one is careful, the “left over” area of BASIC program memory <u>beyond the 2 NULLs</u> can be used for anything you wish (and it’s non-volatile) as long as you remember that editing your program may well overwrite whatever you put there, and of course a “NEW” will erase it completely. If your program is stable and not going to change (so you don’t “EDIT” or “NEW”) then you won’t have any issues using this “trick”.

ANALOG TO DIGITAL CONVERSION ON PORT PIN D3 USING ADC1

This example program demonstrates how to do an Analog to Digital conversion that will measure a voltage between 0 and 3.3VDC (the min and max values for the ADC input).

(WARNING: Do not attempt to measure a voltage greater than 3.3VDC with a direct connection to port pin D3!!! - the ADC reading will be incorrect and you may possibly damage your processor!)

(NOTE: lines beginning with a ‘*’ are not part of the code, but are there to help your understanding of the code)

```
*STM8 CBE ADC CONVERSION
*Setup port D
*Set to Input
1 POKE $500F,0
*No internal pullup
2 POKE $5010,0
*End Of Conversion interrupt disabled
3 POKE $5011,0
*Use ADC1 (Base Address = $53E0)
*Address of Control Status Reg = $5400
*Address of Config Reg 1 = $5401
*Address of Config Reg 2 = $5402
*Address of Data high byte register = $5404
*Address of Data low byte register = $5405
*Set ADC1 Channel to AIN4 input in Control Status
*Register (Input is port/pin D3)
40 POKE $5400,4
*Set ‘Left align data’ using Config Reg 2
60 POKE $5402,0
*Enable ADC1.  First POKE “turns on” the ADC.
*Subsequent POKEs trigger an ADC conversion.
70 POKE $5401,1
*Do conversions and output results
```

```
*Clear the screen
100 CLR
*Trigger a new conversion
105 POKE $5401,1
*Wait for conversion done bit set
*(bit 7 of Control Status Register)
110 L=PEEK $5400
120 IF L&$80 THEN 200
130 GOTO 110
*Conversion complete,
*Reset conversion complete bit
200 POKE $5400,4
*Read the data, high byte, then low byte
210 X=PEEK $5404
220 Y=PEEK $5405
*Print the result in millivolts
*65535 max counts = 3300 mv
*Here we take advantage of the X,Y & Z vars relationship
*for one of it’s intended uses - combining the values of
*two 8-bit registers (a “high byte” register and a “low
*byte” register) containing a single 16-bit value.
*Remember X is Z’s high byte, Y is Z’s low byte
310 PRINT Z/19
*Wait a half second...
320 WAIT 5000
*Do it all over again
400 GOTO 100
```

