TiCodEd

&

Structured Extended Basic

by Stefan 'SteveB' Bauch

Version 1.0 (c)2020

ticoded <at> lizardware.de

Table of Content

What is TiCodEd ?

When I discovered the fabulous ISABELLA Extended Basic Compiler I finally found a way to complete my TI-99/4a game programs I had abandoned for quite some years, because the interpreted basic was so slow.

I was disappointed that working with an emulator still meant using the limited edit functions of the TI. I discovered TIdBiT, a PHP program to get rid of the line numbers in Extended Basic and replace them with labels. So I started with an PC text editor, pasted the text to TIdBiT, converted the program there to standard Extended Basic and pasted it to the Classic99 emulator.

Some say, laziness is the mother of all inventions. I wanted to have a simple way of writing code in a modern environment and test it in an emulator.

TiCodEd (say it like 'decoded', only with a 't' in the beginning: 'TEE-coded') is the TI Code Editor for this. It offers:

- Modern Basic without line numbers
- Translation to Standard Extended Basic
- Saving of files in tokenized format in FIAD

Once you wrote your code, just click Export/Build Project, switch to the emulator, OLD DSK1.YourProg and RUN …

Your program is working perfect? TiCodEd can also write the program in MERGE format for direct use in ISABELLA to be compiled.

What is Structured Extended Basic?

I found TIdBiT quite useful and ported it to Pascal, to be used in the free Lazarus IDE. I learned a lot by doing so, but one finding was, that this was not suitable to be extended for structured elements I wanted to use in my programs:

- REPEAT …UNTIL
- WHILE …ENDWHILE

Those two forms of loops, head controlled WHILE and tail controlled REPEAT are making GOTO redundant. Labels are supported for all statements where line numbers are used in Standard Extended Basic, i.e. GOTO, GOSUB, RESTORE.

You can now write programs like

```
Data "This is a Test ", Does it work? , "End"
repeat
  read a$
  print a$
until a$="End"
end
```

TiCodEd translates this to

```
100 Data "This is a Test ", Does it work? , "End"
110 read a$
120 print a$
130 IF NOT (a$="End") THEN GOTO 110
140 end
```

which can be run with the regular Extended Basic Module.

Installing TiCodEd

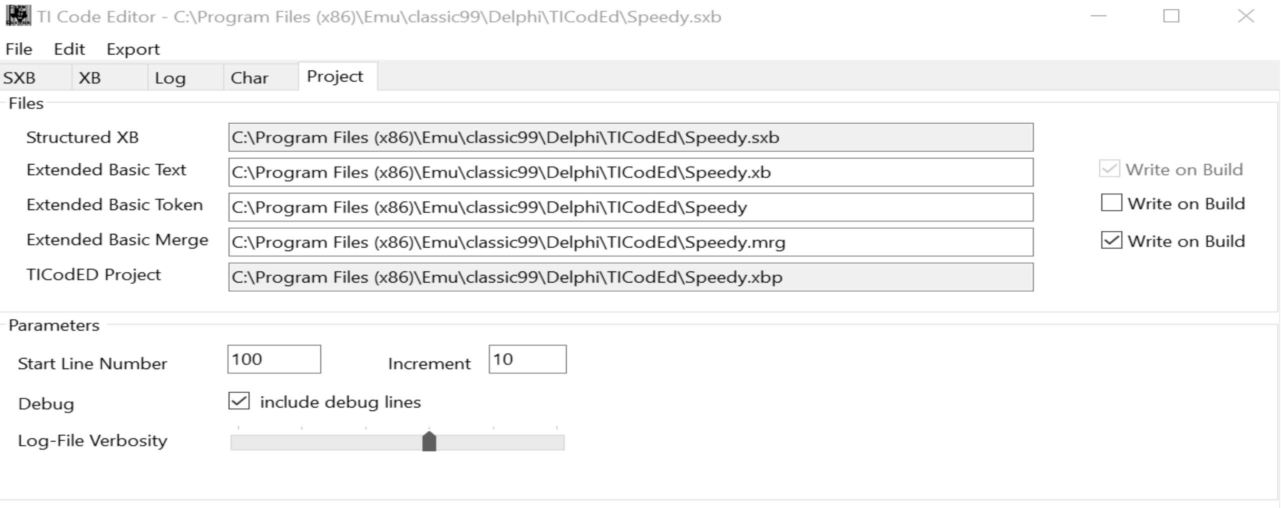
Just download the ZIP file and unpack it to a folder of your choice. As you are reading this documentation you figured this out for yourself, didn't you?

