

TEXAS INSTRUMENTS TI-99/4A

BEGINNERS MANUAL

STRUCTURED EXTENDED BASIC

&

TiCodeD

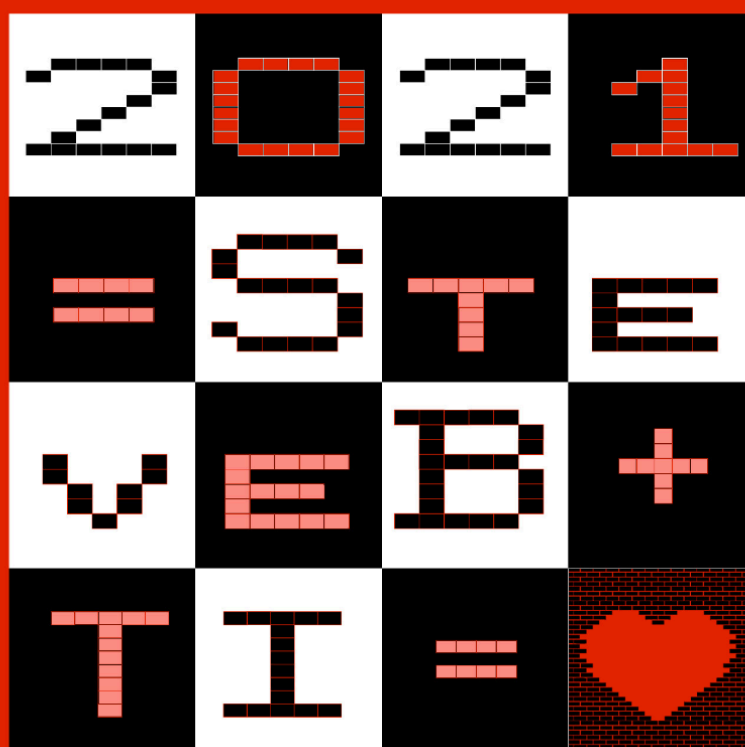


Table of Contents

Introduction	4
Background	5
Who is SteveB?	7
Me and my TI	7
How TiCodEd originated	7
Historical Note	8
Goals for TiCodEd	8
TiCodEd Prerequisites	9
Is this really that much better than in the past?	9
What is next for TiCodEd	9
Why should you use TiCodEd?	10
Helping the developer help you	11
What you need to begin	12
The Classic99 Emulator	12
The TiCodEd Editor	12
The BASIC Compiler (JEWEL, formerly ISABELLA)	12
TI99dir and the Module Creator (optional)	12
AtariAge Account (optional)	13
"Hello New TI World"	14
Setting up Classic99	14
Copy and paste	20
First use of TiCodEd	22
SXB tab - Structured Extended BASIC	22
Project Tab	25
Why Line Numbers?	31
Our first real Structured Extended BASIC Program	34
The use of Labels	37
One more "Comment"	38
Demo Game "SteveB52"	40
The structure of the program	40
The Subroutines	41
The BEGIN/END-Block	44
Get the Compiler ready!	48
Creating a Module	52
Language Reference	56
Considerations for Compiling	56
Extended BASIC Reference	56
Additional Libraries	61
The Standard Library	61
The User Library	62
Packages	62
How to start on your own	69

Start from Scratch	69
Continue an existing project	69
Advanced Topics	72
The Character Definition Tab	72
CALL SUB or GOSUB?	76
Using BEGIN-END for code blocks	77
The CASE Statement	79
IN SET Condition	81
Binary data with BIN\$	82
Embedded Assembly ASM ... ASMEND	82
Prepare the Compiler	82
Writing an ASM Routine	84
Debugging an ASM Routine	85
Assembly libraries	85
The Variable Tab	86
A daunting look at TIFILES	87
Understanding the Log	90
Analyzing the SXB file	91
REPEAT-UNTIL, WHILE-WEND and line-numbers	91
Resolving Labels	92
Dumping the Extended BASIC file	93
Prepare to tokenize	93
Atomize into Tokens	94
Program Statistics	95
Variable Cross-Reference	96
Export Files	96
Errors and Warnings	96
Advanced Project Settings	96
Post Processing Command	97
TiCodEd Preferences	98
Emulator integration and automation	100
Patching the Compiler	102
jewel7.patch	103
no-auto.patch	103
overdrive.patch	103
Include Assembler Data in HighMem	103
Library reading sequence	104
Extended BASIC 2.9 Graphics Enhancement Module	104
XB256 Library	106
RXB - Rich Extended BASIC	106
Extended BASIC 3	106
Forty Columns Text-Mode T40XB	107
Eighty Columns Text-Mode T80XB	107
The Missing Link	107
Multicolor	107
Classic99 Debug Library	107

Embedded Utilities	108
Edit Utilities	108
Advanced Editor Features	109
Hex to BIN\$	111
Edit TIFILES	113
Cart Patching	114
TiCodEd Built Notes	119
The Windows Versions	119
The Mac Version	119
The Linux Version (x64)	120

Version 2025-03 based on TiCodEd 3.0

Introduction

You are about to begin a new and exciting TI-99/4A programming adventure! This manual was written to inform beginners and experienced TI BASIC programmers alike. Here you will learn to take full advantage of some of the best BASIC programming practices which have emerged since the early days of home computing. The material we will cover includes:

- Installing Tursi's **Classic99** TI-99/4A emulator onto a modern PC. Emulators significantly increase productivity by allowing us to move the development workflow over to a modern PC. Basically leveraging forty years of technology to our advantage.

- Installing and utilizing the modern **TiCodEd** program editor for well-structured TI-99/4A BASIC programming. No more quirky TI keyboard layout to type on while programming. No more ancient/awkward line-by-line editing environment. Perhaps most importantly, moving the programming editor over to a modern PC turns your development environment into a mobile platform. Have you ever tried programming the actual TI-99/4A on a flight to Zurich?

- Compiling BASIC code to realize 20x to 50x speed efficiencies over TI's historically slow Extended BASIC programming language. Harry Wilhelm's excellent **BASIC compiler and XB256 Extended BASIC** extensions have really opened-up the TI-99/4A's power to BASIC programmers. This manual will guide you down the pathway toward producing code which runs far faster than any BASIC you've experienced in the past.

- Installing two of Fred Kaal's programs, **TI99dir** and **Module Creator**, to create "ROM files," which run on emulators as modules and on the the real hardware using the latest TI-99/4A SD-cartridge enhancers such as Ralph Benzinger's must-have **FinalGROM**. They'll also run on traditional TI floppy drives as well as Ralph's older FlashROM99.

Even if you have never worked with a modern code-editor or a classic computer emulator before, this manual will get you up to speed with some of the very best productivity enhancing development practices. Soon you may find yourself teaching others, perhaps family and friends, and enjoying the TI-99/4A retro community online via Atariage.

Who helped me? This manual was inspired by a [posting](#) of Odd "Oddemann" Kristensen on AtariAge. He supplied the great cover page and in a two-hour video-call we discussed what was needed to get into TiCodEd and SXB. James "Airshack" Shackles joined us and ironed out my german-english grammar and added lots of useful pieces.

During the initial development Fred Kaal was extremely helpful.

FACTOID: The BASIC Compiler is a software tool which performs analysis on your BASIC program, retrieves appropriate routines from a library, manages storage allocation, and then creates an actual Assembly Language source program out of your original BASIC program. This Assembly version of your program is then “assembled” to yield a machine language file which will run much faster than the BASIC or Extended BASIC version.

Background

Programming in BASIC became extremely popular in the early nineteen-eighties, in-part because most home computers shipped with built-in BASIC interpreters. From the mid-seventies and throughout the nineteen-eighties, many new and innovative home computer systems were released to the consumer marketplace. With each new system came a slew of incompatible yet often exciting feature sets. It was a fast ride, like a binary gold-rush, with high stakes, and a wild west market.

IBM hastily entered the market in 1981 with its own Personal Computer. At first glance this graphically underwhelming model -5150 PC- appeared to be just another overpriced underperformer. Yet...by the turn of the decade the “PC’s” open architecture matured to the point where it eventually became the industry de facto standard. The arrival of the IBM PC meant the pioneering Home Computer Era was basically over. A digital death nail...mostly.

Alas! As with vintage cars, there has been a renaissance based on nostalgia for the earliest of early home computers. Retro-gaming itself, as well, has become so popular that it has its own exhibition hall at the annual Gamescon fair in Cologne, Germany. Ever growing vintage computer gatherings take place every year on all of the populated continents of the world.

Many imaginative hobbyists have created new hardware enhancements for the most popular of the classic machines. Emulation of most classic computers and gaming consoles can be found on modern Windows, Mac, Linux computers. There’s even some web-based emulation. The 40+ year-old TI-99/4A computer has seen new and improved versions of Extended BASIC emerge as well. Example: **Rich Extended BASIC (RXB)**. Also, numerous compilers and interpreters for various programming languages are now available for the classic computers.

Do you remember programming your TI-99/4A during the eighties?

Do you wish TI BASIC or Extended BASIC ran faster?

Do you simply wish to program with an improved keyboard feel?

Will you prefer using a full-screen editor with syntax highlighting?

Do you prefer a well-structured BASIC language with fast running code?

Fortunately for us the TI BASIC programming experience has improved dramatically over forty years! You do not need to climb up into the attic to start your TI-99/4A programming adventure. A Windows PC is sufficient, but we will also cover working with real TI-99/4A metal, as nothing compares to seeing your programs running on original hardware, if you have it, somewhere waiting to be reactivated and reunited.



The TI-99/4A



Who is SteveB?

Let me briefly introduce myself. My real name is Stefan Bauch, I reside in Germany and make a living out of being a computer scientist. For which the TI-99/4a is not completely innocent.

Me and my TI

I was 14 when I bought my TI, in September 1983, with joysticks and a cassette-recorder interface. I had a tiny "Minerva" recorder and no money for a PEB. In the spring of 1984, I obtained an Extended BASIC Module, a speech synthesizer, and a German and an English Extended BASIC book. That was my modest setup. While my friends enjoyed a large number of commercial programs on their Commodore 64s and Sinclair Spectrums, I had no TI Solid State Modules to play. This turned out to be a blessing in disguise. While I joined my friends in playing commercial games on their machines, I had to sit down to write my own games for the TI. Some were nice but the more ambitious titles were never completed. TI Extended BASIC was simply too slow for the games to be fun.

So my abandoned game creations slept in Cinderella's Castle, in my parent's attic, for many years, until Harry Wilhelm wrote his exciting Extended BASIC Compiler. This event motivated me to climb up into the attic, search for my TI-99/4A and old tapes, then install **Tape994a** to see what was still alive. Harry's BASIC compiler was my dream come true!

How TiCodEd originated

I have been playing around with classic computer emulators since the early nineties. It has been amazing for me to re-discover my old TI projects from earlier times. I noticed a few of my programs were abandoned due to speed issues. With Harry's BASIC compiler I felt I finally had the tool I needed to make TI Extended BASIC programs run swiftly! And with the CoViD-19 pandemic's stay-at-home quarantines, I had plenty of free time...

My enthusiasm for my TI reunion quickly faded when I began editing my programs using TI-99/4A line-by-line editor. The TI-99/4a's command line user interface has been set in stone for decades because the operating system resides permanently in ROM. Of course all classic computers of the era predate the concept of periodic operating system updates. Line-by-line editing + dealing with a non-standard TI computer keyboard = not enjoyable. I was motivated to find a way to apply my 30+ years of IT experience to the hobby of classic computing. I wanted to make TI-99/4A programming more enjoyable.

My initial approach to the challenge was to check out what others had already done to improve the process. I developed a workflow for creating fast running BASIC programs on the TI using existing tools. My system looked something like this:

- A. Extract a program to text-files via Fred Kaal's **TI99dir**
- B. Modify the program with an editor tool (such as **Notepad**) on my PC
- C. Cut and Paste code into the **Classic99** TI-99/4A emulator
- D. Test my program in emulation on the PC within **Classic99**
- E. GOTO B.

In time I happily came upon a programming tool called **TIdBiT**, by Matthew Hagerty. TIdBiT is a PHP code translation tool which frees the BASIC programmer from using line-numbers. TIdBiT uses labels which allows program subroutines to become portable and easily relocatable.

Unfortunately, my "new" BASIC programming workflow was quite complex. This inspired me to build an one-in-all tool for TI development which leverages the power of modern PCs. Thus, I created **TiCodEd**.

Historical Note

The BASIC (Beginner's All-purpose Symbolic Instruction Code) programming language was presented free to the world on May 1, 1964, at Dartmouth University, by two Professors named John Kemeny and Thomas Kurtz. By the mid-1980s BASIC became the most widely used computer language in the world. By 1987 Kemeny and Kurtz had refined BASIC to include a range of what was known as "structured constructs." The result was called **True BASIC** which is easier to write, easier to debug, and superior to the traditional variants of BASIC in that program subroutines (modules) are easily reusable, from project to project. **True BASIC** replaces line numbers with labels and adds DO-LOOPS, etc. Of course TI's BASIC and Extended BASIC predate **True BASIC**, therefore they cannot take advantage of these soon to be explained well-structured improvements.

Goals for TiCodEd

TiCodEd is designed to support the whole creation workflow in BASIC -- especially games programming! With TiCodEd you get:

- A comfy editor with syntax highlighting
- Additional structured language constructs not common in the eighties
- Full compatibility with the real iron and TI Extended BASIC
- Easy to use character and sprite editor
- Easy integration to Classic99 and Harry's BASIC compiler

TICodEd Prerequisites

- Having used TI BASIC and/or TI Extended BASIC
- Know how to download and install programs on your PC
- A general understanding of how to use a computer
- An affinity for retro-computing and a love for the TI-99/4a

This manual covers the Windows environment for TiCodEd, which is also available for Linux and MacOS. As there is no File-in-a-Directory capable emulator on these platforms yet, automated integration does not work. Some users use WINE on these platforms.

Is this really that much better than in the past?

Just have a look at some compiled Extended BASIC games created with TiCodEd:

- [Extended Parsec](#) [Video]
- [T.E.R.O.](#) - Texas Emergency Rescue Operations [Video]
- [ScuttleButt/4a](#) – Behind Enema Lines
- [Castle Wolfenstein](#) [Video]

Would this have been possible back in the days? Perhaps with the Editor/Assembler module, but not with the comfort of a structured high-level language.

What is next for TiCodEd

My immediate plans are completed now with version 3.5. I look forward to the day where users of TiCodEd will regularly suggest enhancements for me to address. Perhaps AtariAge users already have ideas for improvement?

TICodEd author **Stefan Bauch**'s email: TiCodEd@lizardware.de

Please feel free to ask me any questions you may have via email. Since this is just a hobbyist endeavor for me it may take some time to answer your questions. Thank you in advance for your patience.

You may find answers to your questions much faster via the TiCodEd thread on AtariAge:

<https://atariage.com/forums/topic/314536-introducing-structured-extended-basic-and-ticoded/>

Why should you use TiCodEd?

It doesn't matter if this is your first look into the TI-99/4A, returning after a hiatus of many 99/4A-less years, or a die-hard TI-99/4A fanboy. This guide will take you from the creation of a simple BASIC program, to the creation of your own "modules" to be used in an emulator. Additionally, you'll even learn to run your programs on your actual TI-99/4A console with the addition of special writable hardware.

If you are an experienced TI programmer you already have your own established preferences while working with the TI. You may be interested in more details in order to decide whether you should spend the time to learn the TiCodEd approach. The most compelling arguments are:

- Modern Structured Extended BASIC without line numbers - No more bothering with renumbering to make space for new lines. Use easy to remember labels instead. TiCodEd will create the line numbers for you in the background.
- Translation to Standard Extended BASIC - You are not locked in some proprietary tool no one else uses. Even if you abandon TiCodEd and the Structured Extended BASIC, your code runs and is maintainable in the standard environment.
- Indent your code to visualize the program - Use REPEAT-UNTIL and WHILE-WEND loops to maintain a clear structure.
- IF-THEN-ELSE can be used with BEGIN-END blocks for better structured programs.
- Simply one ZIP file to unpack - No run-time or anything else required. Unzip and go, no Java, Python, PHP or any additional complications.
- Saving of files in tokenized format in FIAD¹ ("Files In A Directory"). Store the program for immediate use by the TI or **Classic99** emulator.
- Fast turn-around times for development cycles - Build a Project in TiCodEd by pressing Ctrl-B, and your program gets translated to Standard XB, transferred to Classic99 and even started. No laborious file conversions necessary. It can even fully automate the compilation.
- Export your program in MERGE format used by the Extended BASIC Compiler.
- Make full use of newer versions of Extended BASIC like XB 2.9 or RXB.
- Simplified use of libraries, like the XB256, T40XB or The Missing Link Hi-Res graphics.
- Use long self-explaining variable names without wasting precious memory on the TI.

¹ Will be explained later ... like all the other Tec-Talk in this list.

- Use the additional subroutines of the SXB standard library.
- Build your own library of reusable subroutines.
- Even embed assembly language in your compiled XB programs.
- Create cartridges with additional ROM pages containing sound, graphics or other data.
- Use the latest Google Gemini or Gemma models to help you code from within TiCodEd
- Use this excellent manual to master all that is possible today on the TI-99/4a without leaving Extended BASIC.

Helping the developer help you

Can you think of anything missing from TiCodEd? Anything you expect to find in a BASIC development tool such as this? Please tell me! I will see what I can do to improve upon TiCodEd. Use the email above or ask in Atariage.

"SteveB has made a tool that I did not know I needed. Now that I have this tool, I can't imagine NOT using it!"

Odd Kristensen

"Das ist gut!"

Airshack

What you need to begin

To get things up and running we must first install several programs onto a windows PC. Thanks to the efforts of many brilliant hobbyists we have many wonderful programming tools at our disposal. You are encouraged to have a look at all of the alternative options out there. Depending on the security setting of your Windows installation you may need to confirm that these hobby-programs are not digitally signed and click "Install anyway". The examples in this guide require:

The Classic99 Emulator

Download Tursi's Emulator at <http://harmlesslion.com/software/classic99>

Once installed, you have a complete environment, including ROM and some modules. You'll find the Extended BASIC Module under Cartridge/Apps. We will guide you step by step in the next chapter.

The TiCodEd Editor

Download SteveB's specialized editor at <http://www.lizardware.de/>

This editor features syntax highlighting for BASIC, a functionality to save programs in the TI internal format and a language extension, called "Structured Extended BASIC". This revised manual is based on Version 3.0 of TiCodEd.

The BASIC Compiler (JEWEL, formerly ISABELLA)

Senior_Falcon has written an Extended BASIC Compiler which is already included in the Classic99 installation under Contributors/Harry_Wilhelm.

You may update this compiler to the latest release of JEWEL. It can be found on AtariAge:

- [TI-99/4A development resources - TI-99/4A Development - AtariAge Forums](#)
- [XB Game Developers Package](#)

TI99dir and the Module Creator (optional)

We will also cover the creation of a module - at least the content of it, to be used in an emulator or a writable cartridge like FlashROM 99 or FinalGROM. You will need two tools from [Fred Kaal](#):

<http://hexbus.com/ti99geek/Projects/ti99dir/ti99dir.html>

http://hexbus.com/ti99geek/Modules/modcreate/modcreate.html#module_creator

AtariAge Account (optional)

The website AtariAge hosts two very popular forums for the TI-99/4a:

- <https://atariage.com/forums/forum/119-ti-994a-development/>
- <https://atariage.com/forums/forum/164-ti-994a-computers/>

The authors of the programs mentioned above frequently participate in several AtariAge TI-99/4A forums. You may read as a guest, but a free registration is required to post messages.