SET CONSOLE TEXT COLOR

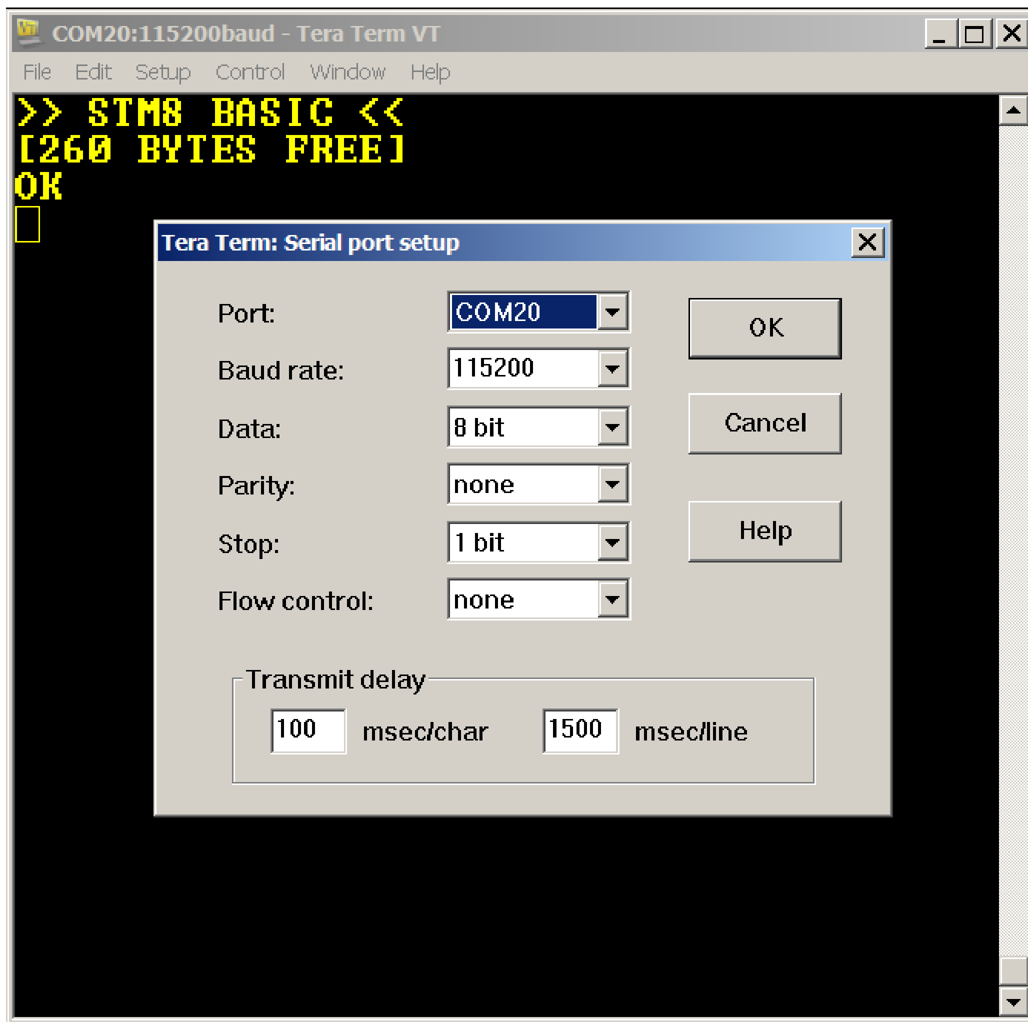
Sets the console text color to your choice of 7 different colors. The chosen text color is also saved to the system flags area (address $427E) of non-volatile EEPROM so that it is retained through successive resets or power cycles.

```
1 CLR
5 M02001B5B33
10 A=PEEK $427E
12 PRINT "---------------"
13 PRINT "RED = 1"
14 PRINT "GREEN = 2"
15 PRINT "YELLOW = 3"
16 PRINT "BLUE = 4"
17 PRINT "MAGENTA = 5"
18 PRINT "CYAN = 6"
19 PRINT "WHITE = 7"
20 PRINT "Current color: ";
30 PRINT A
40 PRINT "New color? ";
50 INPUT A
60 POKE $427E,A
70 FOR B=$0200 TO $0202
80 D=PEEK B
90 POKE $02FF,D
100 USR $8280
110 NEXT B
120 POKE $02FF,A+$30
130 USR $8280
140 POKE $02FF,$6D
150 USR $8280
160 PRINT ""
170 GOTO 10
```

Program Output

LOADING AND SAVING

Although there is no LOAD nor SAVE command in STM8 BASIC Embedded as there is in BASIC on a PC there is a very good way to accomplish loading and saving of your BASIC programs to and from a PC using the “Tera Term” freeware serial terminal program. Most other terminal programs have similar settings and capabilities.



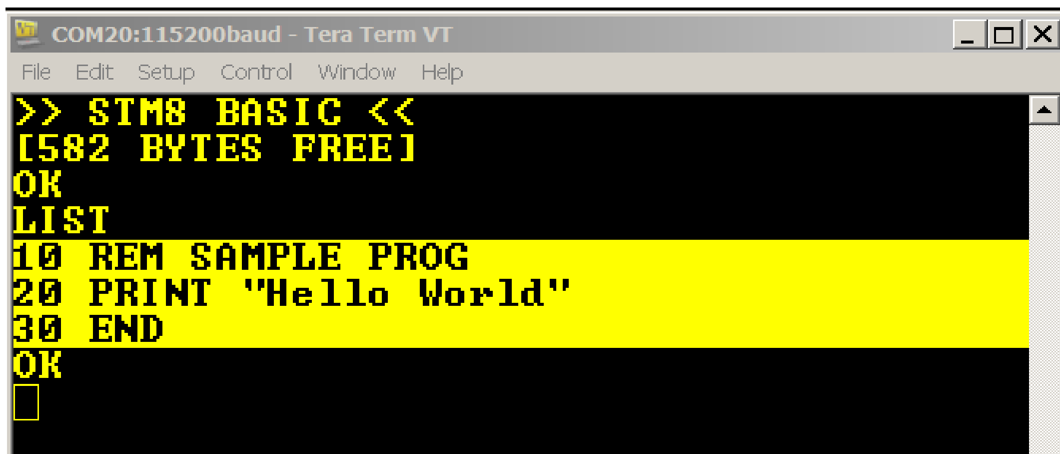
Tera Term Serial Port Settings for STM8 BASIC Embedded. Note the “Transmit delay” settings, these are important to have set correctly for loading of your BASIC programs.

Saving a program via Tera Term is as easy as copy and paste. First open up a new text document, giving it the name you want your program to have. If you want to be able to work with your BASIC programs on the PC you will need a text editor, usually giving your BASIC program a file extension of “.txt”.

You might also prefer to have your BASIC programs saved to the PC with the extension “.BAS”, the standard for BASIC program files. In this case you might need to set up your preferred editor to open on “.BAS” files.