You may associate the SXB extension (Structured Extended Basic) with TiCodEd by double-clicking an SXB-File, search for the TiCodEd.exe file and select always to use this application.

TiCodEd Projects

When you open an SXB file or create a new one, TiCodEd automatically creates project file in the same directory with the extension XBP (Extended Basic Project).

The project file contains the options for your project and is maintained on the Project page.


TI Code Editor - C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy.sxb
File
Edit
Export
SXB
XB
Log
Char
Project
Files
Parameters

Structured XB: C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy.sxb

Extended Basic Text: C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy.xb ☑ Write on Build

Extended Basic Token: C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy ☐ Write on Build

Extended Basic Merge: C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy.mrg ☑ Write on Build

TICodED Project: C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy.xbp

Start Line Number: 100 Increment: 10

Debug: ☑ include debug lines

Log-File Verbosity

Currently the following options are available:

- Which Standard Extended Basic Text file will be written. ('Write on Build' is always checked as this file is mandatory)
- Which Standard Extended Basic Token file will be written (i.e. to a FIAD directory) and this should be written when the 'Build Project' or 'Tokenize' are selected in the Export menu.
- Which Standard Extended Basic Token MERGE file will be written (i.e. to a FIAD directory) and this should be written when the 'Build Project' or 'Tokenize' are selected in the Export menu. Very useful when you plan to compile the program using ISABELLA.
- Start Line Number and Increment when creating the Standard Extended Basic program from the Structured Extended Basic program, which usually has no line numbers as there are not needed.
  Note: Scattered line numbers in the SXB file may be used to create logical blocks, see Page 8, Using Labels and Line-Numbers.
- Debug: Lines starting with a hash # will be included for debugging when box is checked, otherwise they will be commented //.
- Log-File Verbosity configures how many messages written to the Log page

  - Error - Only Errors are shown
  - Warning - Errors and Warnings are shown
  - Information - Errors, Warnings and Information are shown
  - Verbose - Also included debug information
  - Very Verbose - More Debug information
  - Unbelievable Verbose - Debug down to the bits

Place the mouse over the scale to get a pop-up of the selected level.

Additional parameters are planned.

Writing SXB Code

The SXB page is the actual editor. It features syntax highlighting, but currently a generic 'Visual Basic' rule set, which fits most, but not all language features. Especially the # sign is used differently in Extended Basic (Sprites, Files).

A dedicated Extended Basic Highlighter for the SynEdit component of Lazarus is currently not my priority.

Use File/Open to load the demo SPEEDY.SXB, a very simple game to demonstrate the main concepts of SXB.

For an introduction to Structured Extended Basic see chapter XXX.

Exporting SXB Code

Once you finished you code, or at least reached a state you want to test, you can click Export/Build Project or start the steps separately, first 'SXB to XB' and then 'Tokenize'. Export/TIdBiT starts the port of TIdBit to create the XB file.

Depending on your project settings a tokenized and/or tokenized MERGE file will be written.

If you mount for example DSK4 in Classic99 to your can just type OLD DSK4.<program> and RUN it.

