Terminal-BASIC

User reference manual

Andrey Skvortsov <starling13@mail.ru>

http://starling13.clan.su

Version 2.3-a1

11/06/20 16:02:55

Contents

1 Introduction

Terminal-BASIC (TB) is a free implementation of BASIC programming language. It is s partially compliant to the standards ISO/IEC 6373:1984 and USSR/Russian ГОСТ 27787-88 and also supports some features, allowing to use many BASIC type-in utilities and games from old BASIC books and magazines.

TB was inspired by the TinyBASIC port for Arduino-compatible uC-based systems and Darthmouth BASIC - the World’s first BASIC system.

TB is designed to run on the simplest embedded systems, based on microcontrollers and microprocessors with the program memory of at least 16kb and 1kb of RAM (such as AVR 8-bit Arduino boards), but it is cross-platform, has Linux and Windows versions, suitable for evaluation and software debugging.

The main features of TB interpreter:

- supports number of data types (integer (2 bytes signed), long integer (4 bytes signed), real (4 bytes binary floating point), long real (8 bytes binary floating point), boolean and string) using variables and function suffixes;
- multidimensional arrays of arbitrary size and dimensions;
- Dartmouth-BASIC-like matrix operations;
- optional time-sharing system mode with round-robin scheduling using multiple I/O devices for each user (i.e. USART);
- configuration headers provide the number of options, which enable inclusion of the language parts and features thus allowing to adjust the code size.

2 Getting and installing Terminal-BASIC

If you don’t plan to improve/develop Terminal-BASC and work with the source repositories, you can download precompiled packages and arduino sketches from TB sourceforge site: https://sourceforge.net/projects/terminal-basic/files .

2.1 Example 1: POSIX-cmpatible systems

Pure posix version (Linux, *BSD, etc.) has all features except graphical capabilities (see SDL2 version instead).

Default configuration files enables all the necessary options, but you may wish to look at them and change somethng, before building from source package.

3 Commands and functions reference

3.1 Data types

Constants, expressions, variables and arrays can represent different data types:

- integer values (2 bytes signed [-32768; 32768));
  - Variable and array names should end with % suffix: A%, VR1%, BUF%(10).
- long integer (4 bytes signed [-2147483648; 2147483648));
  - Variable and array names should end with %! suffix: A%!, VR%!, BF%!(10).
- floating point real (4 bytes single precision IEEE 754);
  - Variable and array names do not use specific suffix: A, VAR1, BUF(10).
- floating point long real (8 bytes double precision IEEE 754);
  - Variable and array names should end with ! suffix: A!, VR1!%, BUF!(10)
- logical (1 bit, arrays are bit-packed and very compact);
  - Variable and array names should end with @ suffix: A@, VR1@, BUF@(10)
- string (up to 72 characters);
  - Variable and array names should end with $ or ¤ suffix: A$, VR1¤, BUF$(10).

3.2 Language core

3.2.1 Commands

CHAIN

Description: Load a program text from non-volatile memory, preserving the existing variables and arrays.

CONT

Description: Continue execution of a program, stopped by a STOP command.

DATA

Description: Start a section of the data

Example:

```
10 DATA 1,2,3,4,5
20 READ A%,B%
30 READ C%
```

```
40 RESTORE : READ D%,E%,F%,G%,H%
50 PRINT A%;B%;C%;D%;E%;F%;G%;H%
RUN
 1 2 3 1 2 3 4 5
```

DEF

Description: Define a one-line user function

Example:

```
10 DEF FN HYP(X,Y) = SQR(X*X + Y*Y)
20 LET A = 12 : LET B = 14
30 PRINT FN HYP(A,B)
RUN
 18.4391
```

DELAY

Description: Delay execution of a program for interval in milliseconds.

Syntax:

DELAY <expression>

DIM

Description: Define an array

Syntax:

DIM <name>(expression[, expression]*), [<name>(expression[, expression]*)]*

Example:

```
DIM A(12)
A(0) = 3.141592
LET I%=3,J%=2
DIM C%(I%,J%,I%*J%), D%(I%+J%)
```

DUMP

Description: Print contents of BASIC memory, variables and arrays

END

Description: Stop program execution and return to interactive mode.