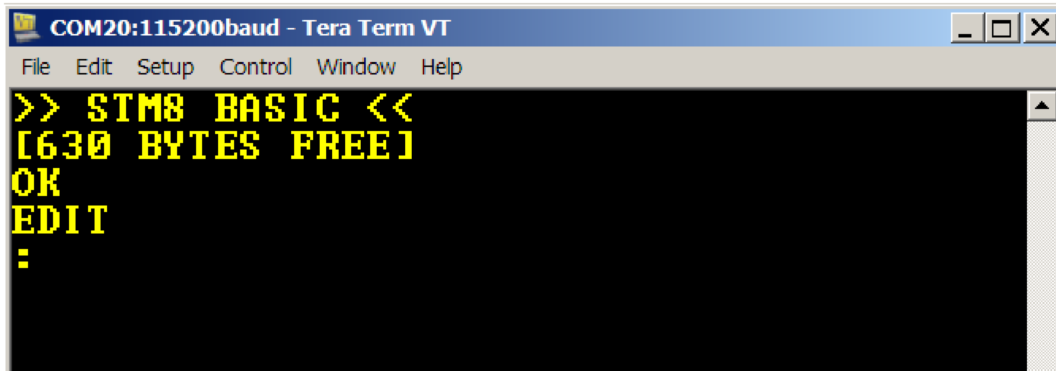
Either of the above naming conventions will work just fine as long as you stick to your favorite convention.

The screenshot below shows selecting a block of code in Tera Term that was output by a “LIST” command, in command mode:

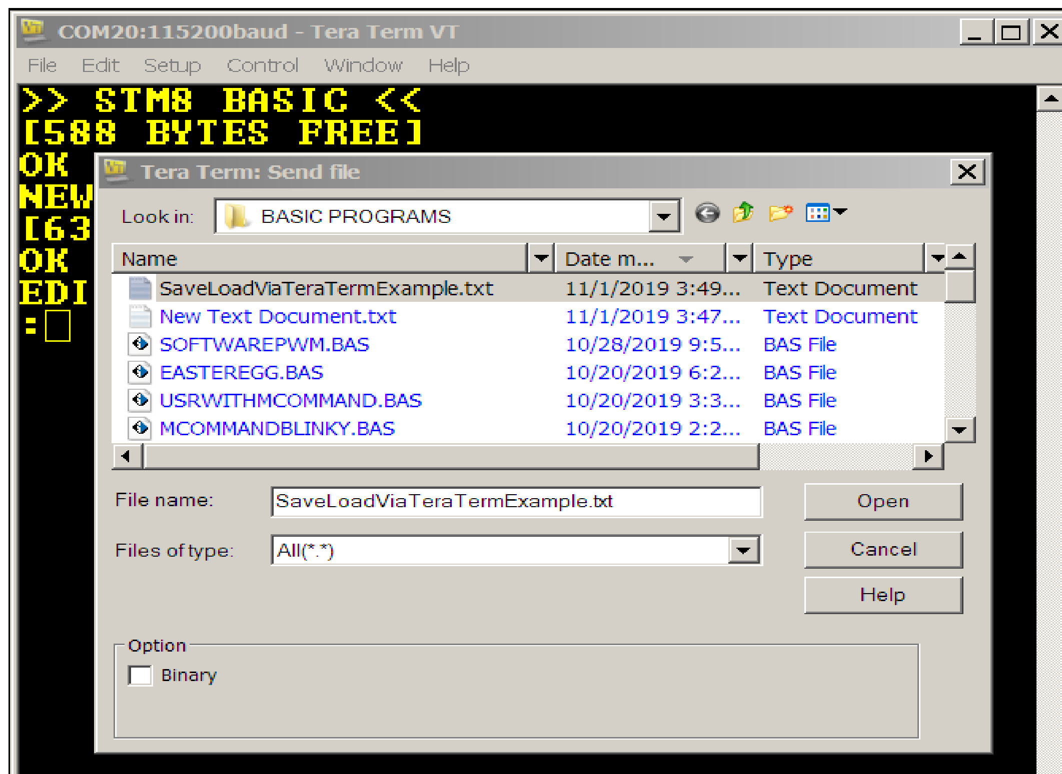


Using the Edit/copy of Tera Term copy the BASIC program text and then paste it into the editor of your choice and save it under your desired file name. Don’t add anything else to the file other than your lines of BASIC code, because this is code you will later load back into STM8 BASIC Embedded as will be explained in the following paragraph.