"Hello New TI World"

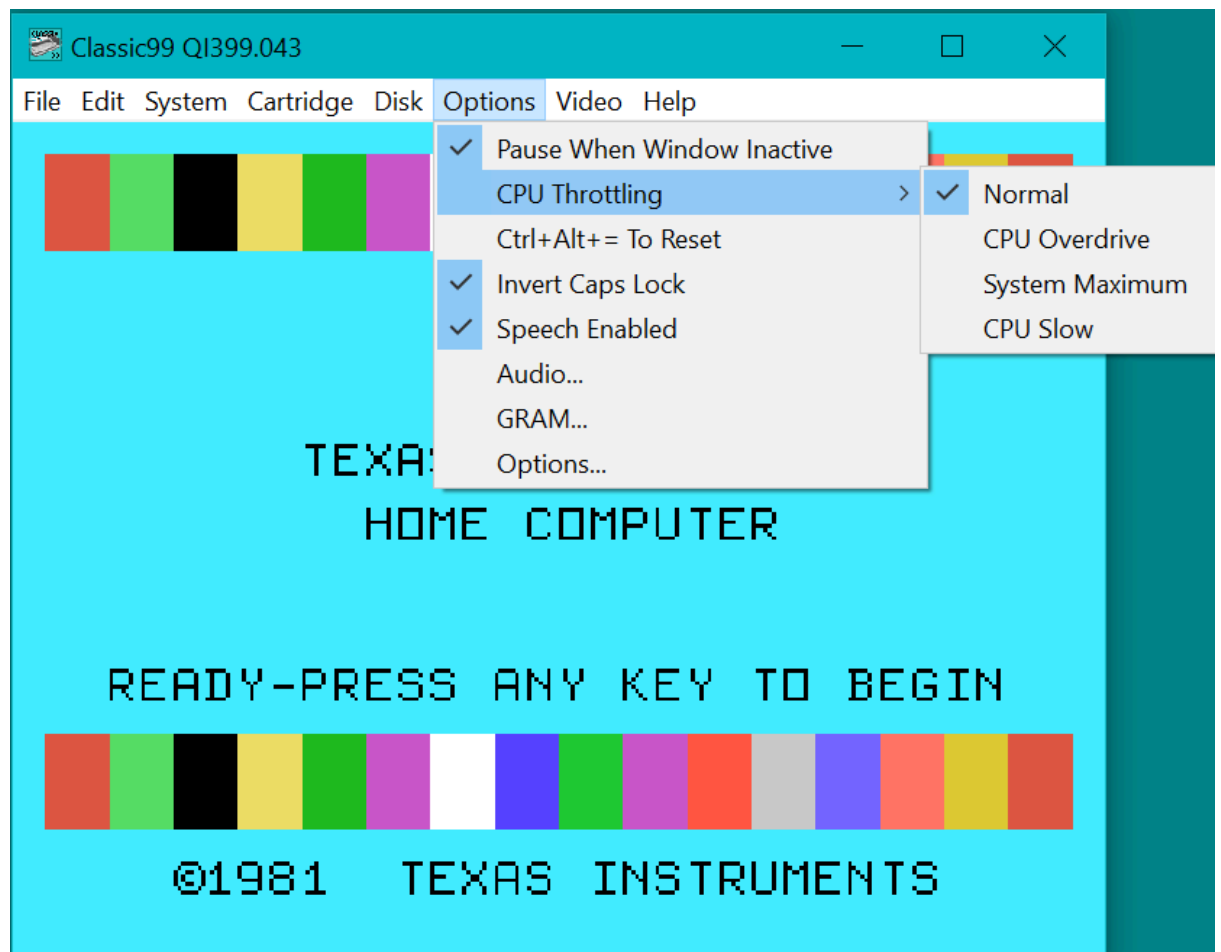
Setting up Classic99

We do not wish to duplicate Tursi's fine Classic99 manual here. Only the few specific details we need for our process are covered. To begin you must install and then launch Classic99 using the following settings:

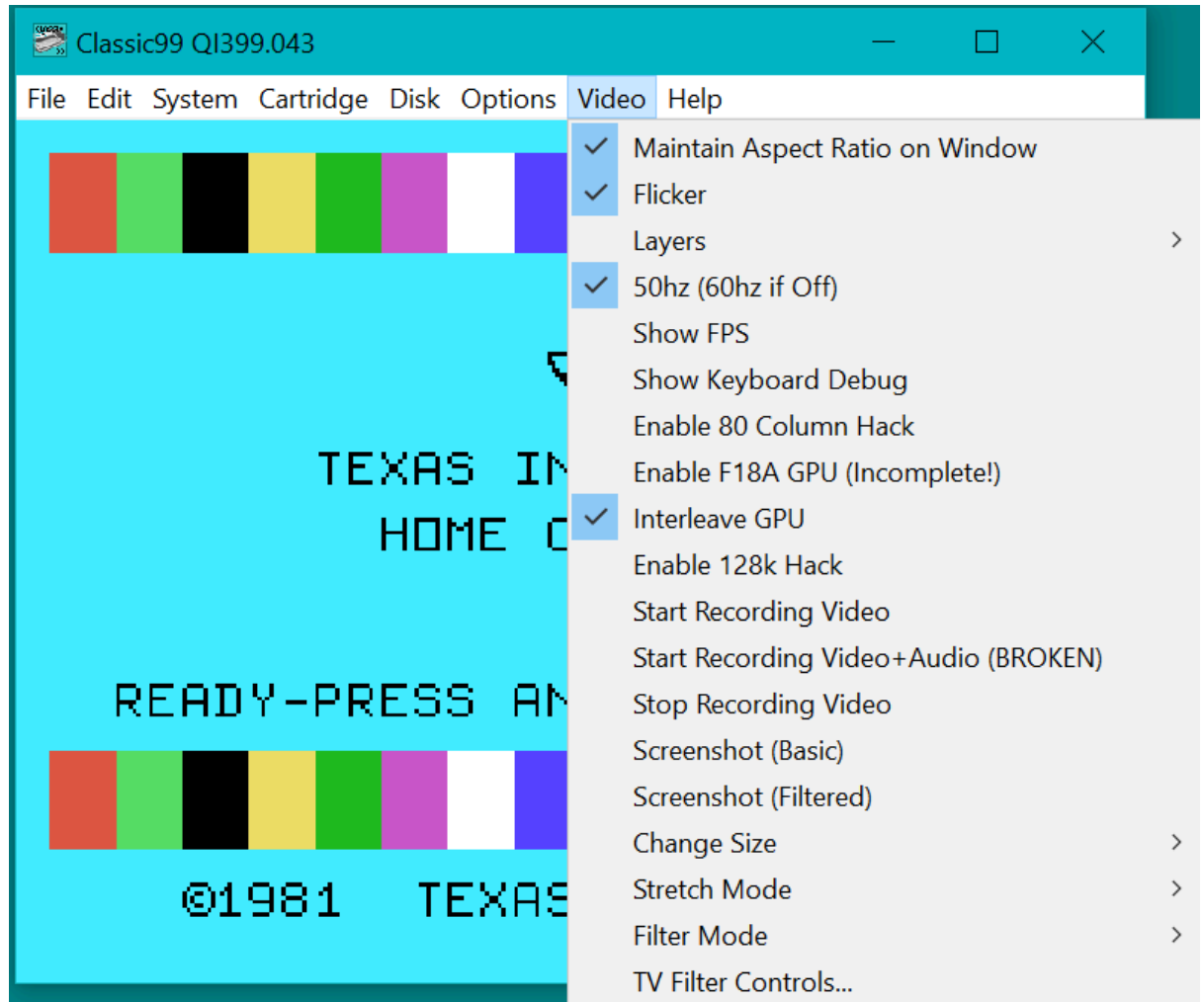
System: Select TI-99/4A

Cartridge: Select Apps/Extended BASIC

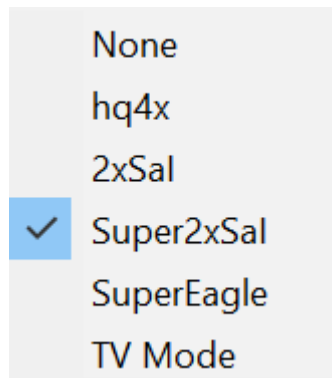
Options:



Video: **60hz** selected for NTSC (US) emulation; **50hz** for PAL (European)

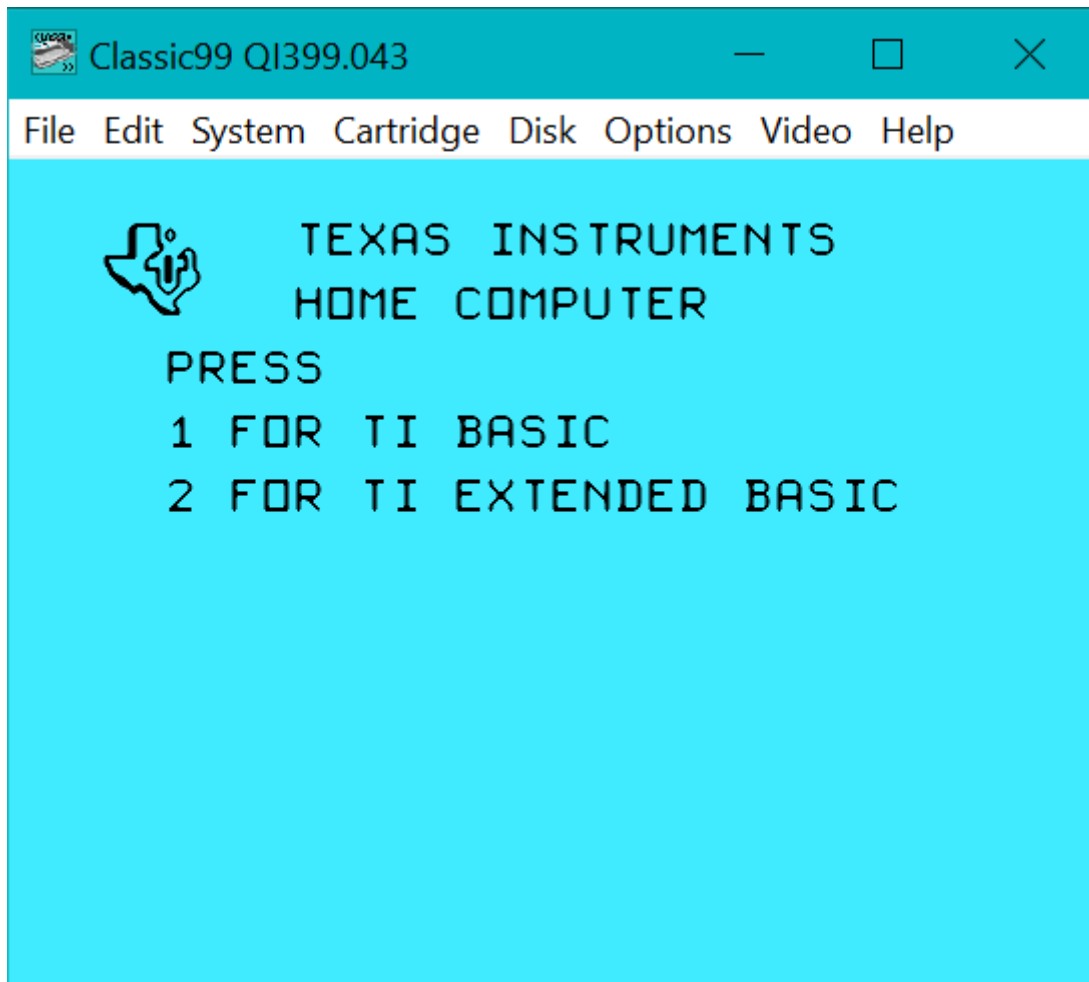


Use "Change Size" to adjust the size of the window to your needs and "Filter Mode" for a setting that appeals most to viewing habits.



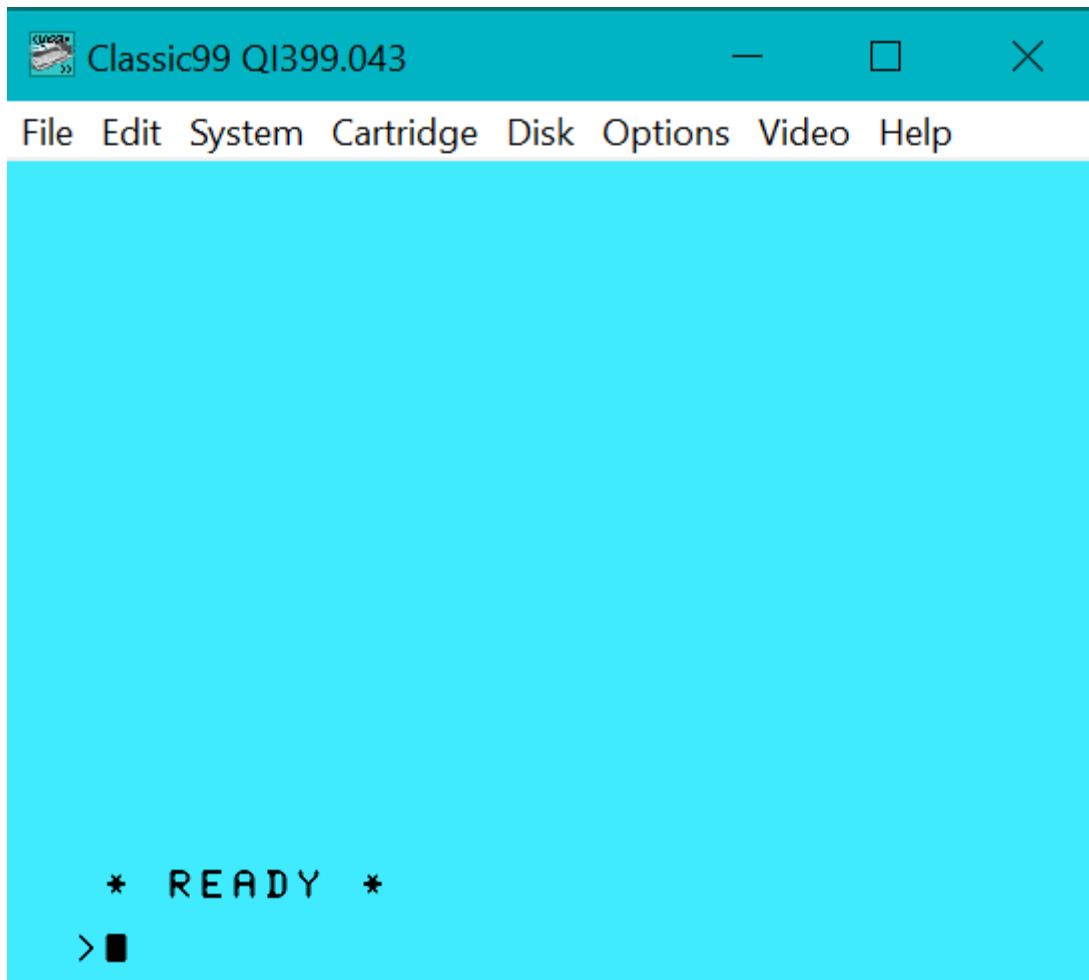
"None" gives you a Pixel-Block view (like above), TV Mode emulates the blurry picture of yesteryear. Everything in between tries to use the higher resolution in a more comfortable way (like below).

Now we're ready to start with the emulation. Launch Classic99 and click on the Cartridge drop down menu, and then select Apps/Extended BASIC. Let Classic99 reset. The emulator will now behave as if you had the Extended BASIC cartridge inserted. Press any key to begin...



Press "2" to start TI Extended BASIC.

You may also use **Extended BASIC 2.9 GEM** included with the Jewel Extended Basic Compiler, adding some useful commands, but if you are just returning to or starting with the TI-99/4a, this would add another new software to learn not necessary at this point.



We are ready to write a program now.

Type in the following commands:

```
10 PRINT "HELLO WORLD"
```

```
RUN
```

Note for Non-English Keyboard Users: The quotes and other special characters may not be where you expect them to be because of the different keyboard mappings on non-english PCs.

Example: On a German keyboard press Shift-Ä to produce the double-quotes.

This is the first reason for using TiCodEd: All keyboards work as expected **universally** within the TiCodEd editor environment.



Congratulations! You just made a TI BASIC program run in emulation!

Before we dive into TiCodEd we need to set up a Classic99 emulated floppy drive for storing programs. Both TiCodEd and Classic99 will have access to this shared storage resource. In this tutorial we use "**DSK4**" as the identifying name for this shared drive. Please do not name the shared floppy "**DSK1**" as we will use that drive name for a purpose described later.

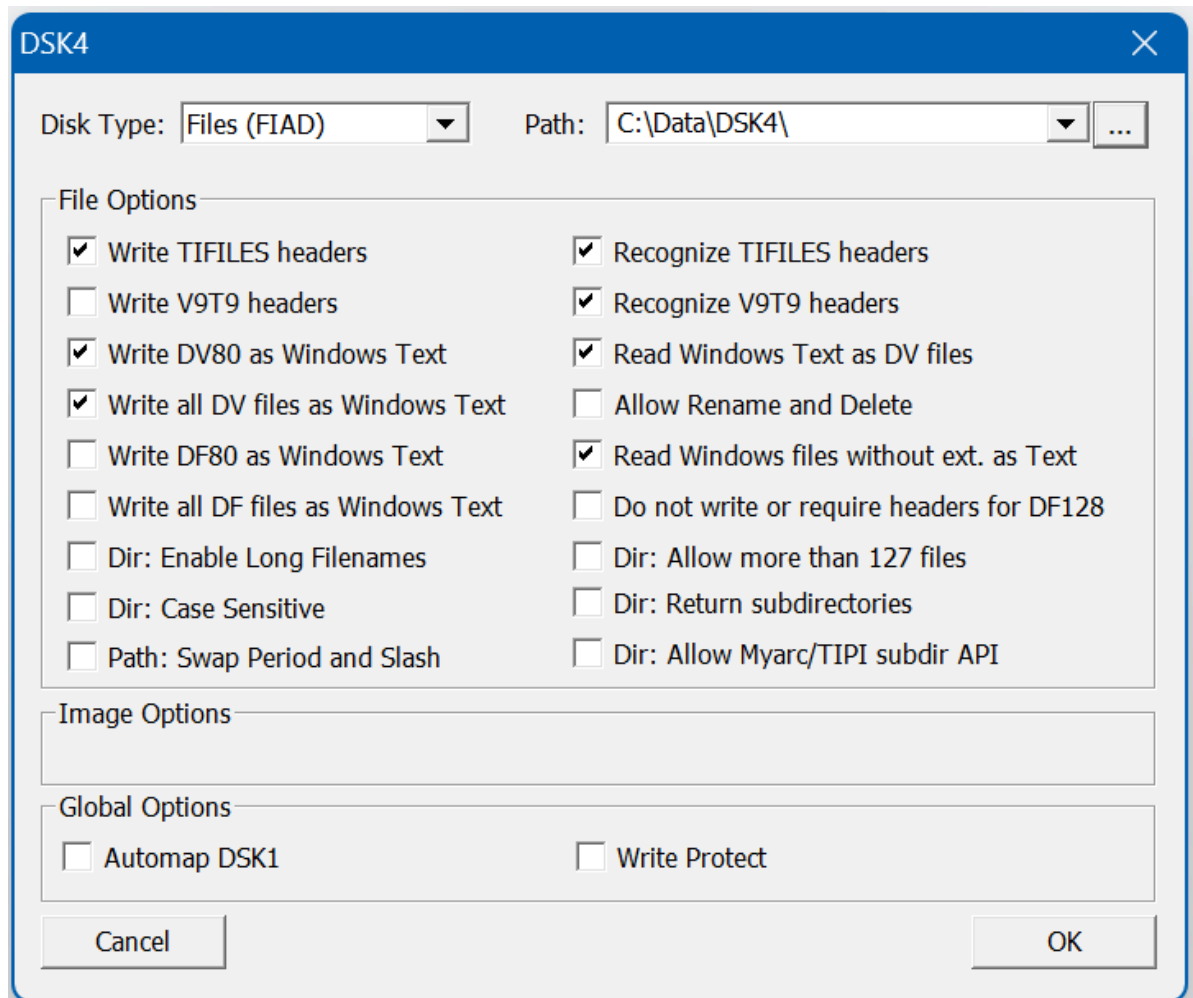
Computer keyboards have improved significantly over the past 40 years. Mass storage drives have improved even more. **Classic99** offers two ways to emulate floppy disk drives.

One is the **Disk-Image** method, which is a single "container file" on your PC which contains a byte-by-byte image of a physical TI-99/4A diskette. You need special programs like **TI99dir** to look inside Disk-Images, and to read or add files within. The Windows file manager will not allow you to view or manipulate these TI Disk-Images. Disk-Images are native to the ancient TI-99/4A floppy disk system, not the modern Windows system.

The second way Classic99 emulates disk drives is the Windows compatible format named "Files in a Directory", or **FIAD** for short. FIAD allows the emulator to treat a windows directory on your PC as a virtual TI-99/4A floppy disk.

Please create a directory for your TI programs (I have created C:\Data\DSK4 for this) and use the following settings for **DSK4**:

Click the "Disk" drop-down menu at the top of Classic99. Under the "Disk" menu you find "DSK4", hover your mouse pointer over it, and you will get two options, select "Set DSK4". Then you will get this pop-up window where you must first select "Files (FIAD)" from the Disk Type drop down menu.



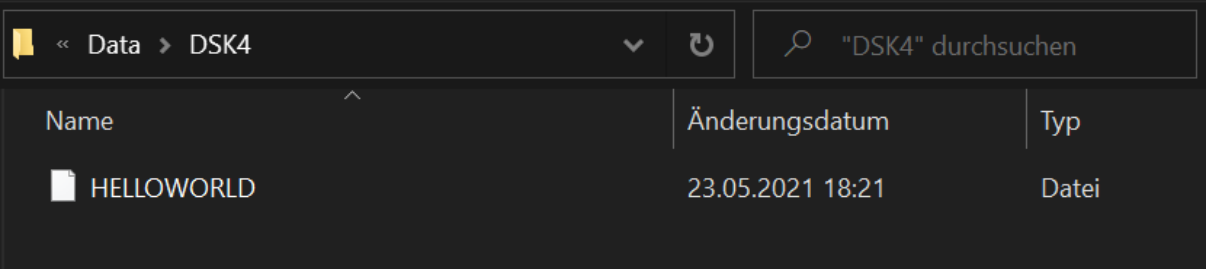
The directory path will be empty. To establish a path you want to click on the "three-dots" button just right of the Path drop-down dialog box. Another pop-up window will appear. I choose to create DSK4 in my C:\Data folder, you may choose differently, i.e. alongside the DSK1-3 in your Classic99 directory. Please remember the location and use it instead of C:\Data\DSK4 in the upcoming examples or stick with my suggestion. The selected directory must not be empty to be selectable. Copy any file to the directory when needed.

Click on "New Folder" to create the folder.

Select the same check-box options as you see above in order to properly configure your virtual floppy drive. If you have questions about any details pertaining to these check-box options you will find the answers thoroughly explained in Tursi's excellent Classic99 manual. Beginners need not burden themselves just yet with this information. Moving right along is encouraged!

Now save your little program by typing: SAVE DSK4.HELLOWORLD

This action will put the file named HELLOWORLD in the FIAD virtual floppy drive folder.