FN

Description: Look at DEF.

FOR

Description: Make an iterative loop.

Example:

```
10 FOR I%=1 TO 6 STEP 2
20   FOR J%=1 TO 3
30     PRINT I%;J%
40   NEXT J%
50 NEXT I%
  1 1
  1 2
  1 3
  3 1
  3 2
  3 3
  5 1
  5 2
  5 3
```

GOTO

Description: Explicit jump to specified program line

Syntax:

GOTO <integer expression>

Example:

```
10 REM ENDLESS LOOP
20 PRINT I% : I% = I% + 1
30 GOTO 20
40 END
```

GOSUB

Description: Explicit jump to subroutine with the ability to return to the call point by RETURN.

Syntax:

GOSUB <integer expression>

Example:

```
10 INPUT A
20 GOSUB 1000
30 PRINT A
40 END
1000 REM SUBROUTINE
1010 A = A ^ 2
1020 RETURN
```

INPUT

LET

Description: Set variable or array element value of the expression.

Syntax:

LET <variable | arrayElement> = <expression>

Example:

```
LET A% = 3
PRINT A%
  3
LET B(1,2) = PI()
PRINT B(1,2)
  3.141592
```

LIST

Dexcription: List current program text.

Syntax:

LIST [startLine[-endLine]]

LOAD

Description: Load program from non-volatile memory, saved previously by the SAVE command.

MAT

Description: Command, performing matrix operations

NEW

Description: Free interpreter memory. No program text, variables, arrays or user functions remain after execution.

NEXT

Description: Next loop iteration, look at FOR.

POKE

PRINT

Description: Output the expressions values to the standart output.

Syntax:

PRINT [[expression | ; | ,]*

Examples:

```
PRINT «Hello, World!»
Hello World
```

```
PRINT «First line»
PRINT
```

```
PRINT «Second one»
First line

Second one
```

```
PRINT 2*2
4
```

```
PRINT «47.31+24.5^2=», 47.31+24.5^2
47.31+24.5^2=       647.56
```

READ

Description: Read a value from data section. Look at DATA;

REM

Description: Make a comment line. Al symbols after REM command to end of line will not be interpreted.

RESTORE

Description: Reset current data section pointer. Next READ statement will start from the begining of the data. Look at DATA.

RETURN

Description: Return from a subroutine, entered by the GOSUB.

RUN

Description: Run a program, stored in BASIC memory

SAVE

Description: Save program to non-volatile storage for loading with the LOAD command.

STEP

Description: Look at FOR.

STOP

Description: Stop a program execution with the ability to continue later by the CONT command.

TO

Description: Look at FOR.

3.2.2 Functions

ABS

Description: Returns the absolute value of its argument.

CHR$

Description: Returns one-character string with the ASCII code, defined by the parameter.

HEX$[1]

Description: Convert numeric expression value to string, containong hexadecimal representation of the value.

INKEY$

Description: Read input character. Unlike INPUT command, there is no waiting for actual input.

INT

RES

RND

SGN

STR$

Description: Return string with the decimal representation of the argument numeric expression.

TIME

3.2.3 Operations

+

Addition

-

Subtraction or unary minus

*****

Multiplication

1 Controlled by option USE_HEX

/

Division

****

Integer division

^

Power

AND

Logical multiplication

DET

Matrix determinant

INV

Matrix inverted

MOD

Integer division remainder

OR

Logical addition

TRN

Matrix transposed

XOR

Logical exclusive or

3.2.4 Constants

CON

Matrix initializer to ones

FALSE

Logical false

IDN

Matrix initializer to identity

TRUE

Logical true

ZER

Matrix initializer to zeros

3.3 Mathematics module

3.3.1 Functions

ACS

ASN

ATN

CBR

COSH

COS²

COT

EXP

LOG

PI

SIN³

SQR

TAN

3.4 String manipulation module

3.4.1 Functions

ASC

2

3

CHR$

HEX$

MID$

LEFT$

RIGHT$

3.5 General purpose IO module

3.5.1 Commands

AWRITE

DNOTONE

DTONE

DWRITE

3.5.2 Functions

AREAD

AREAD%

DREAD

3.6 Filesystem operations module

3.6.1 Commands

DCHAIN

Description: Command equivalent to the sequence of DLOAD и RUN, except that the state of the running program (variables and arrays) is preserved.