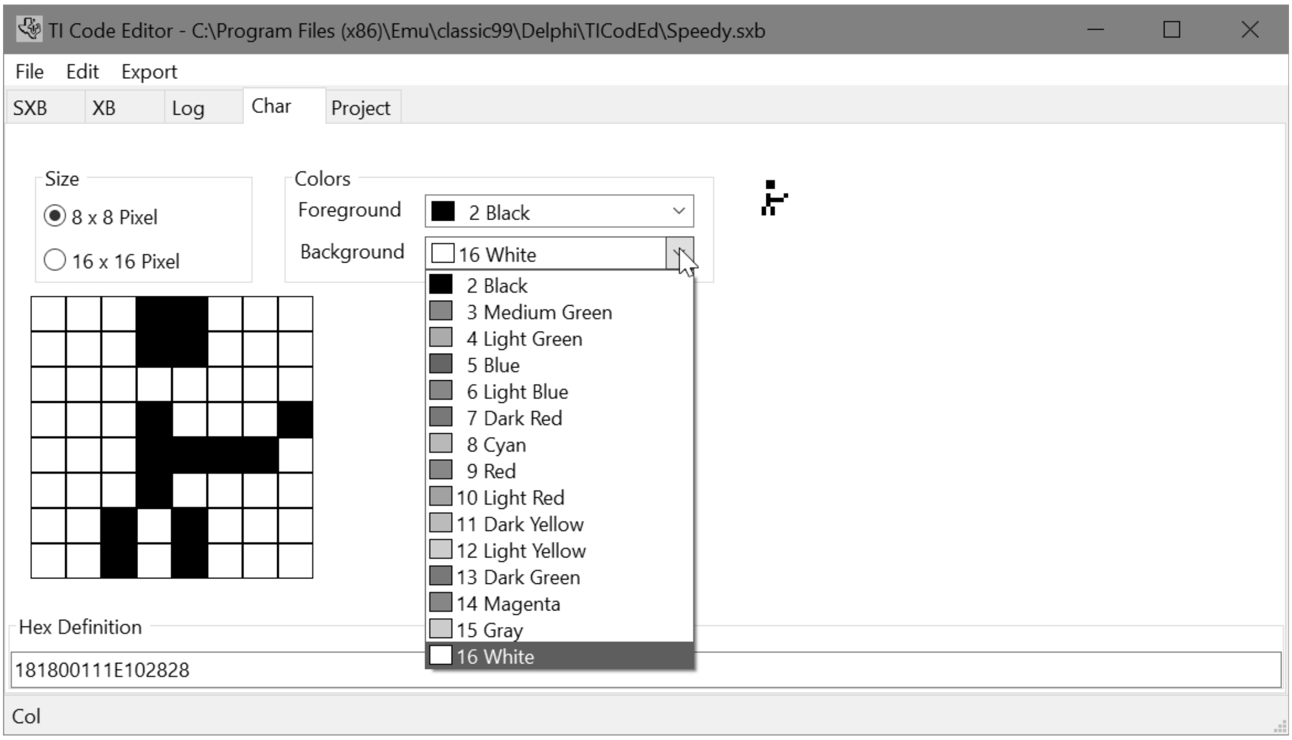
You will find the log of the conversion on the Log page and the Standard Extended Basic file on the XB page, which is read-only by default, but becomes editable double-clicking the text. This might be useful when you want to make a small adjustment and then click Export/Tokenize to create the TI files. The case will be adjusted in the tokenization and remains intact in the XB page.

The Log can be cleared or saved to a file by a left button click on the log text for the context menu.

Please note that TI Files resulting from the export are binary files with the TIFILES header used by most emulators and may get corrupted when opening with a text editor, but may be viewed or changed with a Hex-Editor like XVI32.

Char definition

One common task in game development is the definition of the graphics. The Char page offers a little tool to help with this. You can choose between 8x8 pixel and a 16x16 character definition for single characters or Magnify 3 or 4 groups of four characters. In order to support the design you can choose the fore- and background color and see a small preview.