Name	Änderungsdatum	Typ
HELLOWORLD	23.05.2021 18:21	Datei

This HELLOWORLD file is not stored as a text file nor any other Windows file type. The saved file is coded for use with Extended BASIC and the TI-99/4A system. It is actually stored in what is known as a “tokenized” format known as TIFILES. We will take a look at what “tokenized” means in a later chapter. This more advanced chapter is aptly named: [A daunting look at TIFILES](#). Move along!

Copy and paste

Tursi included a nice feature in the emulator to get around typing-in endless listings. Most of us old guys typed in programs from magazines and books back in the day. Today we can use the windows clipboard to copy program text from any editor or even this online manual, and then conveniently paste the clipped program text into the Classic99 emulator.

Let’s give this skill a try...

In Classic99 we type “NEW” to delete our program from memory.

Next let’s select/highlight the following lines from this document and press Control-C to copy them.

```
100 FOR I=1 TO 10
110 PRINT I
120 NEXT I
```

Go to the emulator and select Edit / Paste XB (which stands for Paste Extended BASIC) to paste the above lines into Classic99.

Convention: The **gray boxes** will always contain something you will need to enter into the TI (usually by cut & paste, no need for typing). The output of the TI or TiCodEd are in **white boxes**. The **blue boxes** are screenshot graphics and can’t be selected as text to be pasted from the emulator.

```
* READY *  
>100FOR I=1 TO 10  
>110PRINT I  
>120NEXT I  
>
```

Even though the spaces after the line numbers are missing, somehow the program is transferred correctly. Type "LIST" to check this and "RUN" to give it a try. By listing see that spaces have been retained after the line numbers.

The handy Cut & Paste feature in **Classic99** empowers us to use a PC-based code editor to write our programs. No more dealing with changed keyboard layouts or typing on TI's compromised keyboard. You will also avoid the anciently cumbersome line-by-line editor which is permanently inefficient.

No longer will you be bound to actual TI-99/4A hardware for development. The thought of developing on "real iron" is rooted in nostalgia ... not efficiency. Truth be told, you will be FAR MORE PRODUCTIVE developing on a modern machine with an emulator and the TiCodEd editor.

Another major benefit of TiCodEd is the ability to easily save your program in the "tokenized" native TI format. Other editors save programs in formats compatible with Windows computers. TiCodEd automatically saves your programs to the native TI format (TIFILES), placed conveniently inside an emulated floppy drive compatible with the **Classic99** emulator.

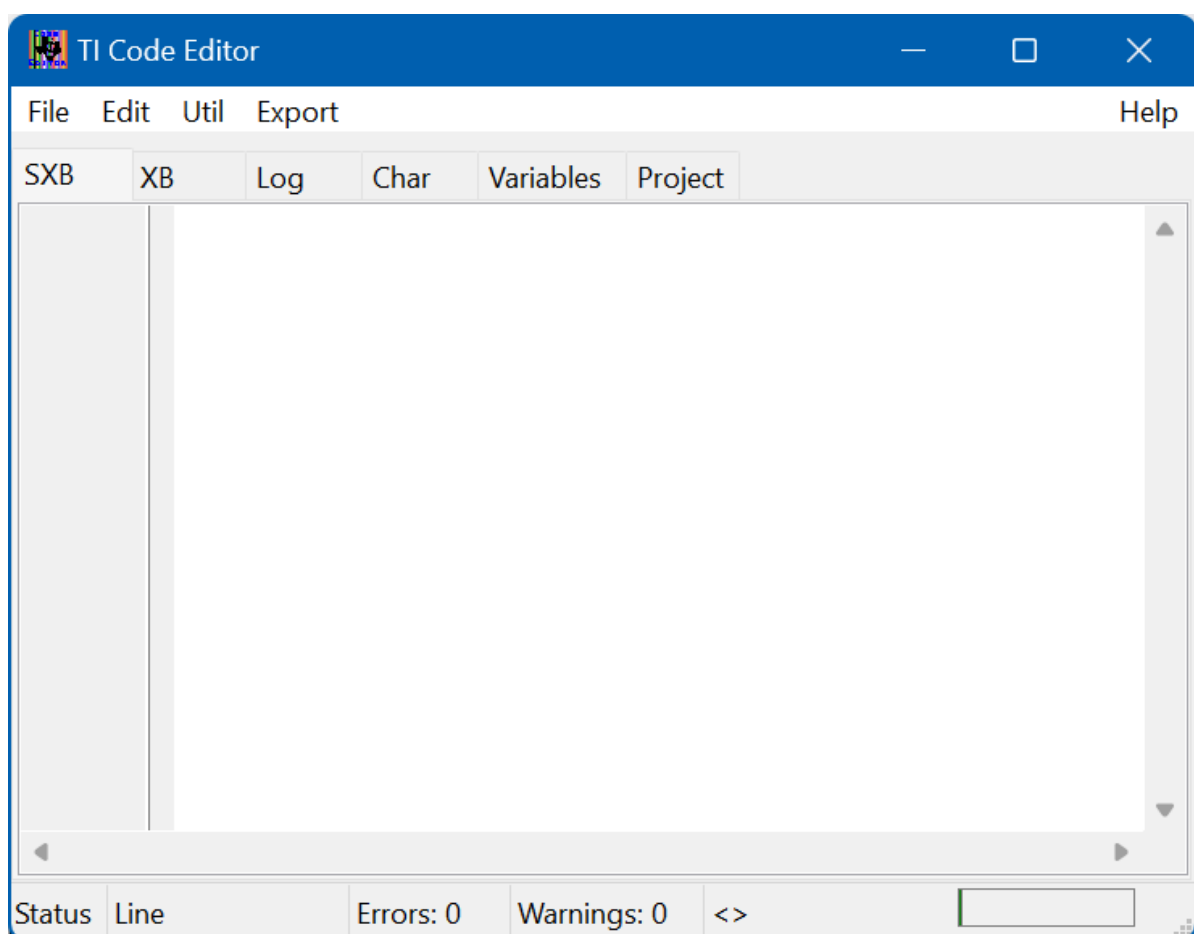
Pro Tip: Cutting and pasting code into Classic99 is indeed a powerful workflow enhancement. That said, as your programs grow in size you will discover that pasting them into Classic99 will become rather time consuming. Cut & Paste is best used with small program segments from this manual for example. The super-slow pasting of code into Classic99 was simply an unavoidable speed-bump until the advent of TiCodEd. When you edit your programs with the TiCodEd editor, and then save them to a Classic99 emulated drive, the entire BASIC workflow becomes faster, more enjoyable, speedy, pleasant, and fun. New programming tools for retro computers remove the headaches we all suffered in the past. More enjoyable programming with less tedious housekeeping is the TiCodEd goal.

Another Pro Tip: Classic99 implements a "CLIP" device for the Windows Clipboard. You may type LIST "CLIP" to copy your program to the clipboard!

DEFINITION: *Cross-Platform Development*, programming your retro-computer using modern computers with custom software tools. Increasing efficiency, and ultimately simplifying the process of developing programs for older machines by leveraging the power differential between your development system and the target system. Working smart, not hard. The opposite of sadistic. Lowering the barriers of entry.

First use of TiCodEd

Start TiCodEd and then have a look around. Go to the folder where you installed TiCodEd and find the file named, "TiCodEd" or "TiCodEd.exe", double-click on it and the program starts up. You should see this screen:



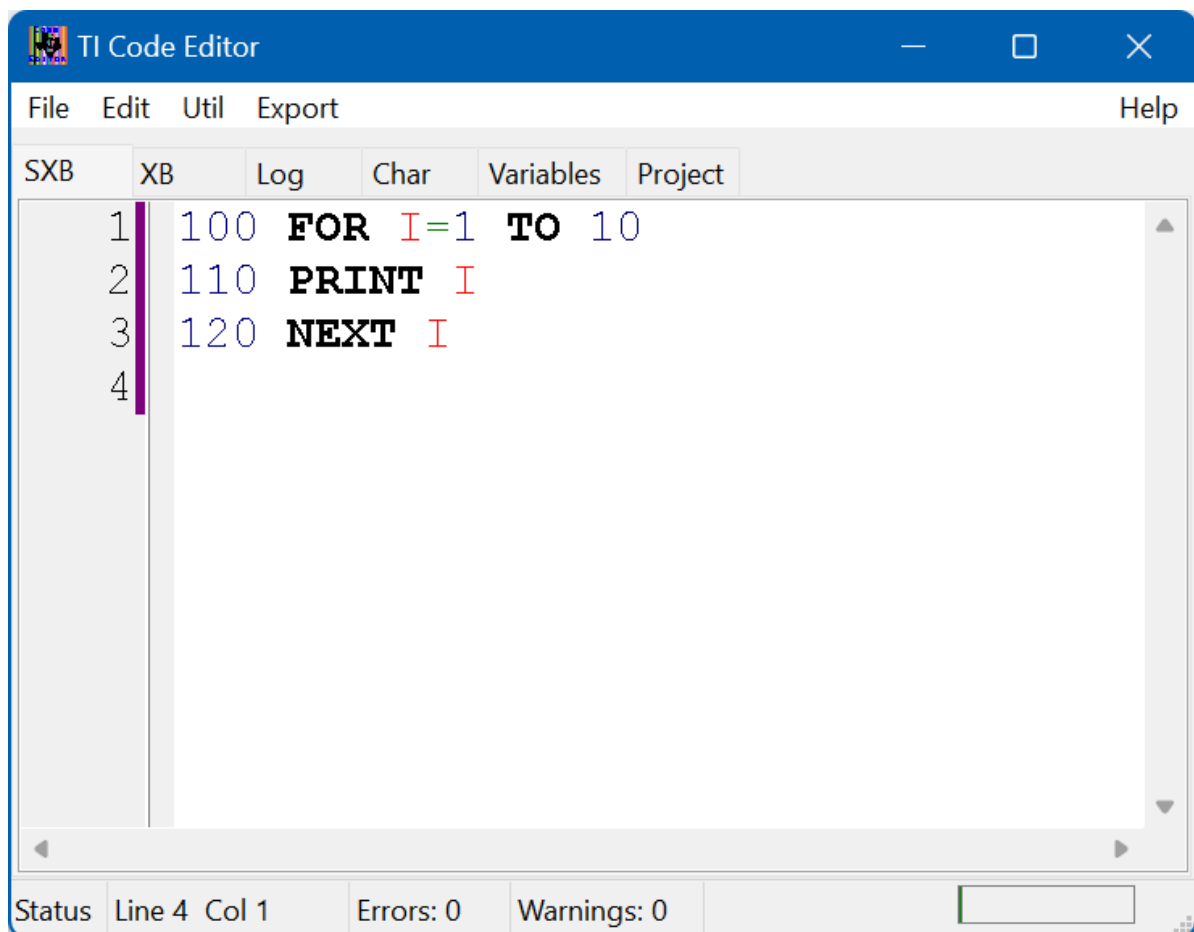
SXB tab - Structured Extended BASIC

You have six tabs below the File/Edit/Export menus. We will start learning the SXB-tab by recalling a previous program example. Once again copy the following code from below and then paste it into TiCodEd's SXB-tab.

```
100 FOR I=1 TO 10
110 PRINT I
120 NEXT I
```

The SXB tab is the main programming tab where we'll do all of our coding. SXB stands for "Structured Extended BASIC", an enhanced (modernized) version of Extended BASIC. In our example we will use line-numbers and stick with the traditional TI Extended BASIC commands.

NOTE: The advantages of no line numbers and powerful new Extended BASIC extensions and commands will be covered later in the tutorial.



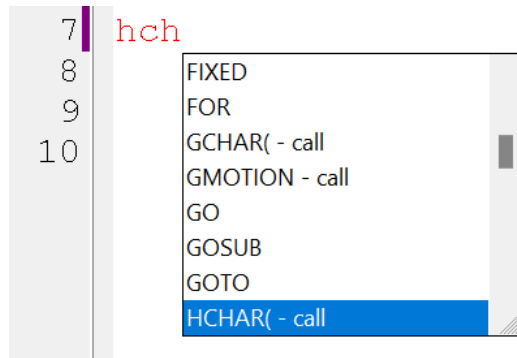
The first thing you may notice is something called **Syntax Highlighting**. In the TI Code Editor window all **keywords** are black, **identifiers** like variables, subroutines and labels are red and **line numbers** and **literals** are colored blue. Secondly, you notice a handy **violet stripe** on the left column, indicating that these lines have not yet been saved. So let's do this.

Select File / Save from the menu. Select the DSK4 directory you created in the previous chapter and give this program a name, i.e. "**forloop**". This not only saves your program as an SXB file named **forloop.sxb**, but also creates a second noteworthy file.

The *second file* is what we call a "Project File," with the extension XBP (**forloop.xbp**), short for "Extended BASIC Project." It is filled with some information we will explain in the next chapter.

The **violet stripe** should have changed to **dark green** to indicate the program lines have been saved (to the DSK4 folder).

The editor has a **code completion** feature. Start typing a command, a subroutine, a label or a variable and press Control-Space.

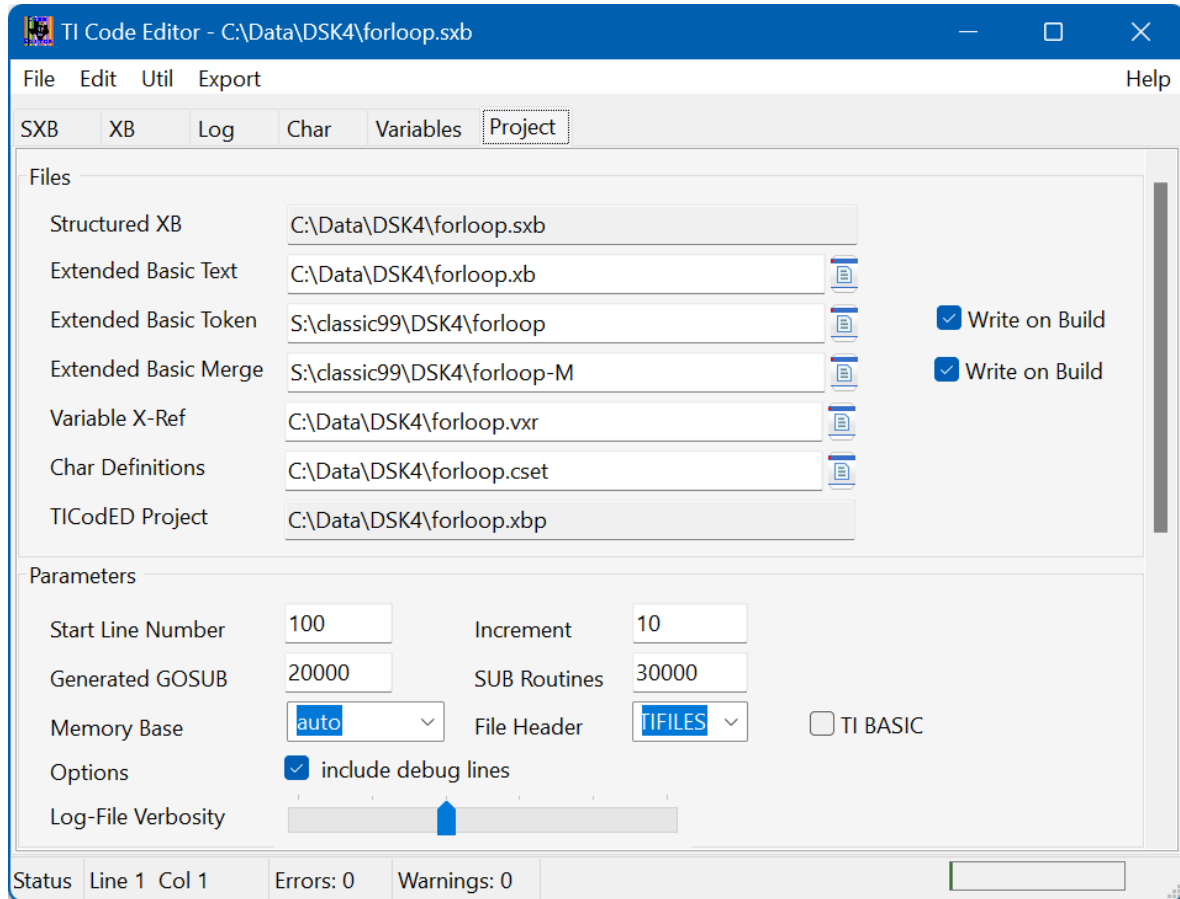


Labels, subs and variables are based on the last build-run. Only variables with at least 3 characters are included. Subs from included libraries are also included.

Details on code completion can be found in chapter [Advanced Editor Features](#).

Project Tab

We select now the Project Tab to see:



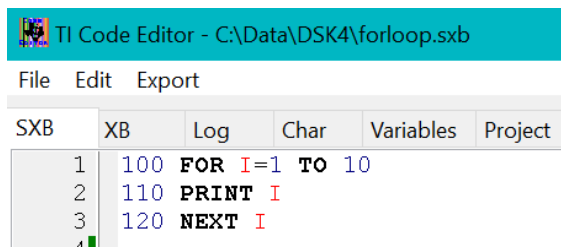
Here there are several auto-filled settings. The filename you entered is taken as the base-name for multiple files related to this project. The **.sxb** file is our Structured Extended BASIC program. This **.sxb** file is closely tied to the **.xbp project** file, containing the configuration of this page for your program. Both the **.sxb** and **.xbp** entries are read-only to be tied together. Each SXB file has its own settings stored in the corresponding .xbp project file.

All other file names may be edited or selected using the icon on the right of all filenames.

TiCodEd automatically translates all Structured Extended BASIC (**.sxb**) program files to TI Extended BASIC compatible text (**.xb**), for convenient cut & pasting into Classic99. This auto-conversion feature completely eliminates the need for third-party code-translator applications such as **TIdBiT**. In fact, Matthew Hagerty's **TIdBiT** inspired this code-translation feature.

Later in the manual we will discuss which enhancements are available to the normal TI Extended BASIC command set. If no SXB extensions are used, as in our first intentionally simple example, the resulting Extended BASIC program (**.xb**) file will look exactly like the Structured Extended BASIC (**.sxb**) file, as there's no SXB specific code to convert to Extended BASIC.

Click on the SXB tab to see the contents of the .sxb file which features syntax highlighting.



Remember that **.xb** files are just glorified text files. They are useful for pasting into the Classic99 emulator and for sharing with friends online. The same type of text files which can be examined by the Windows Notepad app. The Classic99 emulator cannot load **.xb** files so...

The *fourth file type* created by TICodEd is the "tokenized" (real TI-99/4A and Classic99 compatible) version of our BASIC program. This file type (also called TIFILES) has no extension after the program name. Tokenized files can be loaded directly and swiftly into Classic99 by typing: OLD DSK4.FORLOOP

DEFINITION: *Tokenized*, Tokenized BASIC is a method of storing programs in the BASIC programming language by encoding the various keywords of the language as "tokens" instead of as plain text. Such programs take up less storage space in memory and in external storage such as disks or tapes, which was a significant concern in an era when computers were much more limited in memory and disk space than they are at present. Since computers are much faster and have much more memory and disk space now, tokenized languages are rarely used for source code storage.

There's a 'Write on Build' checkbox for Extended BASIC Token files under the Project Tab. This checkbox must be checked if you wish for TICodEd to automatically generate this type of file when Control-B, or when Export/Build Project is selected. You will want to leave this checkbox selected.

In the future you will eventually want to use the BASIC Compiler for speed enhancements, which requires reading our *fifth program file* format called the **Merge** format. You may manually generate a **Merge** file version of your program by typing, "SAVE DSK4.FORLOOP-M,MERGE" on the TI-99/4A, or similarly in Classic99.

Fortunately for us TICodEd users, the **Merge** (Compiler compatible) format version of your program is created automatically by pressing Control-B, or by selecting the drop down menu Export\Build Project. The **Merge** file will be automatically generated if the Extended BASIC Merge "Write on Build" checkbox is checked. This checkbox is also located under the Project Tab.

At this point in the tutorial we want both "Write on Build" checkboxes selected/checked under the Project Tab.

You have the option to modify the program file locations under the Project Tab. You may store these files anywhere you wish. You may also give, for example, any file a shorter name, as you have to type it every time you want to load it in Classic99. An abbreviated "OLD DSK4.FL" comes in handy.

Your character definitions are stored in the **.cset** file.

The final file type is the Variable X-Ref ("Variable Cross-Ref" **.vxr**) which stores a record of which XB line a variable is used. X-Ref comes in handy during debugging. Especially when you struggle to see why a variable value does not match what you expected. The generated file is also used in the "Variables" tab which is explained later in the manual.

With the multiple files you can split between multiple directories. You may put your source code in a Github directory, and your generated files in a local directory used by Classic99.

Let's also have a look at the **Project Tab's Parameter section**. In SXB you can code without line numbers. In place of line numbers is the use of Labels and structured WHILE-WEND and REPEAT-UNTIL loops. This is how modern commercial incarnations of the BASIC Programming Language work.

The **Start Line Number** and the **Increment** options function just like TI Extended BASIC's original RESEQUENCE statement. The first line in the resulting .xb program text file gets the Start Line Number. The following lines in the .xb text file get the Increment added to form additional line numbers, as TiCodEd assigns numbers to all program lines, while converting Structured Extended BASIC programs to TI-99/4A readable (standard) Extended BASIC.

NOTE: TiCodEd allows you to program in the modern structured BASIC style. The only reason TiCodEd creates .xb files with line numbers is to satisfy the 1970s-legacy BASIC program format requirements inside TI-99/4A hardware.

Please be aware that mixing manual line numbers with the automatic line numbering capability of TiCodEd may be problematic. The manual number must always be equal to or higher than the first *Start Line Number* automatic number.