Syntax:

DCHAIN <file name string expression>

DIRECTORY

Syntax:

DIRECTORY [firstFileIndex [, lastFileIndex]]

Description: Print external memory file list

DLOAD

Description: Load program text from file. The file should have BAS extension, but the command parameter has no extension.

Syntax:

DLOAD <BAS file name without BAS extension>

DSAVE

Description: Save current program to text file. Файл будет иметь расширение .BAS, но его имя в команде вводится без расширения. Если файл с указанным именем существовал, он будет перезаписан.

Syntax:

DLOAD <file name without extension>

FCLOSE

Description: Close previously opened with the FOPEN command file.

Syntax:

FCLOSE <file number>.

Example: look at FOPEN.

FSEEK

Description: move file rwead/write cursor to specified position.

FWRITE

Description: Записать в файл 1 байт

Syntax:

FWRITE <целочисленное выражение>, <дескриптор файла>

HEADER

SCRATCH

3.6.2 Functions

FOPEN

Description: Open file in external memory.

Parameters: file name

Return: Integer non-negative file number or -1 if error occures.

Example:

```
F% = FOPEN(«TEST.TXT»)
PRINT F%
0
FCLOSE F%
F% = FOPEN(«TET.TXT»)
PRINT F%
-1
```

FREAD

Description: Read next byte from file and move cursor one position forward.

Parameters: file number of the file, previously opened with the OPEN command.

Return: Значение прочитанного байта, как беззнаковое целое от 0 до 255 или -1 в случае невозможности чтения (в т.ч. достижении конца файла)

Example:

```
  5 REM Print text file content
 10 F% = FOPEN(«TEST.TXT»)
 20 IF F%=-1 THEN GOTO 110
 30 B% = FREAD(F%)
 40 IF B% = -1 THEN PRINT «End of file» : GOTO 100
 50 PRINT B%; : IF B%=10 THEN PRINT CHR$(13);
 60 GOTO 30
100 FCLOSE F%
110 END
```

FSIZE

Description: Получение размера файла

Параметры: дескриптор ранее открытого файла

Возвращаемое значение: размер файла в байтах

3.6.3 Autorun program

Для автоматического выполнения программного кода после загрузки интерпретатора ТБ, в корневом каталоге файловой системы необходимо создать файл программы с именем AUTORUN.BAS.

3.7 Graphics output module

3.7.1 Commands

BOX

Syntax:

BOX <x>,<y>,<width>, <height>

CIRCLE

Syntax:

CIRCLE <x>,<y>,<radius>

COLOR

ELLIPSE

Syntax:

ELLIPSE <x>,<y>,<width>, <height>

CURSOR

LINE

Syntax:

LINE <x1>,<y1>,<x2>, <y2>

LINETO

POINT

Syntax:

POINT <x>,<y>

SCREEN

4 National lexical sets

English	Русский	Français
AND	И	AND
DATA	ДАННЫЕ	DATA
DEF	ОПР	DEF
DIM	РАЗМЕР	DIM
END	КОНЕЦ	FIN
FN	ФУНК	FN
FOR	ДЛЯ	POUR
GOSUB	ВХОД	GOSUB
GOTO	НА	GOTO
IF	ЕСЛИ	IF
INPUT	ВВОД	INPUT
LET	ПУСТЬ	LET
LIST	ЛИСТАТЬ	LISTER
MAT	МАТ	MAT
NEXT	ЦИКЛ	NEXT
NOT	НЕ	NOT
ON	ПРИ	ON
OR	ИЛИ	OR
PRINT	ВЫВОД	PRINT
READ	ВЗЯТЬ	READ
REM	КОМ	REM
RESTORE	СНОВА	RESTORE
RETURN	ВОЗВРАТ	RETOUR
RUN	ПУСК	RUN
STEP	ШАГ	STEP
STOP	СТОП	STOP
THEN	ТО	THEN
TO	ДО	JUSQUA

5 Language extension creation

In order to add the ability to interpret new commands and functions to TB, it is not necessary to change the language core code. A set of functions and commands can be grouped in an extension module, an example of development of which is given in this section.