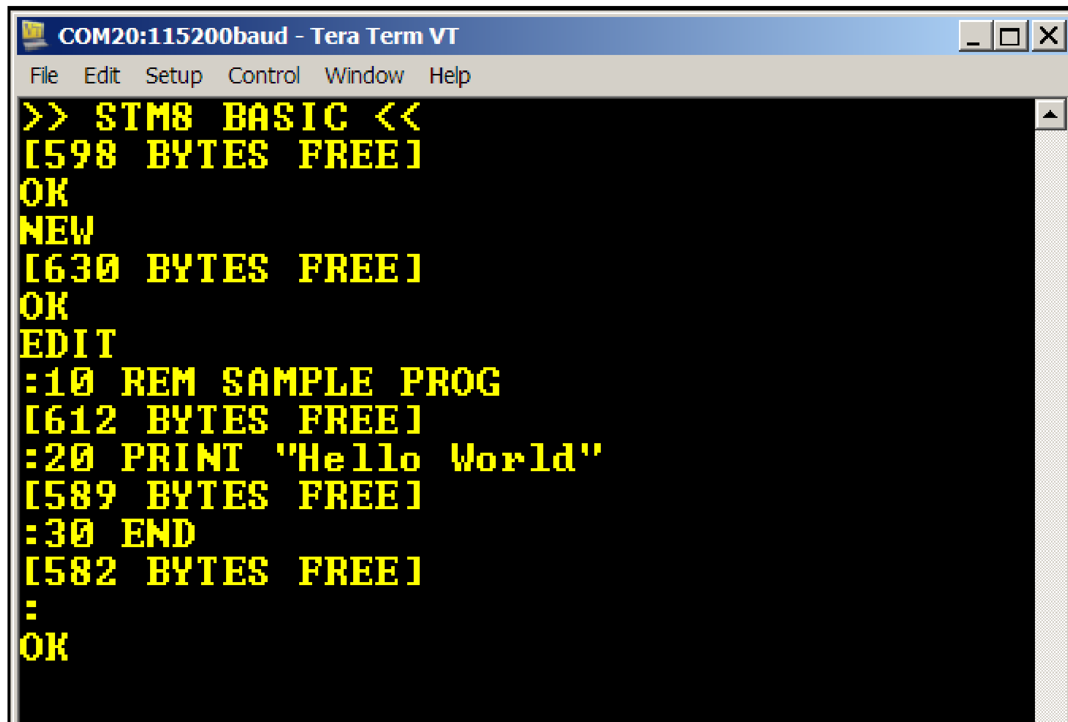
To load a saved BASIC program simply enter the Edit mode after having executed a “NEW” command to clear out BASIC program memory, if needed. Now you are ready to load a BASIC program.



Next choose a file to load using Tera Term’s File/Send facility, as shown below:



Notice that loading is simply having the timing of the terminal program's sending of the text file set correctly, as far as the BASIC environment knows it is a human that is typing in the program. The end result is the same as shown below:



Your program is now loaded and ready to run or modify.

STM8 BASIC ERROR CODES

E1 Basic code memory is full
E2 String Overflow (internal system string error, should not occur - but just in case...)
E30 or E31 GOTO or GOSUB with no matching line number
E4 NEXT without a FOR
E5 FOR/NEXT table size exceeded (too many FOR/NEXT pairs)
E6 'M' is not a valid variable name
E7 Missing line number (check made when entering a line of code)
E8 Invalid Hex Digit (must be only '0' through '9' or uppercase 'A' through 'F')
E9 Evaluation Value Stack Overflow (Your expression is too complex, break it up into two or more lines of code)
E10 Evaluation Operator Stack Overflow (same problem resolution as for error E9)
E11 Invalid expression operator (see list of valid operators)

Non-program related errors may also appear in the Command and Edit modes as a series of one or more question marks instead of an “OK” prompt.

Please note that in such a small amount of memory it simply is not possible for STM8 BASIC to guard against or be able to report every possible error that a user may make in either the environment or in their BASIC programs - especially logic errors. In such cases you may experience unpredictable behavior, yet in any case you can always press the reset button to get back to the initial banner & prompt or auto-running your program again if you previously invoked the ARON command (not recommended until you have a fully stable program). There is no syntax checking, if the interpreter doesn’t understand your line of code it will usually skip it and go on to the next line.

PLEASE NOTE, that for intellectual property protection purposes any attempt at reading out the code memory through the development/debug port for the STM8 BASIC Environment & Interpreter will result in all code memory being irreversibly wiped clean using ST Micro's built-in hardware code protection capability. Therefore, it is not recommended to try to access this code or you will end up owning a blank microcontroller.

Any attempt at accessing, copying, distributing, and/or reverse engineering STM8 Compact BASIC Embedded is a violation of intellectual property laws and violators will be prosecuted to the full extent of the law.

STM8 Compact BASIC Embedded was proudly made in the United States of America.

STM8 BASIC is a trademark of Jeremiah 31:10 Ministries, Inc. a non-profit organization.