Some SXB features generate **GOSUB** statements, which are by default numbered starting 20000, **SUB/SUBEND** routines are by default starting in line 30000. Both may be changed here.

For beginners, the next three settings can remain untouched, they will be discussed under advanced topics.

The **Debug** feature is a simple, yet effective way to enable or disable debug statements in your program. If checked, the statements marked as debug will be included, if unchecked they will be excluded. Similar to the use of **!-symbol** as a remark, the **#-symbol** is used to mark a Debug line. Let's try it.

```
140 # PRINT "This is a Debug Line"
```

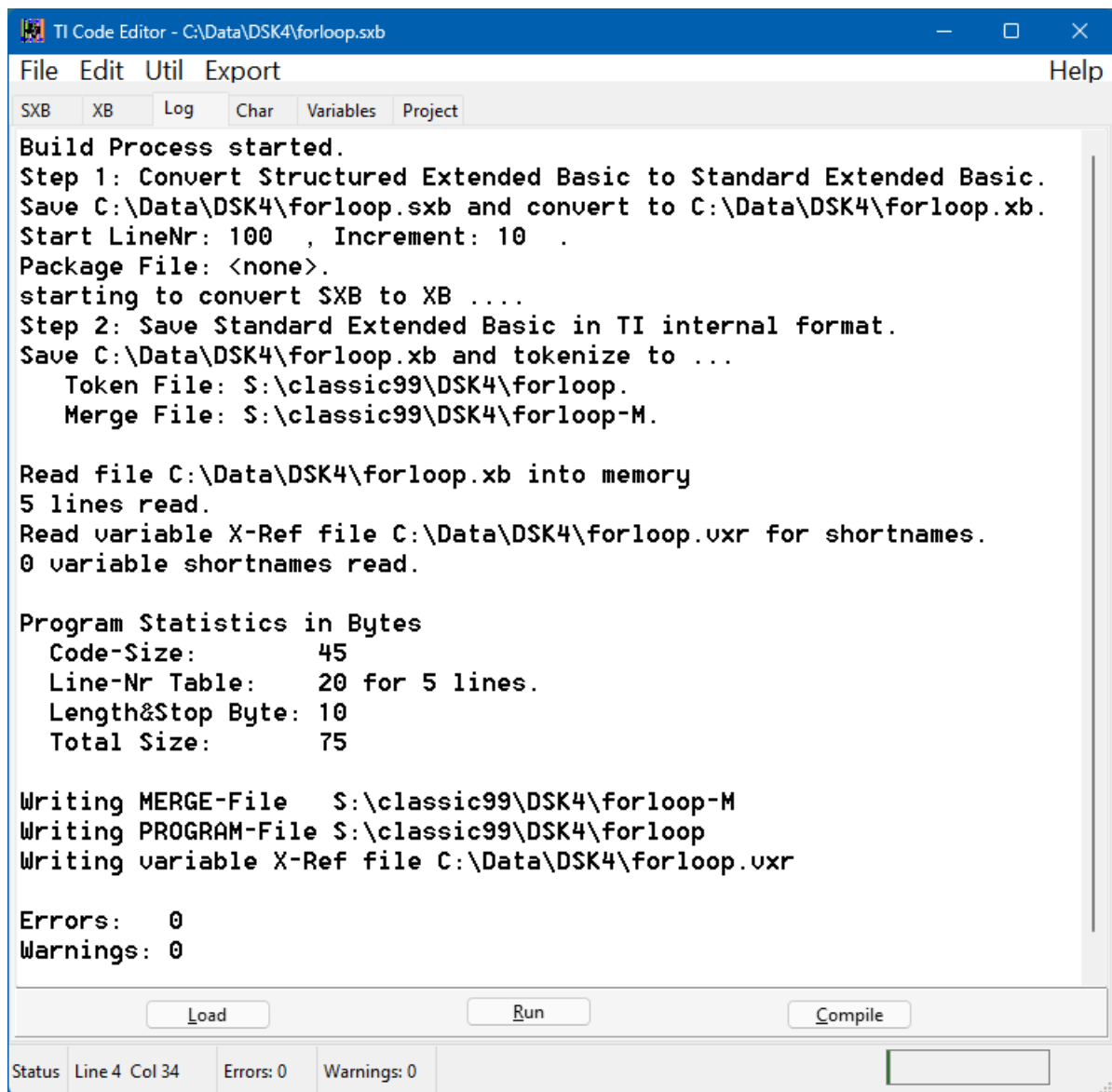
Finally we come to the **Log-File Verbosity**.

1. Error - Only Errors are shown
2. Warning - Errors and Warnings are shown
3. Information - Errors, Warnings and Information are shown
4. Verbose - Also included debug information
5. Very Verbose - More Debug information
6. Unbelievable Verbose - Debug down to the bits

Usually the default of level 3. "*Information*" is sufficient. We will also have a look at the level 6 "*Unbelievable Verbose*" later in chapter [Understanding the Log](#). Understanding the log will help you debug your programs.

The Project Tab lower sections titled **Libraries**, **Post-Processing** and **Integration** will be discussed later. For now let's return to our simple example.

We are all set for the first "**Build**" of our Project. Select **Export / Build Project** from the menu or just press **Ctrl-B**.



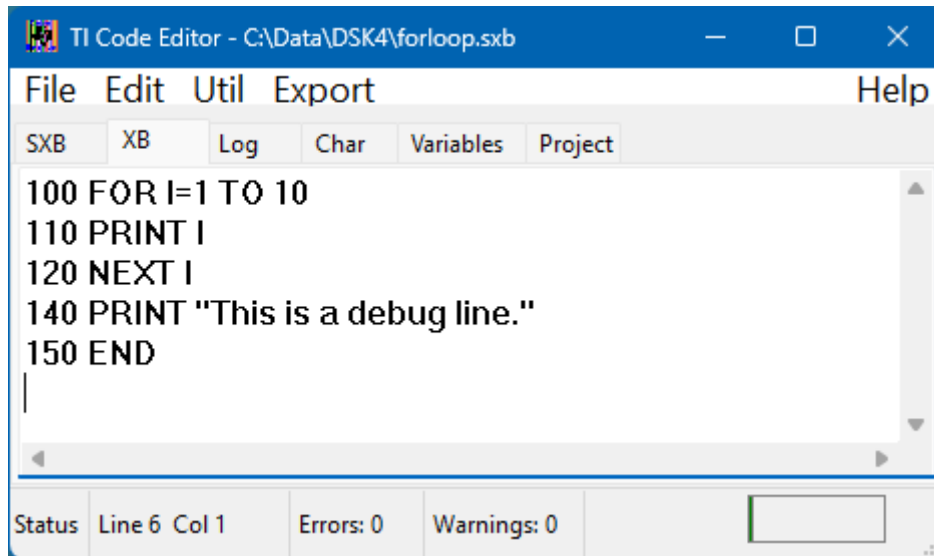
The active tab automatically switches to the Log tab after you build a project. [The first time you build a project you'll actually see the "Save New File" window.] Basically, the single most important part of the log tab is the last two lines: Errors 0, Warnings 0.

The build process consists of two steps:

1. The **.sxb** program file is saved and converted to a standard **.xb** text file.
2. The **.xb** file gets converted in the selected output formats (Extended BASIC Token and/or Extended BASIC Merge) and the Variable X-Ref file gets written.

* Building also generates a short program statistic in the log and refreshes the table for code-completion and other features needing details of your program.

Now lets click on the XB tab to view the auto-generated standard XB text:



The screenshot shows the TI Code Editor window with the title bar "TI Code Editor - C:\Data\DSK4\forloop.sxb". The menu bar includes "File", "Edit", "Util", "Export", and "Help". Below the menu bar are tabs: "SXB", "XB", "Log", "Char", "Variables", and "Project". The "XB" tab is selected, displaying the following BASIC code:

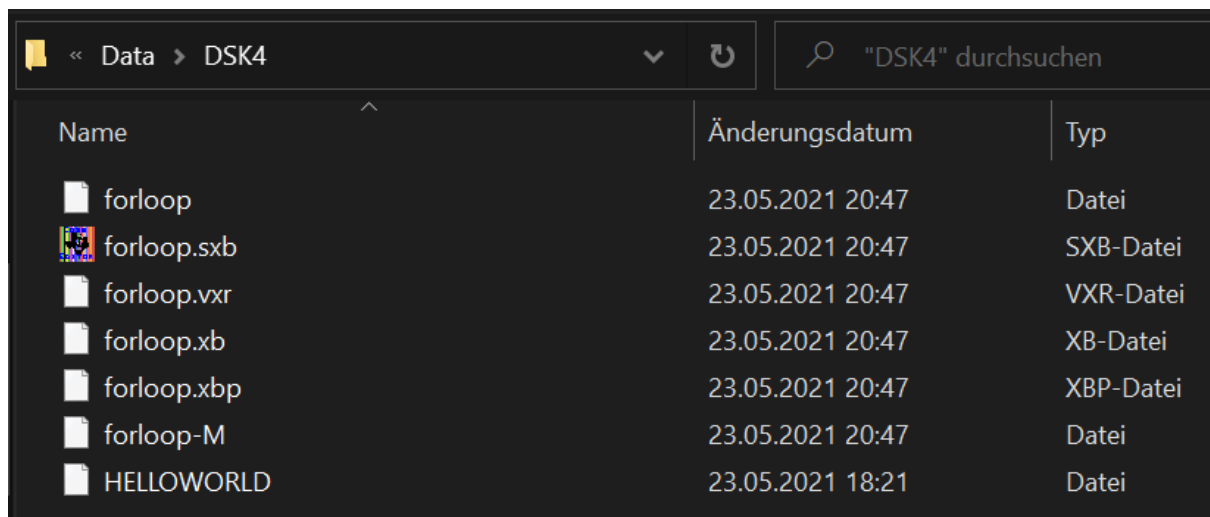
```
100 FOR I=1 TO 10
110 PRINT I
120 NEXT I
140 PRINT "This is a debug line."
150 END
```

At the bottom of the window, the status bar shows "Status Line 6 Col 1", "Errors: 0", and "Warnings: 0".

As we were not using any SXB features in this example, the XB tab code looks exactly like the SXB tab code, minus the syntax highlighting.

The “#” in line 140 got removed as we chose “include debug lines”.

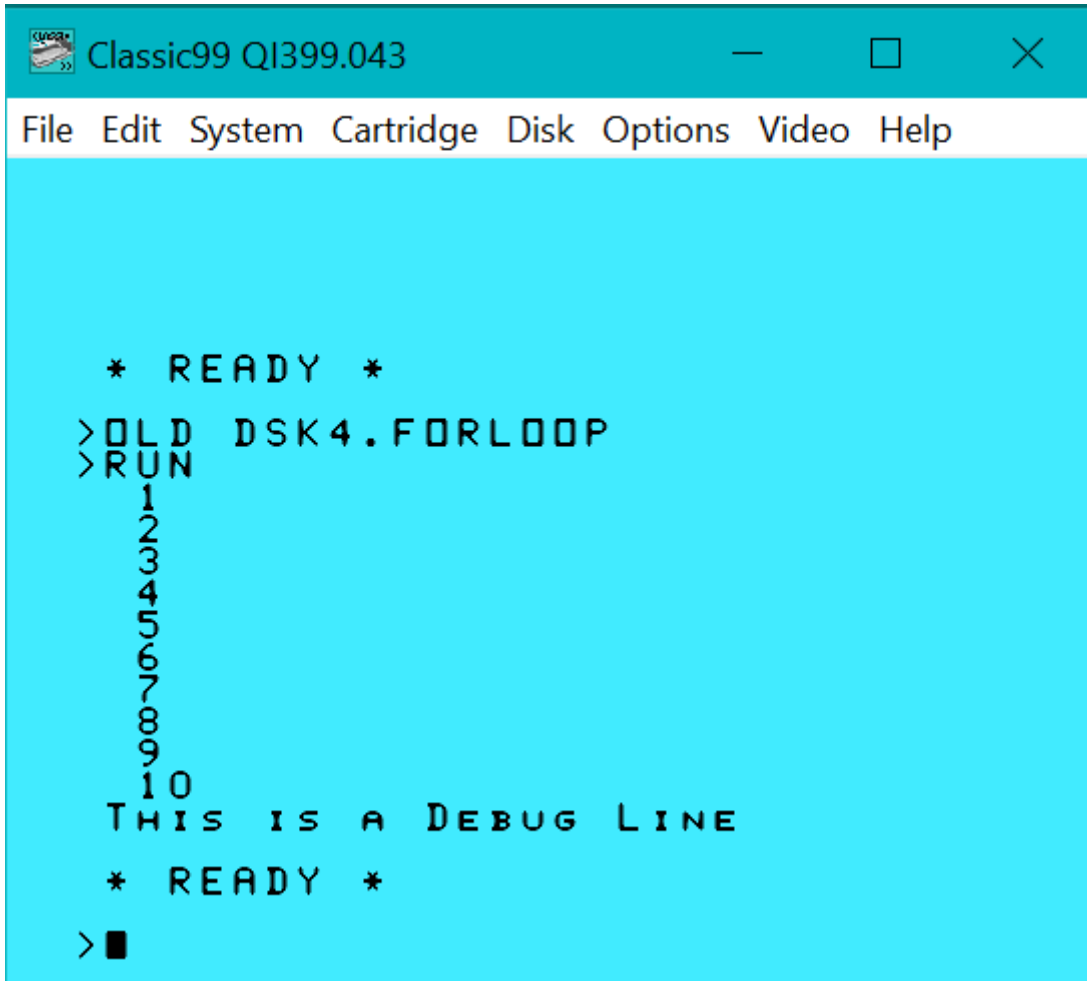
Switch over to the Classic99 emulator to load and run our program. All of the necessary auto-generated files are now in the DSK4 directory:



The screenshot shows a file explorer window with the address bar set to "Data > DSK4". The search bar contains the text "DSK4" durchsuchen. The file list is as follows:

Name	Änderungsdatum	Typ
forloop	23.05.2021 20:47	Datei
forloop.sxb	23.05.2021 20:47	SXB-Datei
forloop.vxr	23.05.2021 20:47	VXR-Datei
forloop.xb	23.05.2021 20:47	XB-Datei
forloop.xbp	23.05.2021 20:47	XBP-Datei
forloop-M	23.05.2021 20:47	Datei
HELLOWORLD	23.05.2021 18:21	Datei

First clear the memory with NEW and then type “OLD DSK4.FORLOOP”.



```
Classic99 Q1399.043
File Edit System Cartridge Disk Options Video Help

* READY *
10 THIS IS A DEBUG LINE
* READY *
>
```

You may deselect "Debug" in the Project tab and build again to see that line 140 will be omitted.

Why Line Numbers?

While programming in most Home Computer variants of BASIC we have three traditional arguments for using line numbers:

1. Line numbers give your code lines a logical sequence. Line-numbers are always executed in ascending order, except you have an explicit jump, like a GOTO statement. When you want to add a line between two existing lines you simply pick a number between these two line-numbers, and Extended BASIC will insert your new line accordingly.
2. Using line numbers is to identify a line for editing. In TI BASIC you have the EDIT nnn command, in Extended BASIC you type the line-number and press Fctn-X and the line-editor comes up with the requested line.
3. You may need to refer to a line in your program with a GOTO, GOSUB, RESTORE etc. Line numbers allow you to transfer control easily.

While these first two points are essential while programming in BASIC on the real TI-99/4A (as well as on all other Home Computers of the era), they both become superfluous while using a full-screen PC-based editor. These two arguments for line numbers essentially disappear when you escape the outdated line-by-line editor environment. TiCodEd is a full-screen editor.

The third argument for using line numbers remains somewhat valid. You will still need to identify individual lines in your code for program flow control.

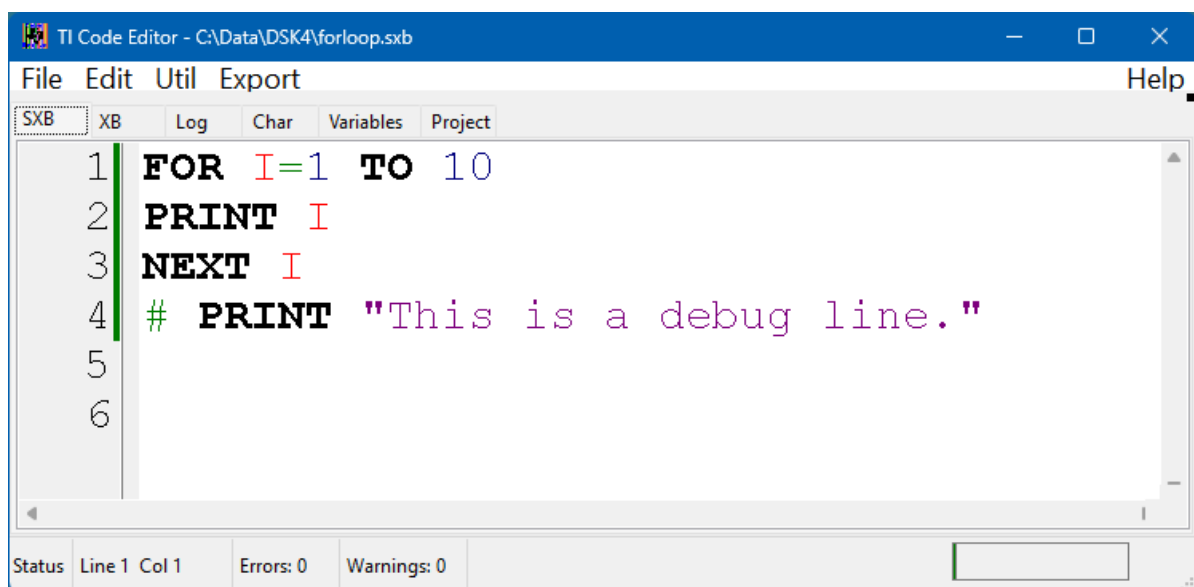
When would we ever need to refer to an individual program line?

There are four cases:

1. You want the program to jump to a specific line.
2. You need to loop-back in a loop.
3. You want to execute a subroutine with GOSUB.
4. You have some information stored there to RESTORE or PRINT USING.

Getting back to our program example, do we need line-numbers for any of the cases listed above? The answer is ... NO!

Lets just try deleting the line numbers for our code in the SXB tab. When we build the project, notice how we still get the same XB translation with line numbers. TiCodEd inserts lines 100, next 110, and so on...to satisfy the format expectations of both the real TI-99/4A and the Classic99 emulator.



```
1 FOR I=1 TO 10
2 PRINT I
3 NEXT I
4 # PRINT "This is a debug line."
5
6
```

Our short program segment has a loop. How does the program know where to go back in the loop? The FOR statement in the original Extended BASIC was already a structured loop command. NEXT does not need any line-number to go back.

A FOR loop has a known number of repetitions. What about loops with variable repetitions, ending when a condition is met, like a key pressed? There Extended

BASIC only offers IF THEN GOTO control option. Other languages like **Pascal**, **C** or **Java** never had line numbers to make this work. They have two other ways of defining a loop. They have **REPEAT...UNTIL** and **WHILE** loops, which are now also introduced to Structured Extended BASIC, and are explained in the next chapter.

What about the GOSUB? A call to a subroutine and returning to the calling statement is a very structured approach. It is a direct implementation of what is known as a Top-Down approach. One statement in the main program is actually executed as many steps in the sub-program. If only a GOSUB could be referred to by something more intuitive. Why don't we give the subroutine a well-sounding name? For example WaitForKeypressed or PrepareScreen?

Which is more intuitive?

- GOSUB 400
- GOSUB PrepareScreen

Trick-Question, yes. The second GOSUB example uses a Label instead of a fixed line number.

Labels are explained in detail in the chapter [The use of Labels](#). We define those Labels at the beginning of a line by typing the name followed by a colon, as in the following example:

```
PrepareScreen:
  CALL CLEAR
  CALL SCREEN(2)
RETURN
```

With the **indentation** you can see what belongs to the subroutine. It is optional to do indentations or keep the label on a line of its own.

You can have simply written:

```
PrepareScreen: CALL CLEAR
CALL SCREEN(2) :: RETURN
```

This will also work but it hurts my eyes. Let us at least try to write beautiful looking code with clarity. It is much easier to understand code which follows the guidelines of structured programming. If it's easier to read, it will be easier to debug -- simple.

When there is no need for line-numbers, is there a need to abandon them?

Yes, when you remove line numbers you remove complexity. Every line number in a non-structured BASIC program is a target "entry-point" for a jump via GOTO or GOSUB. Since numbers can be easily mistyped, a GOTO 100 and GOTO 1000 are just one keystroke apart. Reducing the entry points to an absolute minimum decreases the potential for errors.

Using loops without GOTOs, and using labels for all intended GOSUB entry-points and reference points (USING, RESTORE) will significantly reduce programming errors. Labels are more intuitive and readable than numbers.

Labels are also “portable” as they allow you to re-use program code segments from one project to the next without painstakingly renumbering everything. Line numbers are “fixed,” in-place, and inflexible by design. This practice of programming with line numbers is truly an undesirable relic.

Our first *real* Structured Extended BASIC Program

Let me demonstrate how Structured Extended BASIC loops are used. This chapter demonstrates the basic concept of SXB with a simple demo, calculating the square of a number until the user enters 0. Start a new SXB project by selecting File / New in the menu and paste this code to the SXB tab.