Size

● 8 x 8 Pixel

○ 16 x 16 Pixel

Foreground: 2 Black

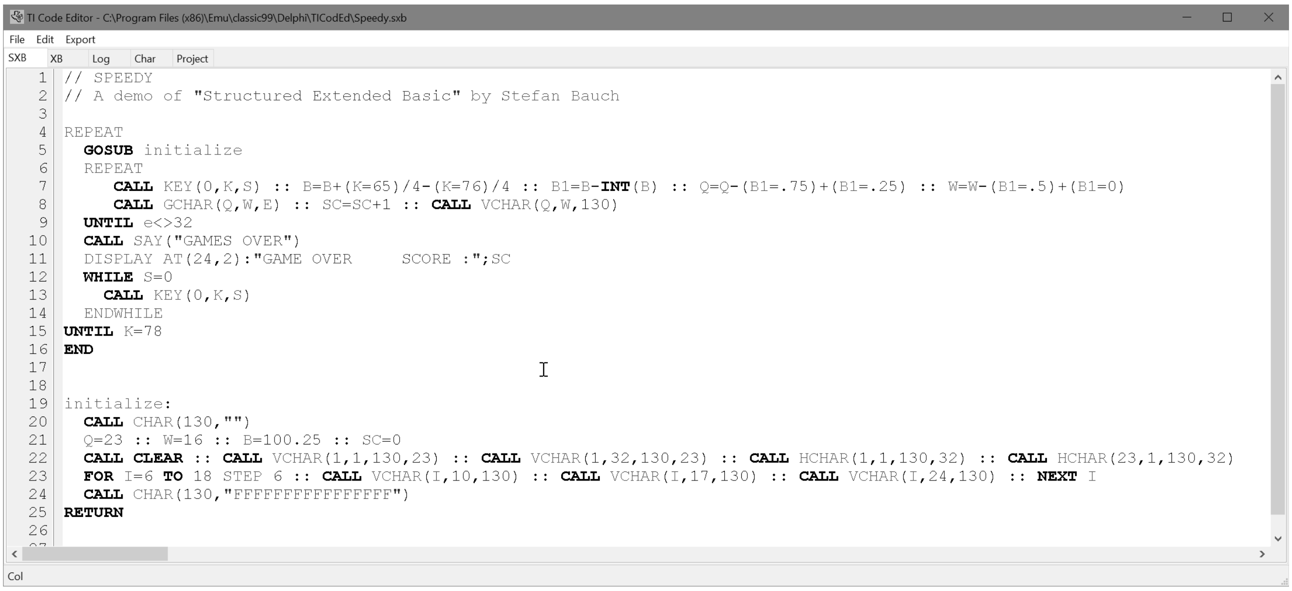
Background: 16 White

Hex Definition: 181800111E102828

The resulting definition string will be updated with every click you make and can be copied with Ctrl-C to your program. The string is also editable, you can change it any time or paste a hex-string. Illegal, non-hexadecimal characters will be erased when you change focus to another element and leave the edit field.

Structured Extended Basic

Let's start with a small included demo game called „Speedy“.


TI Code Editor - C:\Program Files (x86)\Emu\classic99\Delphi\TICodEd\Speedy.sxb
File
Edit
Export
SXB
XB
Log
Char
Project
Col

```
 1 // SPEEDY
 2 // A demo of "Structured Extended Basic" by Stefan Bauch
 3
 4 REPEAT
 5   GOSUB initialize
 6   REPEAT
 7      CALL KEY(0,K,S) :: B=B+(K=65)/4-(K=76)/4 :: B1=B-INT(B) :: Q=Q-(B1=.75)+(B1=.25) :: W=W-(B1=.5)+(B1=0)
 8      CALL GCHAR(Q,W,E) :: SC=SC+1 :: CALL VCHAR(Q,W,130)
 9   UNTIL e<>32
10   CALL SAY("GAMES OVER")
11   DISPLAY AT(24,2):"GAME OVER     SCORE :";SC
12   WHILE S=0
13     CALL KEY(0,K,S)
14   ENDWHILE
15 UNTIL K=78
16 END
17
18
19 initialize:
20   CALL CHAR(130,"")
21   Q=23 :: W=16 :: B=100.25 :: SC=0
22   CALL CLEAR :: CALL VCHAR(1,1,130,23) :: CALL VCHAR(1,32,130,23) :: CALL HCHAR(1,1,130,32) :: CALL HCHAR(23,1,130,32)
23   FOR I=6 TO 18 STEP 6 :: CALL VCHAR(I,10,130) :: CALL VCHAR(I,17,130) :: CALL VCHAR(I,24,130) :: NEXT I
24   CALL CHAR(130,"FFFFFFFFFFFFFFFF")
25 RETURN
26
```