In this example, a module will be created containing the MULTPRINT command and the HYPOT function. The MULTPRINT command takes two parameters: the first is an integer expression, the second is a string expression. The result of the command will be displaying the string specified by the second parameter in the amount specified by the first:

```
MULTPRINT 3, "123"
123
123
123
```

The HYPOT function takes two real parameters a and b and returns the value of the function $z=\sqrt{x^2+y^2}$:

```
PRINT HYPOT(2.1, 0.145)
2.105
```

The extension module is a C ++ class inherited from the BASIC :: FunctionBlock class. The module code will be in the files test_module.hpp and test_module.cpp.

The header file test_module.hpp should contain a description of the module class:

```
#ifndef TEST_MODULE_HPP
#define TEST_MODULE_HPP

#include "basic_functionblock.hpp"

class TestModule : public BASIC::FunctionBlock
{
public:
    TestModule();
};

#endif // TEST_MODULE_HPP
```

The test_module.cpp implementation file will be almost empty at first, and contains only the empty default constructor body.

```
#include "test_module.hpp"
#include "basic_interpreter.hpp"

TestModule::TestModule()
{
```

```
}
```

Next, you need to connect a still empty module to the interpreter.

In the ucbasic_main.cpp file (or in the sketch-terminal-basic.ino file), you must add the directive to include the module header file before starting the module declaration:

```
...
#if USE_SDL_ISTREAM
#include "sdlstream.hpp"
#endif

#include "test_module.hpp"

/**
 * Instantiating modules
 */

#if USEUTFT
...
```

Then in the same file you need to create an instance of the module:

```
...
/**
 * Instantiating modules
 */

static TestModule testModule;

#if USEUTFT
...
```

Finally, before initializing the interpreter, you must add the module to the list of active ones:

```
...
#if USESD
    basic.addModule(&sdfs);
#endif

    basic.addModule(&testModule);

    basic.init();
...
```

Compilation and launch of the program should go without errors, but no new available commands and functions will appear yet.

To add the MULTPRINT command in the module class, you must create a static function for implementing this command. The function name can be any:

```
class TestModule : public BASIC::FunctionBlock
{
```

```
public:
     TestModule();

     static bool comm_multprint(BASIC::Interpreter&);
};
```

The implementation of commands and functions is always a function that returns a value of type bool and takes one parameter — a reference to the interpreter object that called the command or function.

The implementation in test_module.cpp remains empty for now:

```
TestModule::TestModule()
{
}

bool
TestModule::comm_multprint(BASIC::Interpreter& i)
{
     return true;
}
```

The return value indicates whether the command / function completed successfully or something went wrong and it is necessary to signal the occurrence of a run-time error. An empty implementation for now just returns true;

Now you need to give the team a name by which it will be called in the program text. To do this, a module symbol table consisting of two arrays is created in the test_module.cpp file, and in the class constructor pointers to these arrays are passed to the parent class:

```
#include "test_module.hpp"

static const uint8_t tmCommandTokens[] PROGMEM = {
     'M','U','L','T','P','R','I','N','T',ASCII_NUL,
     ASCII_ETX
};

static const BASIC::FunctionBlock::command tmCommandImps[] PROGMEM =
{
     TestModule::comm_multprint
#if FAST_MODULE_CALL
     , nullptr
#endif
};

TestModule::TestModule()
{
     commands = tmCommandImps;
     commandTokens = tmCommandTokens;
}
```

After building and running, the MULTPRINT command called without parameters is

executed without error messages.

Implementation of this command will require the use of the values of the actual parameters specified when it was called. Parameter values are passed on the interpreter stack in reverse order. In this case, the first parameter is the numerical value of the number of output lines, and the second is the output line itself. Thus, first a line will be received from the stack (2nd parameter), and then the number of its outputs (1st parameter):

```
bool
TestModule::comm_multprint(BASIC::Interpreter& i)
{
     // 1. Getting a string
     const char *str; // Pointer to the string in the stack
     if (i.popString(str)) {  // If the last parameter was a string
          // 2. Getting the number of string output lines
          BASIC::Parser::Value v;  //Universal value object
          if (i.popValue(v)) {     // If there was one more parameter
                                   // before the string
               // Explicit conversion of the 1st parameter to integer
               BASIC::Integer num = BASIC::Integer(v);
               // All data received, execution of the command body
               while (num-- > 0) {
                    i.print(str);
                    i.newline();
               }
               return true; // Executed successfully
          }
     }
     return false; // Error in all another cases
}
```