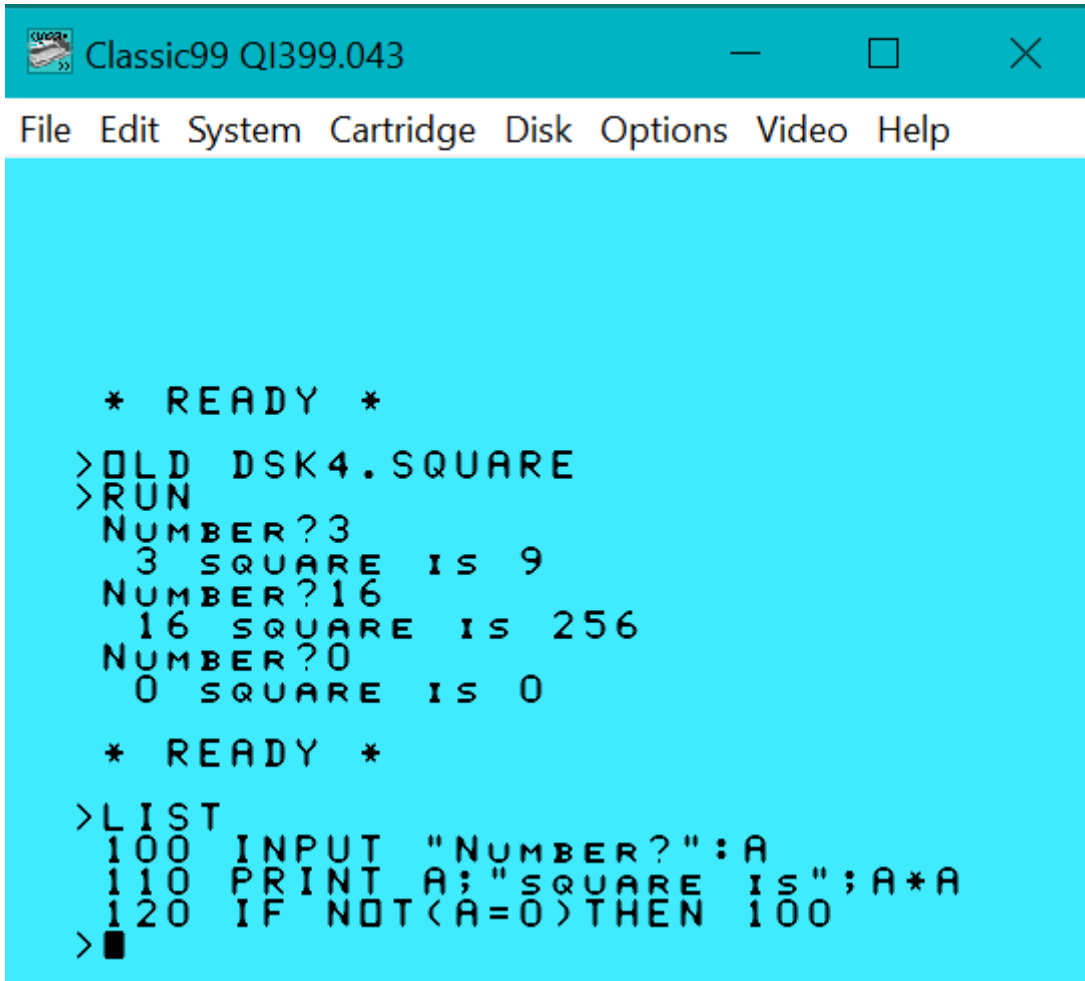
```
REPEAT
  INPUT "Number?":A
  PRINT A;"square is";A*A
UNTIL A=0
```

Now select **File / Save** and save the file under “square”. As the defaults on the project page are reasonable, you directly can type Ctrl-B to build. The XB tab gives you the resulting standard XB file:

```
100 INPUT "Number?":A
110 PRINT A;"square is";A*A
120 IF NOT (A=0) THEN 100
```

The lines get the line numbers assigned as defined in the Start Line Number and the Increment on the Project tab. The REPEAT-UNTIL gets translated to an IF-THEN statement. The condition needs to be negated, as the execution has to be repeated when the criteria is not met.

This can run on standard XB: Load it from the shared directory or cut and paste it into the emulator to try.



```
* READY *
>OLD DSK4.SQUARE
>RUN
NUMBER?3
 3 SQUARE IS 9
NUMBER?16
16 SQUARE IS 256
NUMBER?0
 0 SQUARE IS 0

* READY *

>LIST
100 INPUT "NUMBER?":A
110 PRINT A;"SQUARE IS";A*A
120 IF NOT(A=0) THEN 100
>■
```

This looks good, except one little glitch ... we do not want to know the square of zero, we enter zero to intentionally end the program. Let's have a look at an alternative loop.

Select File / New and paste this code to the SXB tab.

```
INPUT "Number?":A
WHILE A<>0
  PRINT A;"square is";A*A
  INPUT "Number?":A
ENDWHILE
```

Save this with File / Save under the name of "squarew" and build the project with Ctrl-B. This gives the XB file:

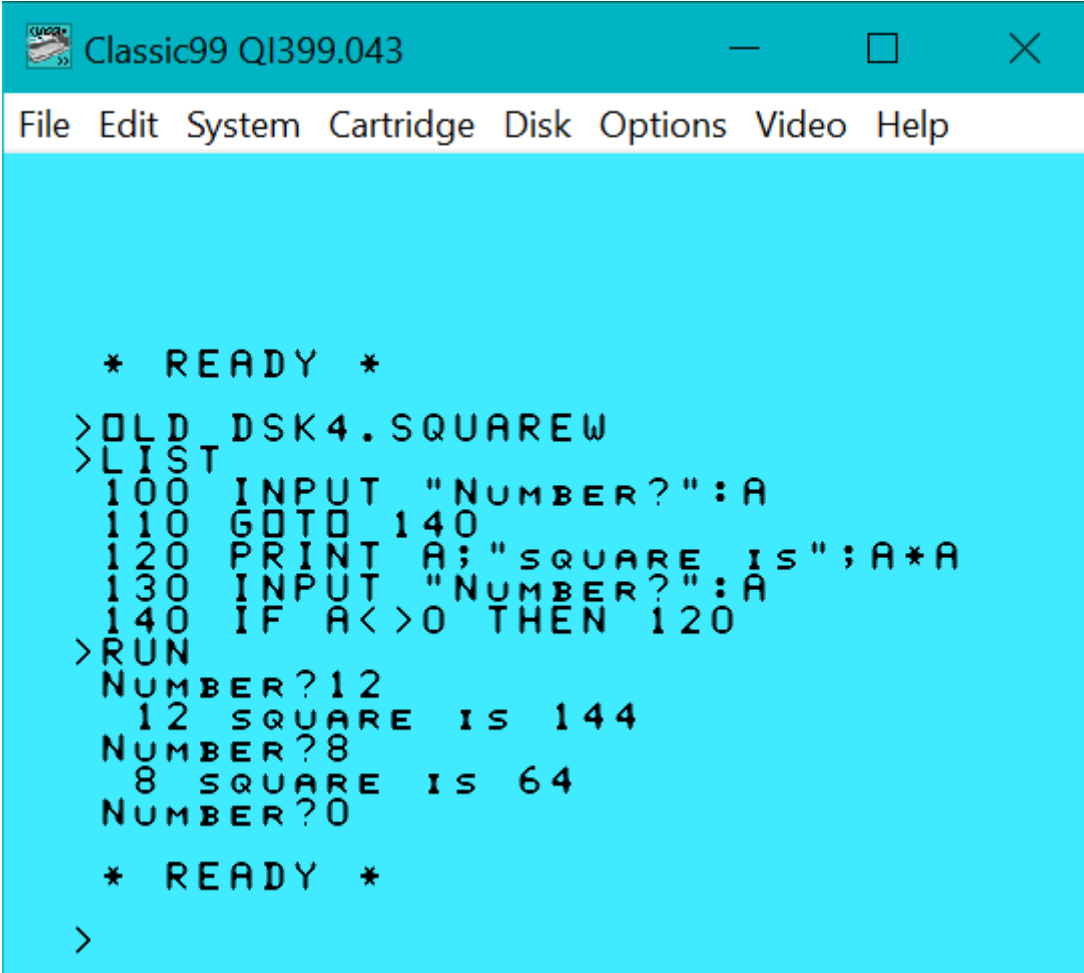
```
100 INPUT "Number?":A
110 GOTO 140
120 PRINT A;"square is";A*A
130 INPUT "Number?":A
140 IF A<>0 THEN 120
```

The WHILE loop is **head-controlled**, meaning the loop is not executed if the condition is immediately met (whereas the REPEAT loop is **foot-controlled** with the condition at the end, therefore executed at least once).

In order to have only one IF statement there is a GOTO inserted to the IF statement to bypass the loop body. It is clear that as long as $A \neq 0$ the loop is executed.

LIMITATION: Please be aware that any use of REPEAT and UNTIL, or WHILE and ENDWHILE, will need to have a line on their own and must not be combined with other statements.

The "read ahead" is a common practice in file processing as you don't know what you read next or there is the end-of-file marker. It requires a read (INPUT) before the loop and as the last line in the loop. Let's check this in the emulator:

A screenshot of a classic BASIC emulator window titled "Classic99 Q1399.043". The window has a menu bar with "File", "Edit", "System", "Cartridge", "Disk", "Options", "Video", and "Help". The main area is black with white text. The program code is as follows:

```
* READY *
>OLD DSK4.SQUAREW
>LIST
100 INPUT "NUMBER?":A
110 GOTO 140
120 PRINT A;" SQUARE IS ";A*A
130 INPUT "NUMBER?":A
140 IF A<>0 THEN 120
>RUN
NUMBER?12
12 SQUARE IS 144
NUMBER?8
8 SQUARE IS 64
NUMBER?0
* READY *
>
```

The ENDWHILE can be replaced with the shorter WEND as used in some BASIC dialects. The indentation of REPEAT and WHILE loop is optional but enhances the readability of the program significantly. Both loops can be nested. A nesting error will be raised when REPEAT/UNTIL or WHILE-ENDWHILE will not match and the build-process aborted.

This is almost all there is about the "structured" in Structured Extended BASIC. How powerful this new SXB capability can be is seen in the game titled "SteveB52" which we will soon discuss. Together with the unchanged FOR-TO-NEXT loop, which does not require line-numbers even in the old

standard XB, you are set to drop GOTO in your programs (almost) completely. In this example we will also introduce the last "structure" bit, the BEGIN/END block.

Why are GOSUB and GOTO statements problematic?

As a programmer you often need to make a few comparisons and then redirect the program's sequence of events to a subroutine. BASIC and Extended BASIC both have the GOSUB statement for such a task. GOSUB is powerful in that it remembers from where it was called, and conveniently returns control back to the calling code sequence after encountering a RETURN statement. GOSUB statements do unfortunately require fixed line numbers (in order to find the called subroutines) in the same way GOTO statements require fixed line numbers to jump to a different sequence of commands.

In a classic game-loop, where you have to start a segment of code over and over again, going back to the top of the loop is commonly accomplished with the GOTO statement. Until SXB we needed to rely on these two fixed line dependent style control-flow statements which often become a challenge to read and debug.

The use of Labels

Perhaps you used [TIdBiT](#) or [xbas99](#) before? Both introduce Labels in place of line numbers. TiCodEd is TIdBiT compatible and even has a port of TIdBiT included in the Export menu. A label is simply an intuitive name for a code segment. We must be careful to avoid name conflicts with statements, variables and CALL subroutines. The syntax for a label is easy. It must be the first word on a line and directly followed by a colon. The basic line may continue after the colon but it is common practice to use labels on a line of its own.

```
GOSUB PrepareScreen
PRINT "Done!"
END

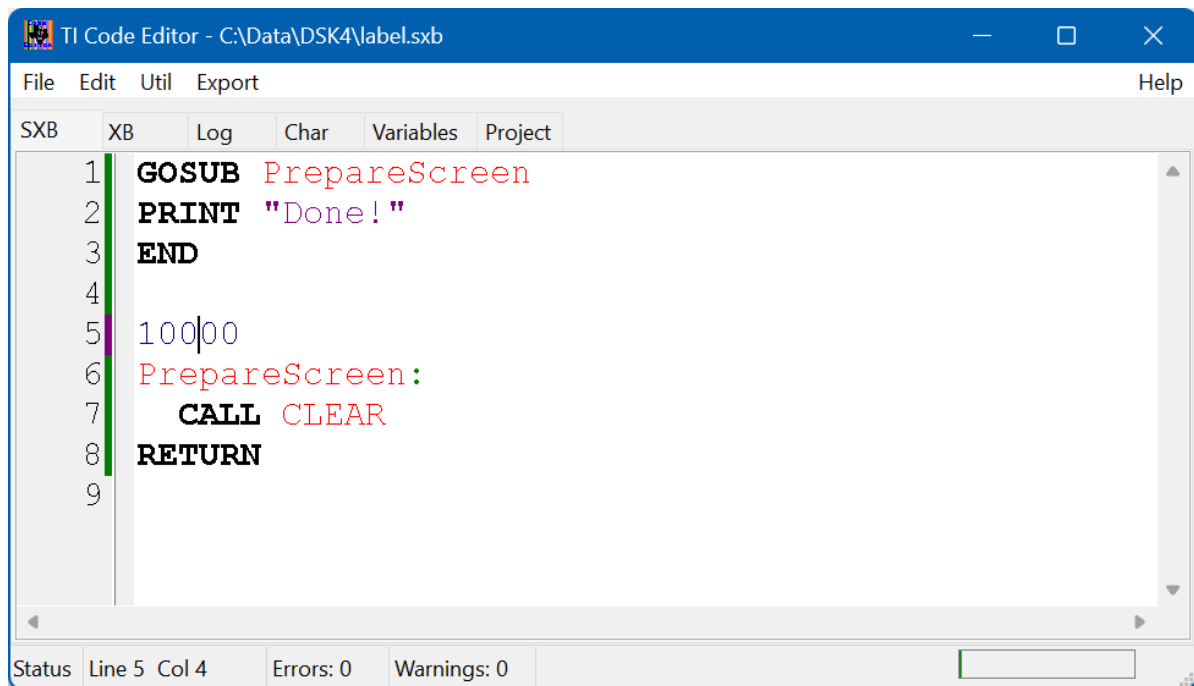
PrepareScreen:
  CALL CLEAR
RETURN
```

Create a new project, paste the code to the SXB tab, save as "label" and build the project by typing Control-B. The XB Tab now reads:

```
100 GOSUB 130
110 PRINT "Done!"
120 END
130 CALL CLEAR
140 RETURN
```

In larger programs it is helpful to manipulate the line numbers to build “blocks of code” in the resulting XB file, i.e. all GOSUB routines starting line 10000, all SUB routines starting 30000.

You can always increase the line number counter by interspersing line numbers in your code, as seen in this example:



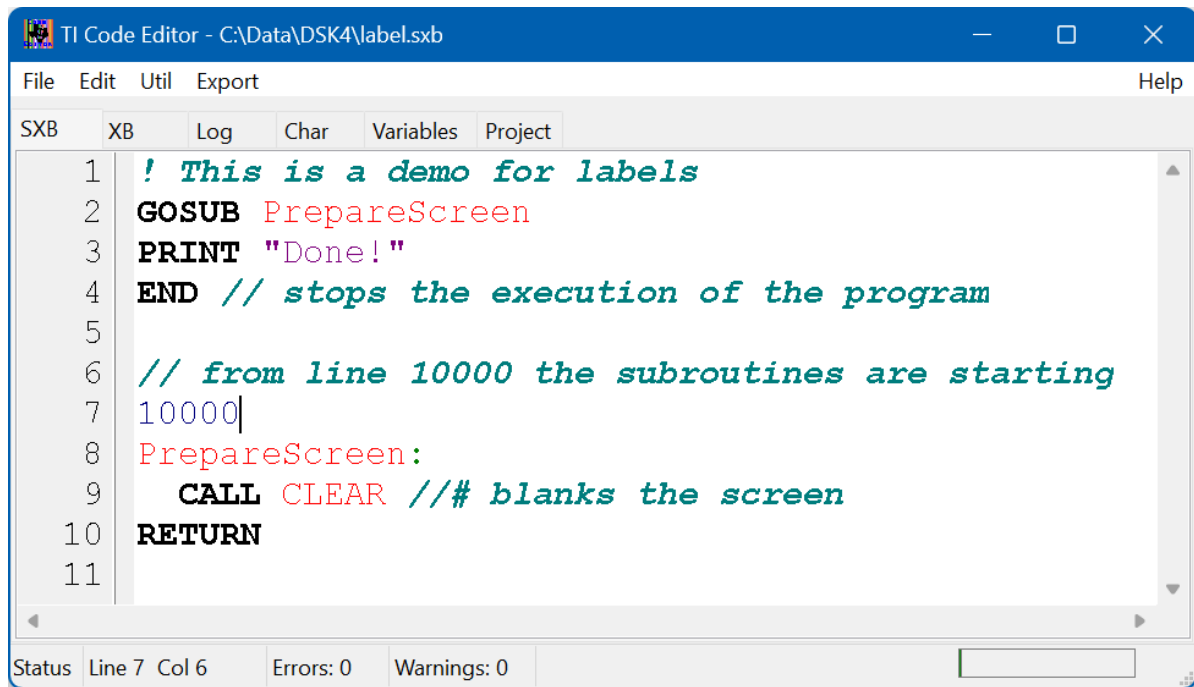
This will result in:

```
100 GOSUB 10000
110 PRINT "Done!"
120 END
10000 CALL CLEAR
10010 RETURN
```

One more “Comment”

Extended BASIC offers only two ways of including comments in your program, the REM statement and the !. Both methods are memory wasters as they store programmer comments in RAM along-side program code. A new third commenting option is added with SXB which avoids wasting sparse memory space inside the TI. Less wasted RAM means more room for program features!

TICodEd is all about cross-development on modern machines with large LCDs. We encourage programming with code indentation for clearer and more-intuitive full-screen editing. Of course adding penalty free comments is always recommended. As seen in contemporary programming languages, SXB introduces the “Double-Slash” comment identifier. Everything following the // will be discarded until the end of the line in the conversion to XB.

The screenshot shows the TI Code Editor window with the title bar 'TI Code Editor - C:\Data\DSK4\label.sxb'. The menu bar includes 'File', 'Edit', 'Util', 'Export', and 'Help'. Below the menu bar are tabs for 'SXB', 'XB', 'Log', 'Char', 'Variables', and 'Project'. The 'SXB' tab is active, displaying the following source code:

```
1  ! This is a demo for labels
2  GOSUB PrepareScreen
3  PRINT "Done!"
4  END // stops the execution of the program
5
6  // from line 10000 the subroutines are starting
7  10000|
8  PrepareScreen:
9      CALL CLEAR // # blanks the screen
10 RETURN
11
```

The status bar at the bottom shows 'Status Line 7 Col 6', 'Errors: 0', and 'Warnings: 0'.

Double-Slash (//) comments will not be retained in the XB listing. Thus, double-slash comments will not take up valuable space in RAM. The resulting XB code will be less-readable without the double-space comments but that doesn't matter since your editable source code is in the SXB tab.

```
100 ! This is a demo for labels
110 GOSUB 10000
120 PRINT "Done!"
130 END
10000 CALL CLEAR
10010 RETURN
```

Note how the traditional !-style comments and REM statements remain intact as they are transferred to the XB-tab code. If you use this style of commenting you will waste valuable RAM.

It is time to put what we have learned about TICodEd to play as we examine SteveB's Structured Extended BASIC *masterpiece*...SteveB52!

Demo Game "SteveB52"

On AtariAge your TiCodEd author goes by the nickname -- SteveB. While thinking of the classic "Bomber" video game style, I was inspired by the classic B52 bomber itself. It was a no-brainer to call my version of the bomber genre **SteveB52**.

The idea of the game is very simple. Your bomber has to land, but there are buildings where you want to land, so you have to clear the runway first. With every flyby you lose some altitude. When dropping bombs in this game there can always be only one bomb falling at a time.

We will develop this game together now. You need to know some Extended BASIC, especially the graphics subroutines, to fully understand it. If you don't know much Extended BASIC you may still follow my explanations to get a rough understanding. If you need a refresh of your Extended BASIC skills, have a look at the [Extended BASIC Reference](#).

The structure of the program

When we think of this game as described above, the fundamental structure of the program begins to materialize. First, we need to do some preparations like defining the shape of the sprites, setting-up the text and the background color, and clearing the screen.

We want to be able to start the game over if we fail or land/win.

We need to prepare the screen with the buildings and let the bomber start.

While in the air we need to manage the user-input, the bomber and the bombs.

When the game is over, either by landing or by hitting a building, we should probably display a clever game over message. After that perhaps we'll ask the player if they wish to start all over or end the program.

```
GOSUB GameInit
REPEAT
  GOSUB PaintScreen
  GOSUB PlayGame
  GOSUB GameOver
UNTIL Answer=78 OR Answer=110
CALL CLEAR
END
```

Done! Game ready! Except for some details of course. This is an example of "structured" programming. Those eight lines define the structure of the program -- top-down. If you want to change something, you immediately know where to go.

The Subroutines

Let's define the first subroutine -- GameInit:

```
GameInit:  
    CALL CHAR(124,"00000080C0E070FFFF070F1C38000000000000000000FEFFC0000000000000")  
    CALL CHAR(128,"28103838381000000000000000000000000000000000000000000000000000")  
    CALL CHAR(132,"FE929292FE9292FFFFFFFF")  
    CALL MAGNIFY(3) :: DIM A(26) :: RANDOMIZE  
RETURN
```

All of the one-time initialization commands go into GameInit. This routine runs only once at the beginning of the first game you play after loading, never in between subsequent games as this is not necessary.