You can see the program structure just by looking at the optional indention. In the first lines you see comments. Everything after the // is ignored to the end of the line and not included in the exported XB file. REM or ! may be used to have persistent comments in XB.

A leading # indicates a Debug-Line. Depending on the checkbox *Debug* on the project page the line gets included (checked) or dismissed (unchecked). This way you can include additional functions during development and just switch them off when you build a release version.

Blank lines will be ignored and may be used for grouping of lines.

The program contains labels (in line 19, used in line 5) and structured loops, REPEAT .. UNTIL and WHILE .. ENDWHILE, discussed in detail in the next two paragraphs. These loops may be nested like FOR..TO..NEXT.

For Structured Extended Basic it is mandatory to have these statements not combined with other statements on the same line, as it is common practice for a better readability of the source code.

[for playing Speedy use A key to turn left, L key to turn right, N key for end.]

Using REPEAT .. UNTIL

REPEAT .. UNTIL is a foot-controlled loop. This means, the code is executed at least once. The condition at the end of the block determines whether to repeat the section starting with REPEAT or leave it.

The loop will be translated in two steps as follows:

```
REPEAT
  <CODE>
UNTIL <CONDITION>
```

Step #1: Create Labels

```
REPEAT001:
<CODE>
IF NOT (CONDITION) THEN  REPEAT001
```

Step #2: Assign Line-Numbers

```
100 <CODE>
110 IF NOT (CONDITION) THEN  100
```

You may use any valid XB code and any valid XB condition. Check the SPEEDY.SXB example and compare the SXB with the XB page.

Using WHILE .. ENDWHILE

WHILE .. ENDWHILE is a head-controlled loop. This means, the code may be omitted completely if the condition is not met.

The loop will be translated in two steps as follows:

```
WHILE <CONDITION>
  <CODE>
ENDWHILE
```

Step #1: Create Labels

```
WHILE001: IF NOT (CONDITION) THEN ENDWHILE001
<CODE>
GOTO WHILE001
ENDWHILE001:
```

Step #2: Assign Line-Numbers

```
100 IF NOT (CONDITION) THEN 130
110 <CODE>
120 GOTO 100
130 ...
```

This does not look as elegant as

```
IF NOT <CONDITION> THEN ENDWHILE001
WHILE001:
<CODE>
IF <CONDITION> THEN WHILE001
ENDWHILE001:
```

but is better readable in nested WHILE loops.

As many BASIC variants use WHILE .. WEND it is also supported to use WEND instead of ENDWHILE.

Using Labels and Line-Numbers

You may use Labels in your program anywhere in your Basic program where line-numbers are used in Standard Extended Basic, i.e. GOTO, GOSUB, RESTORE etc.

Labels are defined by a name, followed by a colon in the first column of your program. Please make sure not to use any reserved words or use a label more than once.

The label is referenced without the colon.

Example:

```
GOSUB Initialize
END
Initialize:
  <code>
RETURN
```

You can always use line numbers as in Extended Basic. If the internal line-number is lower than the explicit line-number, the internal number will be adjusted, otherwise the internal number will be used. This makes sure that line-numbers are in ascending order, but may break existing GOTO/GOSUB/etc. statements using the explicit number.