After assembly and launch, the command works:

```
MULTPRINT 3, "123"
123
123
123
```

However, the numerical parameter was explicitly cast to the integer without checking the actual value, which can lead to undesirable flexibility of the command:

```
MULTPRINT 2.9999, "Real truncation"
Real truncation
Real truncation

…

MULTPRINT TRUE, "True is 1, False is 0 ?"
True is 1, False is 0 ?
```

The following change will not allow the command to run if the first parameter is not an integer:

```
bool
TestModule::comm_multprint(BASIC::Interpreter& i)
{
     // 1. Getting a string
     const char *str; // Pointer to the string in the stack
     if (i.popString(str)) {  // If the last parameter was a string
          // 2. Getting the number of string output lines
          BASIC::Parser::Value v;  //Universal value object
          if (i.popValue(v)) {     // If there was one more parameter
                                   // before the string
               if (v.type() == BASIC::Parser::Value::INTEGER) {
                    // Explicit conversion of the 1st parameter
                    // to integer
                    BASIC::Integer num = BASIC::Integer(v);
                    // All data received, execution of
                    // the command body
                    while (num-- > 0) {
                         i.print(str);
                         i.newline();
                    }
                    return true;
               }
          }
     }
     return false;
}
```

Now, when passing a parameter that is not an integer, a runtime error of 17 is issued:
Error executing the command:

```
MULTPRINT 2+1, "123"
123
123
123

MULTPRINT 3.0001, "123"
SEMANTIC ERROR 17
```

Adding the HYPOT function is similar. First, a module class function is created to implement the new TB function:

test_module.hpp

```

class TestModule : public BASIC::FunctionBlock
{
public:
     TestModule();

     static bool comm_multprint(BASIC::Interpreter&);
     static bool func_hypot(BASIC::Interpreter&);
};
```

The difference between functions and commands is that if a function completes successfully, then before its completion it is necessary to place an object of universal value containing the result of the function on the interpreter stack:

test_module.cpp

```
    }
    return false;
}

bool
TestModule::func_hypot(BASIC::Interpreter& i)
{
    BASIC::Parser::Value v1; // Return value object
                             // Integer 0 by default
    if (i.pushValue(v1)) // Placing the result on the stack
            return true;
    return false;
}
```

For functions in the test_module.cpp file, you need to create a separate symbol table:

```
static const uint8_t tmCommandTokens[] PROGMEM = {
    'M','U','L','T','P','R','I','N','T',ASCII_NUL,
    ASCII_ETX
};

static const BASIC::FunctionBlock::command tmCommandImps[] PROGMEM =
{
    TestModule::comm_multprint
#if FAST_MODULE_CALL
    , nullptr
#endif
};

static const uint8_t testModuleFuncs[] PROGMEM = {
    'H','Y','P','O','T',ASCII_NUL,
    ASCII_ETX
};

static const BASIC::FunctionBlock::function funcs[] PROGMEM = {
    TestModule::func_hypot
};
```

In the constructor, you need to add links to the symbol table in the module:

```
TestModule::TestModule()
{
    commands = tmCommandImps;
    commandTokens = tmCommandTokens;
    functions = tmFuncImps;
    functionTokens = tmFuncTokens;
}
```

After starting, the HYPOT function becomes available, taking no parameters and

returning 0:

```
PRINT HYPOT()
 0
```

It remains to add the receipt of the function parameters and add the code for calculating its result:

```
bool
TestModule::func_hypot(BASIC::Interpreter& i)
{
     BASIC::Parser::Value v1, v2;
     if (i.popValue(v2)) {
          if (i.popValue(v1)) {
               const BASIC::Real rv1 = BASIC::Real(v1);
               const BASIC::Real rv2 = BASIC::Real(v2);
               v1 = sqrt(rv1*rv1 + rv2*rv2);
               if (i.pushValue(v1))
                    return true;
          }
     }
     return true;
}
```