The (between) each-game preparation commands go into PaintScreen:

```
PaintScreen:
    CALL ScrInit(16,2) :: DISPLAY AT(23,10):"SteveB52"
    FOR I=1 to 26 :: A(I)=0 :: NEXT I
    FOR I=1 to 50 :: J=INT(RND*26)+1 :: A(J)=A(J)+1 :: NEXT I
    CALL HCHAR(21,1,133,32)
    FOR I=1 to 26 :: CALL VCHAR(21-A(I),I+3,132,A(I)):: NEXT I
    CALL SPRITE(#1,124,13,1,1,0,16)
RETURN
```

We first call `ScrInit(16,2)`, which is not defined in the program, but in the Standard Library of TiCodEd. We will talk about [libraries later](#), but we need to have the “Standard Library” checked on the Project tab, which is the default.

Libraries

Extension Package

▼

User Library

☐ Lowercase LINK

☒ Standard Library

We display the game name at the bottom of the screen, initialize our array before distributing 50 building elements among the 26 columns in use. We then use HCHAR to draw the ground / runway and draw the buildings on top of it. Finally, we create Sprite #1 in the top left corner and let it fly horizontally with speed 16.

Now we can look at the actual game-play:

```

PlayGame:
  REPEAT
    CALL POSITION(#1,Y1,X1) :: IF X1>240 THEN CALL LOCATE(#1,Y1+4,1)
    CALL KEY(3,K,S) :: IF K=32 AND BOMB=0 THEN CALL SPRITE(#2,128,11,Y1,X1,24,0) :: BOMB=1
    IF BOMB=0 THEN NoBomb
      CALL POSITION(#2,Y2,X2) :: CALL GCHAR(Y2/8+1,X2/8+1,C2)
      IF C2=132 OR Y2>155 THEN CALL DELSPRITE(#2):: BOMB=0
      IF C2=132 THEN CALL HCHAR(Y2/8+1,X2/8+1,32,1) :: CALL SOUND(-250,-7,1)
    NoBomb:
      CALL GCHAR(INT((Y1-1)/8)+2,INT((X1-1)/8)+1,C1)
    UNTIL (Y1>148 AND X1>200) OR C1<>32
  RETURN

```

We execute the game-loop until either we land or hit an obstacle. We use variable Y1 and X1 for the position of the bomber as sprite #1 and Y2 and X2 as position of the bomb as sprite #2. The variable BOMB indicates whether a bomb is already falling(1) or can be dropped(0).

First we query the position of sprite #1, the bomber. If beyond column 240 it is re-inserted at column 1, but 4 pixels below.

Next we query the keyboard. When space (key=32) was pressed and no bomb was already falling, we create a sprite #2 at the position of the bomber and let it move vertically at speed 24.

We skip the next three lines when no bomb is falling. As Extended BASIC knows no blocks of statements like C with { } or Pascal with BEGIN .. END, we still must use a very local GOTO statement to skip those lines and continue at label NoBomb:.

If there is a bomb falling we first query the position of sprite #2. As sprites use pixel coordinates whereas characters use 32x24 character screen-positions, we divide the sprite position by 8 and add 1 to query the character with GCHAR at the position of the bomb in C2. If it hits a building (character 132) or falls below the ground (Y2>155) the bomb gets deleted. If it hits a building, the character gets replaced with a space (32) and an explosion sound will be created.

Finally, we check if the bomber has hit a building with variable C1.

When we leave the loop for either reason, the game is over.

```

GameOver:
  CALL MOTION(#1,0,0) :: CALL DELSPRITE(#2)
  IF C1<>32 THEN DISPLAY AT(8,10):"Game Over" ELSE DISPLAY AT(8,10):"You landed"
  DISPLAY AT(12,8):"Play again? (Y/N)"
  REPEAT
    CALL KEY(3,Answer,S)
  UNTIL S<>0
  RETURN

```

First we stop the bomber and delete the bomb (no need to check if it is really there as CALL DELSPRITE gives no error on a non-existing sprite). By the value of C1 we can decide whether we landed or hit a building. We ask if we want to play again and loop until a key is pressed using the status of CALL KEY.

That is it. Paste all elements in the order of appearance together into the SXB tab after creating a new project and save the game under SteveB52. On the project page we shorten the Token and Merge name to SB52:

Now we build the project by pressing Control-B.

```

100 GOSUB 170
110 GOSUB 220
120 GOSUB 290
130 GOSUB 380
140 IF NOT (Answer=78 OR Answer=110) THEN 110
150 CALL CLEAR
160 END
170 CALL CHAR(124,"00000080C0E070FFFF070F1C380000000000000000FEFFC00000000000")
180 CALL CHAR(128,"281038383810000000000000000000000000000000000000000000000000")
190 CALL CHAR(132,"FE929292FE929292FFFFFFFF")
200 CALL MAGNIFY(3) :: DIM A(26) :: RANDOMIZE
210 RETURN
220 CALL ScrInit(16,2) :: DISPLAY AT(23,10):"SteveB52"
230 FOR I=1 TO 26 :: A(I)=0 :: NEXT I
240 FOR I=1 TO 50 :: J=INT(RND*26)+1 :: A(J)=A(J)+1 :: NEXT I
250 CALL HCHAR(21,1,133,32)
260 FOR I=1 TO 26 :: CALL VCHAR(21-A(I),I+3,132,A(I)):: NEXT I
270 CALL SPRITE(#1,124,13,1,1,0,16) :: BOMB=0
280 RETURN
290 CALL POSITION(#1,Y1,X1) :: IF X1>240 THEN CALL LOCATE(#1,Y1+4,1)
300 CALL KEY(3,K,S) :: IF K=32 AND BOMB=0 THEN CALL SPRITE(#2,128,11,Y1,X1,24,0) :: BOMB=1
310 IF BOMB=0 THEN 350
320 CALL POSITION(#2,Y2,X2) :: CALL GCHAR(Y2/8+1,X2/8+1,C2)
330 IF C2=132 OR Y2>155 THEN CALL DELSPRITE(#2):: BOMB=0
340 IF C2=132 THEN CALL HCHAR(Y2/8+1,X2/8+1,32,1) :: CALL SOUND(-250,-7,1)
350 CALL GCHAR(INT((Y1-1)/8)+2,INT((X1-1)/8)+1,C1)
360 IF NOT ((Y1>148 AND X1>200) OR C1<>32) THEN 290
370 RETURN
380 CALL MOTION(#1,0,0)::CALL DELSPRITE(#2)
390 IF C1<>32 THEN DISPLAY AT(8,10):"Game Over" ELSE DISPLAY AT(8,10):"You landed"
400 DISPLAY AT(12,8):"Play again? (Y/N)"
410 CALL KEY(3,Answer,S)
420 IF NOT (S<>0) THEN 410
430 RETURN
440 SUB ScrInit(fg,bg)
450 CALL SCREEN(bg)

```

```

460 CALL DELSPRITE(ALL)
470 CALL CLEAR
480 FOR I=0 to 14 :: CALL COLOR(I,fg,1) :: NEXT I
490 SUBEND

```

Please note lines 440 to 490. The SUB ScrInit(fg,bg) was pulled from the standard library and appended to the program (see chapter [The Standard Library](#)).

The BEGIN/END-Block

When looking at the PlayGame routine you may felt that the "IF BOMB=0 THEN NoBomb" looks a little bit weird:

```

IF BOMB=0 THEN NoBomb
  CALL POSITION(#2,Y2,X2) :: CALL GCHAR(Y2/8+1,X2/8+1,C2)
  IF C2=132 OR Y2>155 THEN CALL DELSPRITE(#2):: BOMB=0
  IF C2=132 THEN CALL HCHAR(Y2/8+1,X2/8+1,32,1) :: CALL SOUND(-250,-7,1)
NoBomb:
...

```

We need a GOTO to omit some part of the code, when the bomb is falling. While GOSUB keeps our program in shape by returning to the caller, GOTO allows any destination, enabling "Spaghetti-Code", BASIC is infamous for. Extended BASIC already has a limited kind of block-building in the IF-THEN-ELSE Statement. You may have multiple statements in the the THEN and the ELSE branch:

```

IF <cond> THEN <Stmt1> :: <Stmt2> ELSE <Stmt3> :: <Stmt4>

```

The limitation is the line-size. Above example would not fit in a TI XB line. The nested IF/THEN are a second problem. The THEN branch is delimited either by ELSE or the end of the line, the ELSE branch always by the end of the line. There is no ENDIF or similar. The second inner IF needs to start on a own line to be separated from the first.

Structured Extended BASIC therefore offers BEGIN/END blocks in IF-THEN-ELSE statements like you may know them from PASCAL or the {} blocks in C or other languages. They can spread over multiple lines and may even be nested. Let's look at an improved version the PlayGame routine:

```

PlayGame:
  REPEAT
    CALL POSITION(#1,Y1,X1) :: IF X1>240 THEN CALL LOCATE(#1,Y1+4,1)
    CALL KEY(3,K,S) :: IF K=32 AND BOMB=0 THEN CALL SPRITE(#2,128,11,Y1,X1,24,0) :: BOMB=1
    IF BOMB=1 THEN BEGIN
      CALL POSITION(#2,Y2,X2) :: CALL GCHAR(Y2/8+1,X2/8+1,C2)
      IF C2=132 OR Y2>155 THEN CALL DELSPRITE(#2):: BOMB=0
      IF C2=132 THEN CALL HCHAR(Y2/8+1,X2/8+1,32,1) :: CALL SOUND(-250,-7,1)
    END
    CALL GCHAR(INT((Y1-1)/8)+2,INT((X1-1)/8)+1,C1)
  UNTIL (Y1>148 AND X1>200) OR C1<>32
RETURN

```

How can this be translated to standard XB? Obviously, all statements do not fit in an XB line. This is one of the cases, a GOSUB will be used for an “Out of sequence” processing. Remember the “Generated GOSUB” setting on the project tab, with the 20000 default? TiCodEd shifts the block to a GOSUB/RETURN subroutine:

```

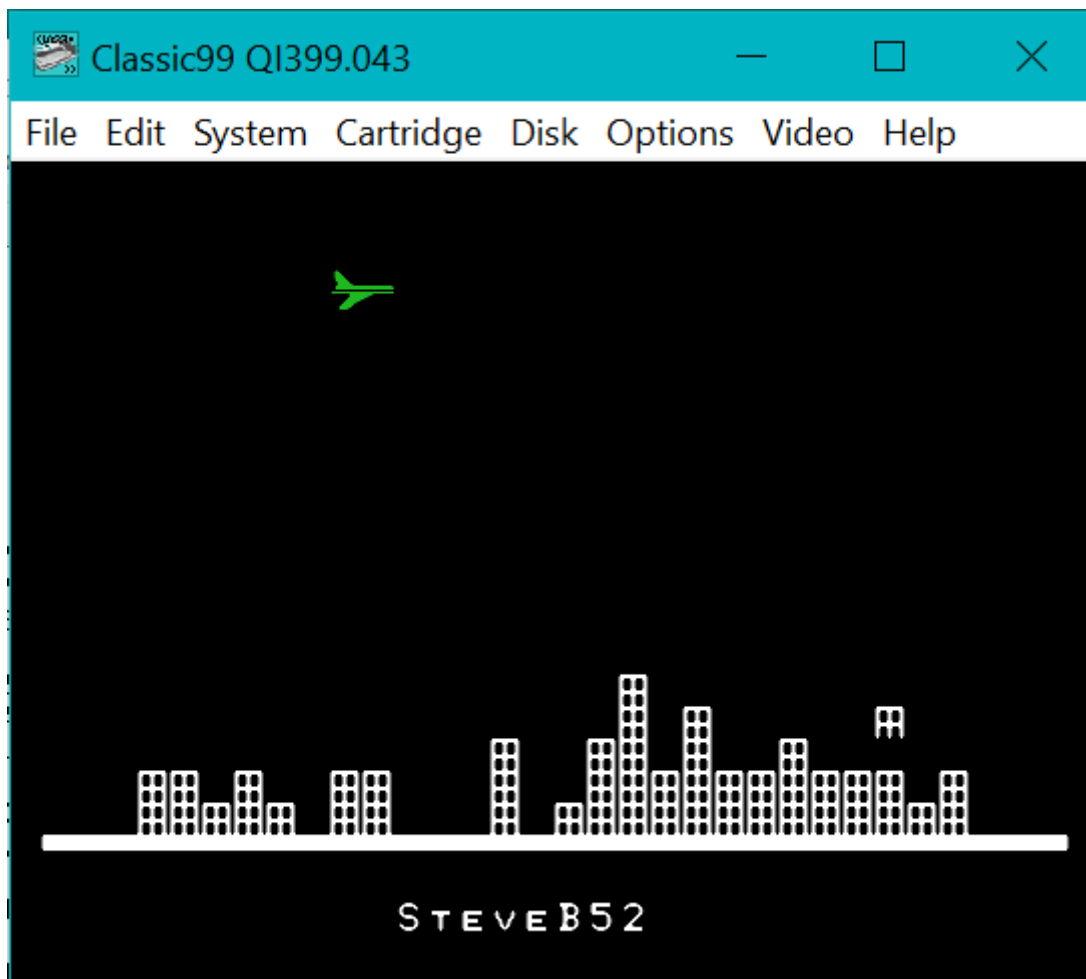
100 GOSUB 170
110 GOSUB 10000
120 GOSUB 10070
130 GOSUB 10130
140 IF NOT (Answer=78 OR Answer=110) THEN 110
150 CALL CLEAR
160 END
170 CALL CHAR(124,"00000080C0E070FFFF070F1C38000000000000000000FEFFC000000000000")
180 CALL CHAR(128,"28103838381000000000000000000000000000000000000000000000000000")
190 CALL CHAR(132,"FE929292FE929292FFFFFF")
200 CALL MAGNIFY(3) :: DIM A(26) :: RANDOMIZE
210 RETURN
10000 CALL ScrInit(16,2) :: DISPLAY AT(23,10):"SteveB52"
10010 FOR I=1 TO 26 :: A(I)=0 :: NEXT I
10020 FOR I=1 TO 50 :: J=INT(RND*26)+1 :: A(J)=A(J)+1 :: NEXT I
10030 CALL HCHAR(21,1,133,32)
10040 FOR I=1 TO 26 :: CALL VCHAR(21-A(I),I+3,132,A(I)):: NEXT I
10050 CALL SPRITE(#1,124,13,1,1,0,16) :: BOMB=0
10060 RETURN
10070 CALL POSITION(#1,Y1,X1) :: IF X1>240 THEN CALL LOCATE(#1,Y1+4,1)
10080 CALL KEY(3,K,S) :: IF K=32 AND BOMB=0 THEN CALL SPRITE(#2,128,11,Y1,X1,24,0) :: BOMB=1
10090 IF BOMB=1 THEN GOSUB 20000
10100 CALL GCHAR(INT((Y1-1)/8)+2,INT((X1-1)/8)+1,C1)
10110 IF NOT ((Y1>148 AND X1>200) OR C1<>32) THEN 10070
10120 RETURN
10130 CALL MOTION(#1,0,0)::CALL DELSPRITE(#2)
10140 IF C1<>32 THEN DISPLAY AT(8,10):"Game Over" ELSE DISPLAY AT(8,10):"You landed"
10150 DISPLAY AT(12,8):"Play again? (Y/N)"
10160 CALL KEY(3,Answer,S)
10170 IF NOT (S<>0) THEN 10160
10180 RETURN
10190 END
20000 CALL POSITION(#2,Y2,X2) :: CALL GCHAR(Y2/8+1,X2/8+1,C2)
20010 IF C2=132 OR Y2>155 THEN CALL DELSPRITE(#2):: BOMB=0
20020 IF C2=132 THEN CALL HCHAR(Y2/8+1,X2/8+1,32,1) :: CALL SOUND(-250,-7,1)
20030 RETURN
30000 SUB ScrInit(fg,bg)
30010 CALL SCREEN(bg)
30020 CALL DELSPRITE(ALL)
30030 CALL CLEAR
30040 FOR I=0 TO 14 :: CALL COLOR(I,fg,1) :: NEXT I
30050 SUBEND

```

In general, the THEN branch or the ELSE branch or both can contain a BEGIN/END block.

```
IF a=b THEN BEGIN
  CALL xx
  a=a+1
END ELSE BEGIN
  CALL yy
  b=b-1
END
```

In the Advanced Topics you will find a more detailed discussion [Using BEGIN-END for code blocks](#).



When playing the game you may notice that some hits get overseen by the program, i.e. a bomb falls through a one-story building or the upper story of a higher building. That is because the TI remains as slow as it was 40 years ago. Your electric toothbrush may have a microcontroller more powerful than the venerable TI. One way to remedy this situation is by turning the emulation speed up. Use Options / CPU Throttling and select CPU Overdrive. This is a weak solution as you can not do this on the real machine.

If only there were a better way?

This is where Harry Wilhelm (aka, Senior Falcon) steps in to save the day!

With Harry's BASIC Language Compiler we are able to translate programs written in BASIC, Extended BASIC, or Structured Extended BASIC, into what is known as "Threaded" Assembly Language code.

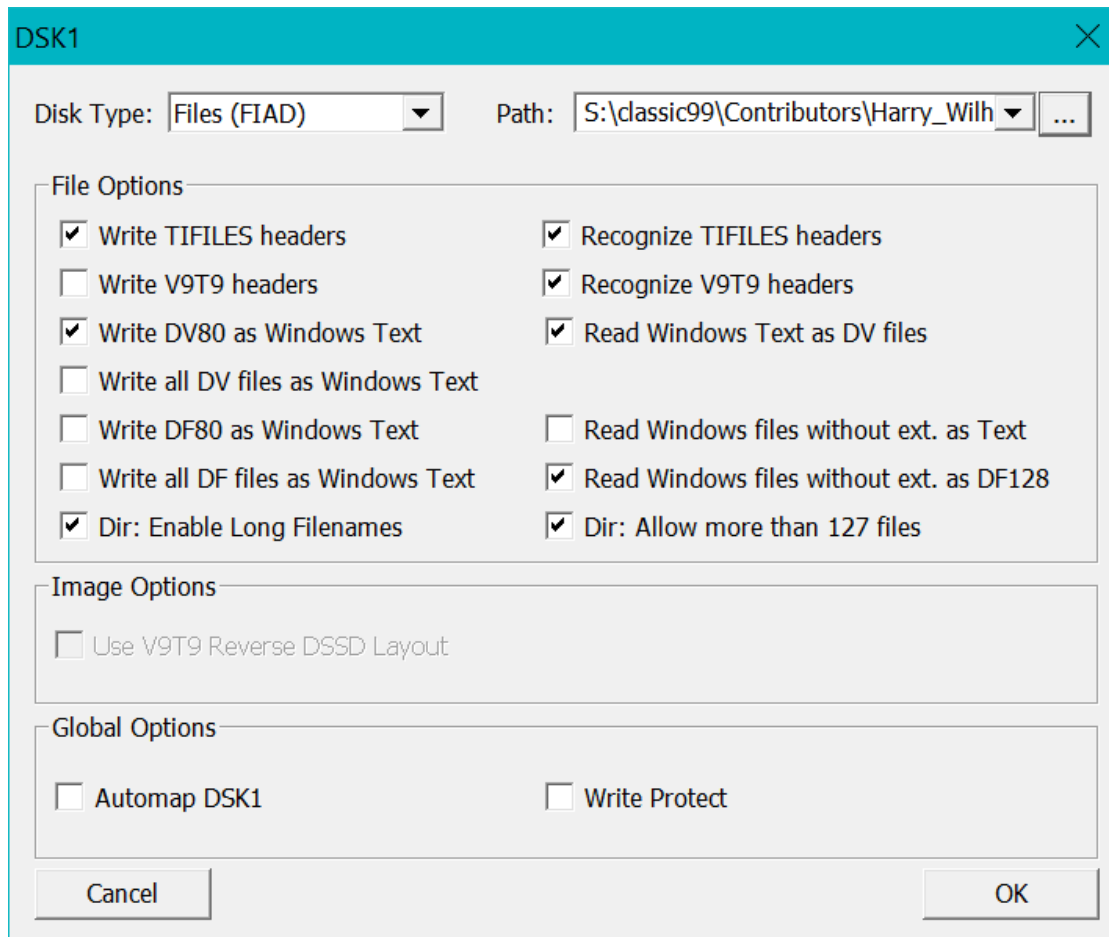
The resulting "Threaded" Assembly Language code can then be "Assembled," which is basically translating the code again into Machine Language. The beauty of the Compiler is you don't have to know Assembly Language to exploit some (upwards of **25X**) speed advantages.

NOTE: Since the BASIC Compiler's conversion process from BASIC to "Threaded" Assembly Language yields code not as optimal as hand-written Assembly Code, the speed advantages of the compiler, though incredibly significant, are no match for pure hand-written Assembly.

That said, running your BASIC programs at up to **25-times** as fast as TI-99/4A normally runs BASIC programs is good! Actually, it's awesome. Some say it's Insanely Great! Just not as good as an Assembly Language programmer can do. With pure hand-written Assembly one can achieve performance somewhere around **200-times** faster than standard TI BASIC.

Get the Compiler ready!

I asked you to not use DSK1 for your program FIAD directory because we want to use DSK1 for the compiler. So please locate the Contributors\Harry_Wilhelm\ISABELLA directory in your Classic99 installation. You will also need the Asm994a.exe from this directory.