It is generally depreciated to use line-numbers. For a better readability of the generated Extended Basic source it is suggested to use scattered line-numbers for blocks of code to be found again easily. Do so by only entering a line-number on a line of its own or in front of a statement. This will offset the line-number, but keep the increment, i.e. GOSUB routines start at line 10000, DATA statements at 20000 and SUB routines at 30000.

Implicit Code-Blocks

Extended Basic has neither BEGIN/END like Pascal nor { } like C to build blocks of statements. But you may group some few statements in an IF-THEN-ELSE clause by using :: for multiple statements in one line, i.e.:

```
IF A<0 THEN <STMT1> :: <STMT2> else <STMT3> :: <STMT4>
```

This technique is limited by the allowed length of the line in Extended Basic.

It can be used in Structured Extended Basic as well and is the reason that in contrast to other programming languages the line has a meaning and may not be split without changing semantics. Each line with executable code in Structured Extended Basic will be one line in the resulting Standard Extended Basic code.

Editor Features

Lazarus offers the SynEdit component, which is used on the first page of TiCodEd. Beside the syntax-highlighting it offers some useful shortcuts beside the usual navigation:

Undo	Alt Backspace
Redo	Shift Alt Backspace
Block Indent	Shift Control I
Block Unindent	Shift Control U
Column Select	Shift Control C
Line Select	Shift Control L
Normal Select	Shift Control N

Porting existing XB code to SXB

There is no general approach, but the following worked quite good for me:

Go through the code and look for any reference to a line-number. Replace the line-number with a label and add this label right in front of the target line.

Once finished you may remove all your line-numbers and start formatting the code. Add blank lines to separate logical blocks of code. Use line-numbers to group your code. Add // comments to make the code easier to understand. Refactor blocks by using REPEAT and WHILE loops instead of IF/GOTO.

Pretty soon your code looks completely different. Export the code to XB and compare to the original code.

Limitation

There are some design limitations you should be aware of, but usually should not limit your options:

- Nesting level of REPEAT/WHILE has combined a maximum of 32
- Not more than 999 REPEAT/WHILE loops may be used in one program
- Labels may have up to 32 characters
- You may use up to 100 labels
- SXB Programs may have up 1024 lines
- A line may have up to 128 token (remember, special characters like plus or parenthesis are also tokens). A "Line too long" error will be shown when the limit is exceeded
- REPEAT, UNTIL, WHILE, ENDWHILE and WEND need to be the only statements on a code-line
- Tokenized files over 11775 bytes will be written in long format
- TIFILES header will only partly filled
- SXB files will be automatically saved before conversion to XB
- Syntax errors may not be detected as both conversions (to XB and to Tokens) are based on substitution of strings, not a syntax-graph
- The editor uses the Visual Basic highlighter and show some weird interpretations in SXB, especially the # sign in sprite- and file-numbers is misinterpreted. Lazarus volunteers are welcome to do a SXB highlighter
- DATA statements may have strings with quotation. Please be aware of the removal of spaces at the beginning and end
- GUI status bar is still without function

This section is likely to be extended when people send me errors I can not easily fix.

Trouble-Shooting

Something not working? Take a look at the log page. If you set the log-level to 5 = 'unbelievable verbose' you get log information about seven times the size of your source when running 'Export / Build Project'.

First, check if the code is correctly translated from SXB to XB. The log gives you indication of all substitutions.