When you now start Classic99 and select "2 FOR EXTENDED BASIC" you get the following menu from the autostart file of DSK1 (or an older version of ISABELLA)



We already wrote the SB52-M file as selected on the Project tab, so we can use the arrow-keys to select COMPILER and hit Return.

```

*****
*  EXTENDED BASIC COMPILER  *
*    Harry Wilhelm 2023    *
*****

XB merge file to compile?
DSK-M

```

Overwrite the DSK-M with DSK4.SB52-M and press Return. Confirm the Assembly File as DSK4.SB52.TXT with Return, Select Y for "Assembling with Asm994a" and N to Put Runtime in low Memory.

```

*****
*  EXTENDED BASIC COMPILER  *
*    Harry Wilhelm 2023    *
*****

XB merge file to compile?
DSK4.SB52-M

Assembly file to create?
DSK4.SB52.TXT

Assembling with Asm994a?      Y
Put runtime in low memory?    N
Proceed?                      Y

```

Press Y to Proceed. To speed things up you may switch to CPU Overdrive here as well.

The compiler gives some output about the passes and the lines processed and if everything is fine will quit to the menu.

```

XB Game Developers Package
      "Jewel"

XB256
EXTENDED BASIC
COMPILER
ASSEMBLER
> LOADER

```

Here it skipped the Assembler as we chose to use Asm994a.exe from the JEWEL directory. Asm994a.exe is originally from the Win994a emulator package from BurrSoft (Cory Burr) and included with the compiler by permission. It is

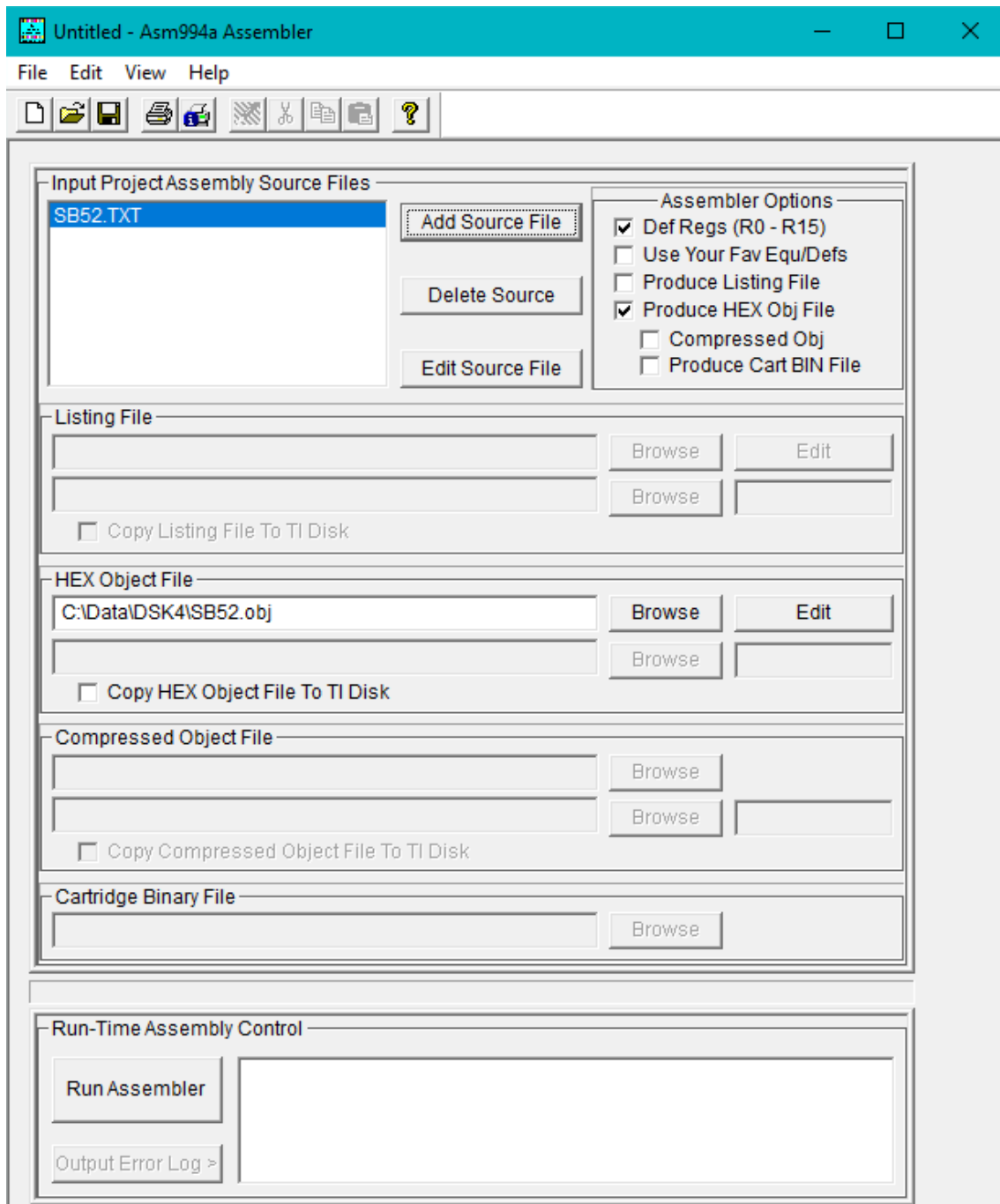
superior to the ASSEMBLER (4th option, old TI Assembler) in this menu for many reasons including processing speed.

We have just completed compiling/converting XB code to threaded Assembly code. The threaded Assembly code is saved as SB52.TXT in your Classic99 DSK4. Next we must Assemble/convert the threaded Assembly code into speedy Machine Language. How exciting, right?

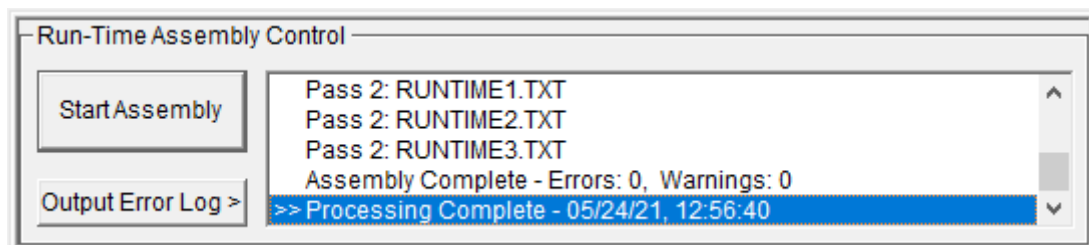
Start the Asm994a.exe program by double-clicking on it in your Windows Explorer in the directory of the compiler (DSK1) and press "Add Source File". You should find SB52.TXT in your DSK4 FIAD Directory. Check the "Def Regs (R0 - R15)" and "Produce HEX Obj File", leave the others unchecked.

The HEX Object File should automatically be set to SB52.OBJ in your DSK4 directory. This is going to be the name of the Machine Language program file.

Finally, press "Run Assembler" at the bottom.



The Output Error Log should show "Assembly Complete - Errors:0, Warnings 0":



Now you can return to Classic99 and select the Loader in the menu:

```

LOAD COMPILED PROGRAM

Using Assembly Support? N
Enter filename to be loaded:
DSK4.SB52.OBJ

Runtime in high memory
16804 bytes remaining

File is Loaded

Press Enter (F4 will bypass)

>CALL LINK("EA5", "DSK4.SB52-E
")
>SAVE DSK4.SB52-X
>RUN■

```

First chose N for not using the assembler support, then confirm the DSK4.SB52.OBJ file with Return, then confirm the CALL LINK statement as well and the proposed SAVE command. With this the menu has auto-created a small XB program with embedded machine code from the assembler.

Turn off CPU overdrive in the Options back to Normal to get the actual speed.

Confirm RUN and enjoy your compiled game!

Did you find this compiling, running the assembler and then linking with the loader cumbersome and time-consuming? TiCodEd offers an integration with Classic99 and Fred Kaal's command-line assembler to completely automate this step. See chapter [Emulator integration and automation](#).

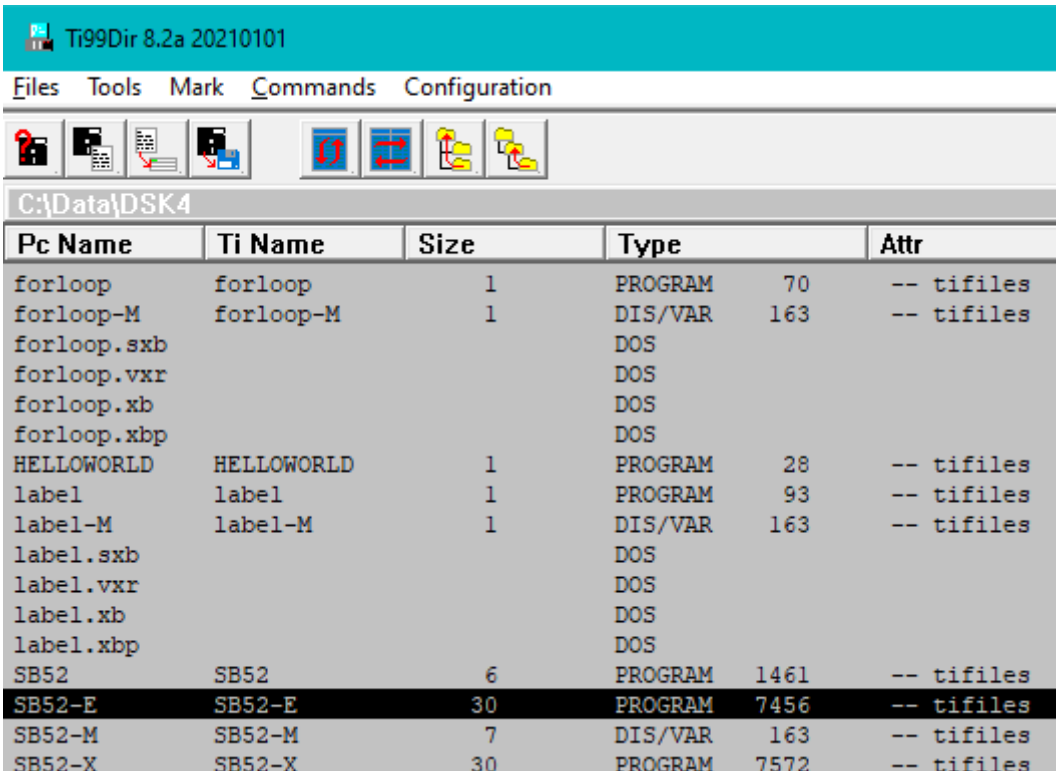
Creating a Module

If you want to play the game as a module in an emulator or on a cartridge like FlashROM99 or FinalGROM you may use Fred Kaals Module Creator. Please follow the install instructions very thoroughly. Especially make sure that there is no space anywhere in the installation path, so "C:\Program Files" will not work.

Additionally, you will need to utilize Fred's "TI99dir" to change the file header from TIFILES to V9T9, or change the settings from DSK4 to "Write V9T9 Headers" before running the linker.

We will assume you have your files in TIFILES format. Start TI99dir and point it to your DSK4 directory by selecting File / Select another Directory.

Select the SB52-E file and go to Tools / Convert TIFILES to V9T9 file. The attribute in the last column changes to V9T9.



Ti99Dir 8.2a 20210101

Files Tools Mark Commands Configuration

C:\Data\DSK4

Pc Name	Ti Name	Size	Type		Attr
forloop	forloop	1	PROGRAM	70	-- tfiles
forloop-M	forloop-M	1	DIS/VAR	163	-- tfiles
forloop.sxb			DOS		
forloop.vxr			DOS		
forloop.xb			DOS		
forloop.xbp			DOS		
HELLOWORLD	HELLOWORLD	1	PROGRAM	28	-- tfiles
label	label	1	PROGRAM	93	-- tfiles
label-M	label-M	1	DIS/VAR	163	-- tfiles
label.sxb			DOS		
label.vxr			DOS		
label.xb			DOS		
label.xbp			DOS		
SB52	SB52	6	PROGRAM	1461	-- tfiles
SB52-E	SB52-E	30	PROGRAM	7456	-- tfiles
SB52-M	SB52-M	7	DIS/VAR	163	-- tfiles
SB52-X	SB52-X	30	PROGRAM	7572	-- tfiles

Changes to:

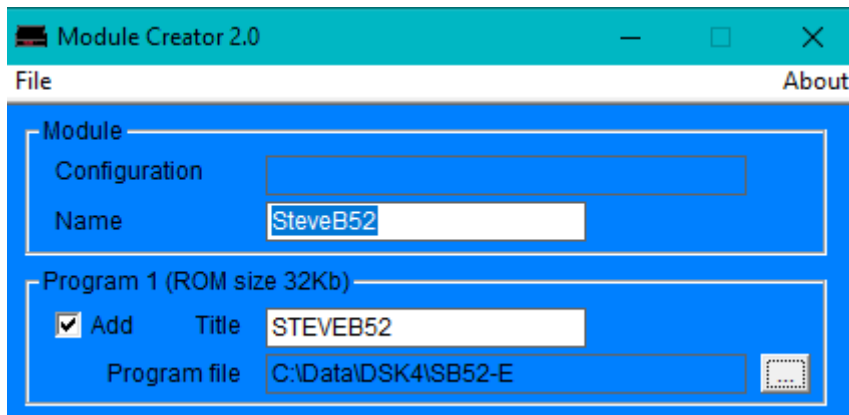
SB52	SB52	6	PROGRAM	1461	-- tfiles
SB52-E	SB52-E	30	PROGRAM	7456	-- v9t9
SB52-M	SB52-M	7	DIS/VAR	163	-- tfiles

The resulting files from the compiler are split in size at 8kB. Larger programs have additional files ending -F and -G, i.e. SB52-F. A 16kB program will have two files (-E and -F) of 8kB each. A 24kB program will have three files: name-E, name-F, and name-G. When using the "Put runtime in low memory" you may also get a name-H file.

NOTE: The largest size program you can write for the TI-99/4A using this workflow is 24kB in total size. You will find your SXB program may slightly exceed 24kB in size as the resulting Machine Language (known also as "Object Code") version will typically be smaller than the SXB version. So compiling and then assembling your SXB programs essentially shrinks them in the process of speeding them up. Double Bonus!

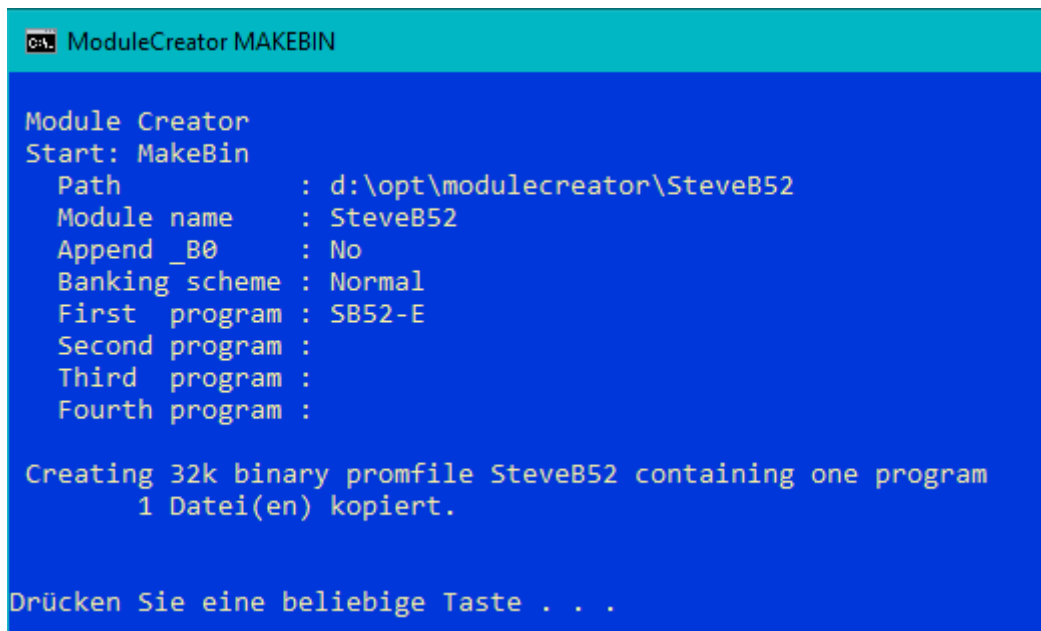
If multiple "object" files are present, convert them the same way to V9T9. You may select multiple files for conversion by pressing the Control-Key while clicking on them.

Now start the **Module Creator 2.0**. First give a name to the Module, then select "Add" in the Program 1 box. Enter the title of the module in the screen after the initial power-up screen. Select the SB52-E file as Program File using the "..." button.



Subsequent -F and -G files are used automatically when present. Program 2 to Program 4 remain empty, both options unchecked. Now press "Create Module" at the bottom.

Multiple windows in different colors will pop up, the last line always indicating "No Errors found".

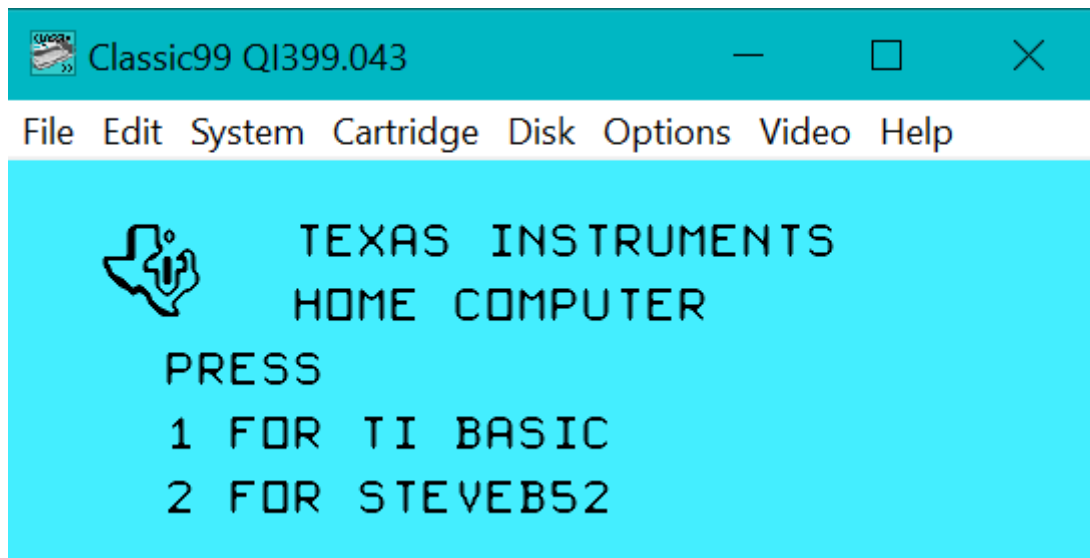


Press any key to close this third and last window to return to the initial menu. You may save your configuration if you want to repeat the module creation for the same program.

You will find your module in the path mentioned above, in the directory of the Module Creator, in a subdirectory with the name of the module. Re-read the previous sentence three times and you'll understand. The module in this case will be in SteveB52 and will be named SteveB52.bin.

NOTE: A .bin file is what is known as a "binary" file, or a "ROM" file. It's machine code formatted in a way which is ready to be loaded into an SD device such as FinalGROM99 or FlashROM99, or loaded into Classic99 as a cartridge.

Try your module by selecting Cartridge / User / Open in Classic99.



You may also try <https://js99er.net/#/> to play your module online.

Perhaps your game is good enough to bring to market? This cartridge .bin file is what you will need to supply Greg at www.arcadeshopper.com if you wish to market your game(s) in the traditional uber-retro TI Solid State Software cartridges.

At this point you have learned to: use TiCodEd to edit programs in SXB on a modern PC, convert the SXB code to threaded Assembly Language using a Compiler, and assemble the threaded assembly code to object code which is native TI-99/4A machine language. Additionally, you now know how to easily produce binary (.bin) files for emulation and online distribution. If making a TI-99-4A game cartridge has been a life-long goal of yours, now you have the tools to make that dream a reality.

Language Reference

Considerations for Compiling

Please read the documentation of Harry Wilhelm's BASIC Compiler Jewel at least once for details. Here are some quick reminders.

- The compiler only allows for 16bit integers (-32768 to 32767) for numeric variables, no floating point arithmetic available
- RND has a special handling: Multiply with n gets a number from 0 to n-1
- Use INT() in XB to get the same results as when compiled
- Files need to have #1, #2 or #3 and are limited to DV1 to DV254
- INPUT requires a prompt when using multiple variables
- PRINT and DISPLAY have limited formatting capabilities ("USING").
- SUB statement is restricted to only six significant characters and there is a huge list of not allowed names in the manual. TiCodEd checks this and gives warnings.
- Test your program in CPU Overdrive mode before compiling, as there are a few runtime checks in the compiled program.

Extended BASIC Reference

This List is based on the original Texas Instruments Extended BASIC Reference Card supplied with the module. If you need more details, check out the [Book "TI Extended BASIC"](#). Commands are meant to be issued interactively and therefore not relevant for TiCodEd and SXB. Functions and statements not supported by the JEWEL Compiler are marked red, yellow with limited support. Refer to the compiler manual for details.

C : COMMAND **F** : FUNCTION **S** : STATEMENT

ABS (numeric expression)
returns the absolute value of
numeric-expression **F**

ACCEPT [[AT(row, column)] VALIDATE
(datatype, ...)] [BEEP] [ERASE ALL] (
SIZE(numeric-expression)] :]variable
suspends program execution until data is
entered from the keyboard . Optionally , data is
entered at the position specified by row and
column , the data is validated , and / or
option(s) are executed . **S , C**
VALIDATE datatypes:
UALPHA permits all uppercase alphabetic char.
DIGIT permits 0 through 9.
NUMERIC permits 0 through 9, ".", "+", "-", &
"E".

String-expression permits the characters
contained in string-expression.

BEEP: causes an audible tone

ERASE ALL : places the space character in all
screen positions before accepting input.