The second step is more tricky, as it involves the binary token format. The log file also lists each line in token format. First the XB line is shown, then after the -> arrow how it is split up separated by pipe symbols | and finally the hex presentation. For bytes larger 0x80 the corresponding token is listed in square brackets for easier interpretation, disregarding whether or not it is actually a token (i.e. part of a line number).

```
120 FOR I=0 TO 14 :: CALL COLOR(I,16,2) :: NEXT I :: CALL
COLOR(9,10,16) -> FOR|I|=|0|TO|14|::|CALL|COLOR|(|I|,|16|,|2|)|::|
NEXT|I|::|CALL|COLOR|(|9|,|10|,|16|)| -> 8C[FOR] 49 BE[=]
C8[Unquotedstring] 01 30 B1[TO] C8[Unquotedstring] 02 31 34 82[::]
9D[CALL] C8[Unquotedstring] 05 43 4F 4C 4F 52 B7[(] 49 B3[,]
C8[Unquotedstring] 02 31 36 B3[,] C8[Unquotedstring] 01 32 B6[)]
82[::] 96[NEXT] 49 82[::] 9D[CALL] C8[Unquotedstring] 05 43 4F 4C 4F
52 B7[(] C8[Unquotedstring] 01 39 B3[,] C8[Unquotedstring] 02 31 30
B3[,] C8[Unquotedstring] 02 31 36 B6[)] (?I???0???14????COLOR?I???
16???2???I????COLOR???9???10???16?)
```

If the tokenized file does not load try pasting the XB source code to the emulator, save it and compare the files. Or try loading the MERGE file instead.

Found an Error? Having a suggestion? Future plans?

I am happy to hear from you, whether TiCodEd works for you or you found a problem. I won't debug your code though. When you have an example of something going wrong, chances increased when your code has less than 15 line or you can point to an exact line where an error occurs and what is actually right.

I set up an email address I check when I have time for my TI hobby:

TiCodEd <at> lizardware <dot> de

Please accept my apologies that it may take some time for me to respond. Rainy autumn weekends may be better than sunny spring workdays.

I already have some features in mind for future releases.

I never tried XB256 as I started TiCodEd to finish my old projects, where I stick to standard functionality.

A second, extended graphics page looks very tempting for new projects.

```
CALL LINK("CHAR2 ,character-code,pattern-identifier[,...])
```

looks a little bit complicated though. How about

```
CALL CHAR2(character-code,pattern-identifier[,...])
```

in Structured Extended Basic instead? Of course for all CALL LINK in XB256.

I also have some few routines I use in more than one program. I think of something like a library. When you use one of the routines from a library in a CALL statement, the SUB code is automatically appended to the program.

Let me know what you think and see what the winter brings...

Building TiCodEd from Source

TiCodEd is written in PASCAL using the free Lazarus IDE and libraries.

https://www.lazarus-ide.org/

This IDE offers cross-platform support for Linux and Mac (sadly not yet compatible with the latest MacOS). Ports to those platforms may not be too complicated. Currently only Win32 and Win64 are supported and available for download.

A new Syntax Highlighter for SXB would be nice if someone feels the need to dig into the depths of the SynEdit component. I currently prefer not to.

Use LibXBTKN32.dll or LibXBTKN64.dll

I wrote a wrapper for the uTokenize.pas unit to be used in other programs. It is used to convert Extended Basic Text-Files to tokenized XB and publishes the following function:

```
Function XBTokenize(xbin,tknout,mrgout,logfn,opt:PChar):Integer; cdecl;
```

Parameter	**Description**
xbin	Input Text filename with Standard Extended Basic Program
tknout	Output filename with TIFILES Header, empty to omit
mrgout	Output filename with TIFILES Header in MERGE format, empty to omit
logfn	Logfile name, empty to omit
opt	Options: v=[0..5] Verbosity, default 3
Return-Code	00 : No Errors 10 : Could not open XB input file 11 : Error reading XB input file 20 : Error converting file 30 : Error writing MERGE file 40 : Error writing PROGRAM file 50 : Could not open log file

BSD License

Acknowledgements

Some people helped me, for pandemic and geographical reasons mostly by mail.

- Matthew Hagerty with hints on porting his TIdBiT to Pascal
- Ralph Benzinger for hints on tokenizing and the 'long format'
- Harry Wilhelm for writing ISABELLA, which encouraged me to resume coding for the TI-99/4a
- Fred Kaal for Ti99Dir and testing of TiCodEd and the Tokenizer LibXBTKN32.DLL and including it in Ti99Dir
- Helge Spahrbier for testing of TiCodEd

In memory of Frank Euler, my late TI programming companion. I miss you.