SIZE (numeric-expression) : allows only
numeric expression characters to be entered. If
numeric expression is positive , that many
positions are blanked . If it is negative , no
positions are blanked.

ASC(string-expression)
returns the ASCII code of the first character of

string-expression. **F**

ATN(numeric-expression)
returns the trigonometric arctangent of numeric
expression. **F**

BREAK [Line-number-list]
causes the program to halt when encountered
or optionally when lines in line-number-list are
encountered. **C,S**

BYE
closes open files and leaves TI Extended BASIC.
C

CALL subprogram [(parameter-list)]
calls the indicated subprogram. An optional
parameter-list can be passed . **S**

CALL CHAR(character-code, pattern-identifier [,
...])
defines the specified ASCII character code(s)
using a 0 through 64 character hexadecimal
coded string pattern-identifier. **S,C**

CALL CHARPAT(character-code ,
string-variable[,...])
returns in string-variable the hexadecimal code
that
specifies the pattern of character-code. **S,C**

CALL CHARSET

restores the standard character patterns and colors for characters 32 through 95. **S,C**

CHR\$(numeric-expression)
returns the string character corresponding to the ASCII numeric-expression. **F**

CALL CLEAR

places the space character in all screen positions. **S,C**

CLOSE #file-number [:DELETE]

stops the programs use of the file referenced by # file number and optionally deletes the file. **S,C**

CALL COINC(#sprite-number,#sprite-number, tolerance, numeric-variable)

CALL

COINC(#sprite-number,dot-row,dot-column, tolerance,numeric-variable)

CALL COINC(ALL,numeric-variable)

returns in numeric-variable -1 if there is a coincidence and 0 if there is no coincidence. If ALL is present , a coincidence of any two sprites is reported. If two sprites are identified by number, their coincidence is reported. If a sprite and a position are identified their coincidence is reported. Except when ALL is specified , a tolerance of from 0 to 255 is specified. The distance between the given sprites or sprite and position must be less than tolerance for a coincidence to be reported . **S,C**

CALL COLOR(#sprite-number, foreground-color [, ...])

CALL COLOR(character-set, foreground-color, background color [, ...])

specifies either a color for # sprite-number or a Foreground-color and background-color for characters in character-set. **S,C**

CONTINUE

CON

resumes execution after a break . **C**

COS(radian-expression)

returns the trigonometric cosine of radian-expression **F**

DATA data-list

stores numeric and string constant data in program. **S**

DEF function-name [(parameter)] = expression
associates user-defined numeric or string expression with function-name **S**

DELETE device-filename

removes filename from device. **C,S**

CALL DELSPRITE(#sprite-number [,...])

CALL DELSPRITE (ALL)

removes the specified sprite (s) from the screen. ALL removes all sprites from the screen. **S,C**

DIM array-name(integer1 [,integer2] .. [integer7])

dimensions the listed array (s) as specified. **S,C**

DISPLAY [[AT(row,column)] [BEEP] [ERASE ALL] [SIZE(numeric-expression)] :] variable-list
ransfers variable-list to the display screen.

Optionally, data is displayed at the position specified

by row and column. **S,C**

Options:

BEEP : causes an audible tone.

ERASE ALL : places the space character in all screen positions before displaying

SIZE(numeric expression):blanks numeric-expression characters in the indicated position

DISPLAY [option-list:] USING string-expression [:variable-list]

DISPLAY [option-list:] USING line-number[: variable-list]

has the same options as DISPLAY with the addition of the USING clause , which specifies the format. If

string-expression is present , it defines the format. If

line - number is present , it refers to the line number of IMAGE statement. See IMAGE **S,C**

CALL DISTANCE(#sprite-number, #sprite-number, numeric-variable)

CALL DISTANCE(#sprite-number, dot-row, dot-column, numeric-variable)

returns in numeric-variable the square of the distance between the sprites or the sprite and the location. **S,C**

END

terminates program execution. **S**

EOF(file-number)

returns the end-of-file condition of file-number **F**

0 : not end-of-file

1 : logical end-of-file

-1 : physical end-of-file

CALL ERR(error-code, error-type [, error-severity , line number])

returns the error-code and error-type of the most recent uncleared error . Optionally , returns the error-severity and line-number in which the error occurred. **S,C**

Error-code : consult manual

Error-type : Negative number : execution error.

Positive number: number of file in which the error occurred

Error-severity : 9, indicating that the error is not recoverable.

EXP(numeric-expression)

returns exponential value (e^x) of numeric-expression

The value of e is 2.718281828. **F**

FOR control-variable = initial-value TO Limit [STEP increment]

repeats execution of statements between FOR and NEXT until the control-variable exceeds the limit. STEP increment default is one. **S,C**

CALL GCHAR(row, column, numeric-variable)

returns in numeric-variable the ASCII code of the character located at row and column. **S,C**

GOSUB line-number

GO SUB line-number

transfers control to a subroutine at line-number **S**

GOTO line-number

GO TO line-number

unconditionally transfers control to line-number.
S

CALL

HCHAR(row,column,character-code[,repetition]
)
places the ASCII pattern of character-code at row and column and optionally repeats it repetition times horizontally. **S,C**

IF relational-expression THEN line-number1
[ELSE line-number2]

IF relational-expression THEN statement1 [ELSE
statement2]

IF numeric-expression THEN line-number1
[ELSE line-number2]

IF numeric-expression THEN statement1 [ELSE
statement2]
transfers control to line-number1 or performs
statement1 if relational-expression is true or
numeric-expression is not equal to zero.
Otherwise control passes to the next statement,
or optionally to line-number2 or statement2. **S**

IMAGE format-string

specifies the format in which data is PRINTed or
DISPLAYed when the USING clause is present
Format-string may be any or all of the following:
Letters, numbers . characters not listed below
transferred directly
: replaced by the print-list values given in
PRINT or DISPLAY
^ : replaced by the E and power numbers. Must
be four or five of these **S**

CALL INIT

prepares the computer to load and run
assembly language subprograms **S,C**

INPUT [input-prompt:] variable-list
suspends program execution until data is
entered from the keyboard . The optional
input-prompt may indicate what data is
expected. **S**

INPUT #file-number[,REC record-number]
variable-list
assigns data from the indicated file to the
variables in variable-list. Records are read
sequentially unless the optional REC clause is
used. **S**

INT(numeric-expression)
returns the greatest integer less than or equal
to numeric-expression. **F**

CALL JOYST (key-unit, x-return, y-return)
accepts data into x-return and y-return based
on the position of the joystick labeled key-unit
Values are -4, 0 and 4. **S,C**

CALL KEY(key-unit, return-variable,
status-variable)
assigns the code of the key pressed on key-unit
(0 to 5) to return-variable Status information
is returned in status-variable. 1 means a new
key was pressed. -1 means the same key was
pressed.
0 means no key was pressed . **S,C**

LEN(string-expression)
returns the number of characters in
string-expression. **F**

[LET] numeric-variable [, numeric-variable, ...]
= numeric-expression

[LET] string-variable [, string-variable , ...] =
string-expression
assigns the value of an expression to the
specified
variable(s). **S,C**

CALL LINK(subprogram-name [,
argument-list])
passes control to an assembly language
subprogram. **S,C**

INPUT [[#file-number [, REC
record-number]:] string-variable

INPUT [input-prompt:] string-variable
assigns data from the indicated file to
string-variable
or suspends program execution until data is
entered from the keyboard. If data is assigned
from a file
records are read sequentially unless the optional
REC clause is used . If data is entered from the
keyboard the optional input-prompt may
indicate what data is expected . **S**

LIST ["device-name":] [line-number]

LIST ["device-name":] [start-line-number] -
[end-line number]

sequentially displays program statements or
optionally a single line number or all lines
between specified line numbers. **C**

CALL LOAD("access-name"[,address,byte1 [...
] ,file-field, ...])
loads an assembly language subprogram. **S,C**

CALL

LOCATE(#sprite-number,dot-row,dot-column [,
....])
moves the given sprite (s) to the given
dot-row (s) and dot-column(s). **S,C**

LOG(numeric-expression) returns the natural
logarithm of numeric-expression **F**

CALL MAGNIFY(magnification – factor)
sets the size and magnification of all sprites.
S,C

Magnification – factors:

- 1 : single size unmagnified
- 2 : single size magnified
- 3 : double size unmagnified
- 4 : double size magnified

MAX(numeric-expression1,
numeric-expression2)
returns the larger of numeric-expressio1 and
numeric-expression2. **F**

MERGE [""] device-filename [""]
merges lines in filename from the given device
into the program lines already in the computer's
memory **C**

MIN(numeric-expression1,
numeric-expression2) returns the smaller of
numeric-expression1 and numeric-expression2
F

CALL MOTION(#sprite-number,row-velocity,
column-velocity [...]) changes the motion of a
sprite(s) to the indicated row-velocity and
column-velocity. **S,C**

NEW clears the memory and screen and

prepares for a new program. **C**

NEXT control-variable See FOR statement **S,C**

NUMBER [initial-line] [, increment]

NUM [initial-line] [, increment]

generates sequenced line numbers starting at 100 in increments of 10. Optionally, you may specify the initial-line and / or increment. **C**

OLD ["device-program-name"] loads program-name from device into memory. **C**

ON BREAK STOP

ON BREAK NEXT

determines the action taken if a breakpoint is encountered either in the program or by SHIFT C (CLEAR). The default is STOP, which halts execution of the program. The keyword NEXT causes breakpoints to be ignored and execution of the program to continue. **S**

ON ERROR STOP

ON ERROR line-number

determines the action taken if an error occurs. The default is STOP, which halts execution of the program. If line-number is given, control is transferred to it when an error occurs. See RETURN. **S**

ON numeric-expression **GOSUB** line-number [, ...]

ON numeric-expression **GO SUB** line-number [, ...]

transfers control to the subroutine with a beginning line number in the position corresponding to the value of numeric-expression. **S**

ON numeric-expression **GOTO** line-number [, ...]

ON numeric-expression **GO TO** line-number [, ...]

unconditionally transfers control to the line number in the position corresponding to the value of numeric expression. **S**

ON WARNING PRINT

ON WARNING STOP

ON WARNING NEXT

determines the action taken if a warning condition occurs. The default is PRINT, which prints a message and continues with the program. The keyword STOP causes the warning message to be printed and execution of the program to stop. The keyword NEXT causes no message to be printed and the program to continue. **S**

OPEN #file-number:"device-filename" [file-organization] [file-type] [open-mode] [record-type]

enables the program to use the given filename. **S,C**

File-number: 0-255

File-organization: RELATIVE or SEQUENTIAL

File-type: DISPLAY or INTERNAL

Open-mode: INPUT, OUTPUT, UPDATE, APPEND

Record-type: FIXED or VARIABLE

OPTION BASE 0

OPTION BASE 1

sets the lowest allowable subscript of arrays to zero or one. The default is zero. **S**

CALL PATTERN(#sprite-number,character-value [,...])

changes the pattern number of the specified sprite(s) to the specified character-value(s). **S,C**

CALL PEEK(address, numeric-variable-list)

returns values in numeric-variable-list corresponding to the values in address. **S,C**

PI returns the value of pi as 3.14159265359 **F**

POS(string1, string2, numeric-expression)

returns the position of the first occurrence of string2 in string1. Search begins at the position specified by numeric expression. Returns zero if no match is found. **F**

CALL

POSITION(#sprite-number,dot-row,dot-column [, ...]) returns the positions in the given dot-row(s) and dot-column(s) of the specified sprite(s). **S,C**

PRINT [#file-number [, REC record-number]:][print-list]

transfers optional print-list to the display screen or optionally to an external file. The REC clause directs print-list to the specified record-number. **S,C**

PRINT [#file-number [, REC record-number]] **USING** string-expression:print-list

PRINT [#file-number [, REC record-number]] **USING** line-number:print-list

acts the same as PRINT with the addition of the USING clause, which specifies the format. If string-expression is present, it defines the format. If line-number is present, it refers to the line number of an IMAGE statement. See IMAGE. **S,C**

RANDOMIZE [numeric-expression]

resets the random number generator to an unpredictable sequence. With optional numeric-expression, the sequence is repeatable. **S,C**

READ variable-list

assigns numeric and string constants from DATA statements to variable list. **S,C**

REC(File-number)

returns the current record position in file-number **F**

REM character-string

indicates internal program documentation with no effect on program execution. **S,C**

RESEQUENCE [initial-line] [, increment]

RES [initial-line] [, increment]

automatically rennumbers lines starting at 100 in increments of 10. Optionally, you may specify the initial-line and / or increment. **C**

RESTORE [line-number]

indicates that the next READ operation will take data from the first DATA statement in the program or optionally, from the first DATA statement after line number **S,C**

RESTORE #file-number [, REC record-number]

resets file pointer to the beginning of the file or optionally, to record-number. **S,C**

RETURN

transfers program control from a subroutine to the statement following the corresponding

GOSUB or ON ... GOSUB statement **S**

RETURN [line-number]

RETURN [NEXT]

controls program action after an error has occurred when an ON ERROR statement has been executed . With nothing following it, returns control to the statement which caused the error and executes it again. Followed by a line-number, it transfers control to the given line. Followed by NEXT, it transfers control to the statement after the one in which the error occurred. **S**

RND

generates a pseudo-random number greater than or equal to zero and less than one. **F**

RPT\$(string-expression, numeric-expression)

returns numeric-expression occurrences of string-expression concatenated together. **F**

RUN ["device.program-name"]

RUN [line-number]

starts program execution at the lowest program statement of the program currently in memory. Optionally program-name is loaded from device or execution starts at line-number. **C,S**

SAVE device.program-name [,PROTECTED]

SAVE device.program-name [,MERGE]

places a copy of the current program in device as program-name PROTECTED makes it impossible to change or list the program later. MERGE enables later merging of the program with another program. See MERGE. **C**

CALL SAY(word-string [, direct-string] [, ...])

causes the speech synthesizer to speak the given word-string or direct-string. **S,C**

CALL SCREEN(color-code)

changes the screen color to color-code. **S,C**

SEG\$(string-expression, position, length)

returns a substring of string-expression beginning at position and extending for length characters. **F**

SGN(numeric-expression)

returns 1 if numeric-expression is positive, 0 if it is zero, and -1 if it is negative. **F**

SIN(radian-expression) returns the trigonometric sine of radian-expression. **F**

SIZE

displays on the screen the number of unused bytes of memory. **C**

CALL SOUND(duration, frequency1, volume1, [, ..., frequency4, volume4]) **S,C**

controls up to three tone and one noise generators. Tone and noise parameters can occur in any order. Negative duration causes immediate sound update

Duration: 1 through 4250 ms, -4250 through -1 ms

Frequency : 110 through 44733 Hz for tone, -1 through -8 for noise .

Volume : 0 (loudest) through 30 (softest).

CALL SPGET(word-string , return-string)

returns in return-string the speech bit pattern that corresponds to word-string . **S,C**

CALL SPRITE(#sprite-number, character-value, sprite-color, dot-row , dot-column[, row-velocity, column-velocity][, ...])

specifies the existence of sprite(s) sprite-number with a pattern specified by character-value , a color of sprite-color, a screen position of dot-row and dot column, and optionally a velocity of row-velocity and column-velocity. **S,C**

SQR(numeric-expression)

returns the square root of numeric-expression. **F**

STOP

terminates program execution. **S,C**

STR\$(numeric-expression)

converts the value of numeric-expression to a string. **F**

SUB subprogram-name [(parameter-list)]

indicates the beginning of subprogram-name with optional parameter-list. **S**

SUBEND

indicates the end of a subprogram and transfers program control from a subprogram to the statement following the CALL statement. **S**

SUBEXIT

transfers program control from a subprogram to the statement following the CALL statement. **S**

TAB(numeric-expression)

controls column position of the output from a PRINT or DISPLAY statement. **F**

TAN(radian-expression)

returns the trigonometric tangent of radian-expression. **F**

TRACE **C,S**

lists line numbers of lines before each is executed.

UNBREAK [line-list]

removes all breakpoints or optionally those in line-list. **C,S**

UNTRACE

cancels the action of the TRACE command. **C,S**

VAL(string-expression) **F**

converts string-expression into a numeric constant.

CALL VCHAR(row,column,character-code [,...])

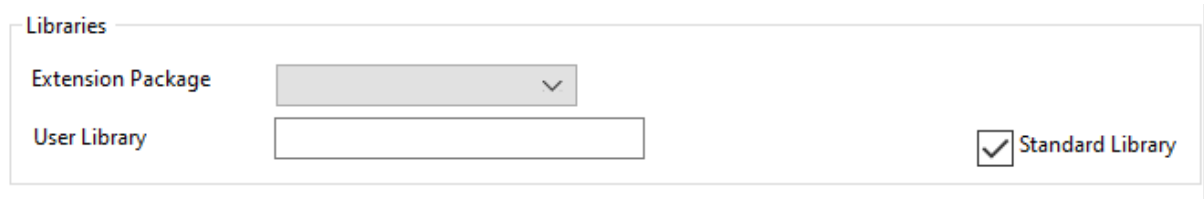
places the ASCII representation of character-code at row and column and optionally repeats it repetition times vertically. **C,S**

CALL VERSION(numeric-variable)

returns a value indicating the version of BASIC being used. TI Extended BASIC returns a value of 110. **C,S**

Additional Libraries

With libraries you are able to reuse code and import additional features.



The screenshot shows a 'Libraries' dialog box. It has a title bar with the word 'Libraries'. Inside, there is a label 'Extension Package' next to a dropdown menu. Below that is a label 'User Library' next to a text input field. To the right of the input field is a checked checkbox with the label 'Standard Library'.

There are three different kinds of libraries.

The Standard Library

In the game SteveB52 we already used the `CALL ScrInit(Fg,Bg)` subroutine which is neither part of Extended BASIC nor defined in our program. It is derived from the "Standard Library", which is enabled by default. The Standard Library is still under development and currently contains the following subroutines:

CALL ScrInit(fg,bg) Clears the screen, deletes all sprites, sets a background color bg and all chars in foreground color fg with a transparent background.

CALL RAND(SEED,UL,RES) Returns an integer $0 \leq \text{RES} < \text{UL}$ and advances the seed. Useful when you want a repeatable sequence, but be aware that neighbor seeds will compute similar results, while the seed is updated with distinct values.

CALL CreateQ(A\$,L) Initializes a Queue with the length of L (max 84) and stores it in the string a\$. Each queue entry consists of three byte, the last byte of the string is the current entry.

CALL enQ(a\$,c,p1,p2,d) Adds a record to the first free entry of the queue d steps ahead of the current position (use 1 for the next available) with the command c and the parameters p1 and p2 (values 1-255 for command, 0-255 for parameters). If the queue is full a\$ is set to 'full' and appropriate actions should be taken.

CALL deQ(a\$,c,p1,p2) Gets next entry from the queue, command 0 means empty slot.

CALL trim(a\$) Removes leading and trailing spaces and unprintable characters.

CALL upStr(a\$) Converts a\$ to uppercase characters.

CALL loStr(a\$) Converts a\$ to lowercase characters.

CALL Mod(n,d,m) Calculates the modulo $n \text{ MOD } d$ (Remainder n/d). Results may differ when compiled if n or d are no integer values.

Subroutines in the Standard Library can be used without prior declaration. The definition will be appended to your XB source code automatically as shown in the SteveB52 example.

The User Library

You can create your own set of procedures you frequently use. The user Library is actually a text-file containing Extended BASIC. Only the parts between SUB/SUBEND or ASM/ASMEND will be used, so you may use any XB program you have as a User Library and use the included SUB routines in other programs or include tests for your routines in the same file.

Packages

Packages are declaring additional functionality. One purpose is to declare additional routines that are in your version of Extended BASIC, but not in the Standard Extended BASIC module. TiCodEd checks for used subroutines with every CALL statement. If it is not an XB standard routine and can't find the routine as SUB in your program and not in the User or Standard Library it issues an error message. When you use an enhanced version of Extended BASIC like XB 2.9 or RXB those additional routines are built-in and only need to be added to the list of internal subroutines.

For RXB 2023 the following package is included in the LIB directory as RXB 2023.xbpkg:

```
// Declares additional CALL Routines of RXB 2023 as internal  
  
INTERNAL: ALPHALOCK BEEP BIAS BYE CAT CLEARPRINT CLSALL COLLIDE DIR EA EALR EXE EXECUTE  
INTERNAL: GMOTION HEX HGET HONK HPUT INVERSE IO ISROFF ISRON JOYLOCATE JOYMOTION  
INTERNAL: MOD MOVES NEW ONKEY PEEKG PEEKV PLOAD POKEG POKER POKEV PRAM PSAVE  
INTERNAL: QUITOFF QUITON RMOTION ROLLDOWN ROLLLEFT ROLLRIGHT ROLLUP SAMS SCROLLDOWN  
INTERNAL: SCROLLLEFT SCROLLRIGHT SCROLLUP SWAPCHAR SWAPCOLOR USER VDPSTACK VGET VPUT XB
```

All those additional routines can be used without error messages when selecting this package.

When you have a not yet supported version of XB just copy this file and change the list of subroutines after the INTERNAL: keyword. Keep the extension xbpkg.

The second use of packages is the use of assembly routines like T40XB or XB256. They provide CALL LINK routines like CALL LINK("COLOR2",81,12,1). Wouldn't it look better if we have CALL COLOR2(81,12,1) instead? This type of package can do the trick.

Before we look into the details, let's have a look at an easy example, which could be the beginning of an Arcade game. Feel free to experiment.

First create a new project and copy the following program to the SXB tab. Save it under the name lscroll.

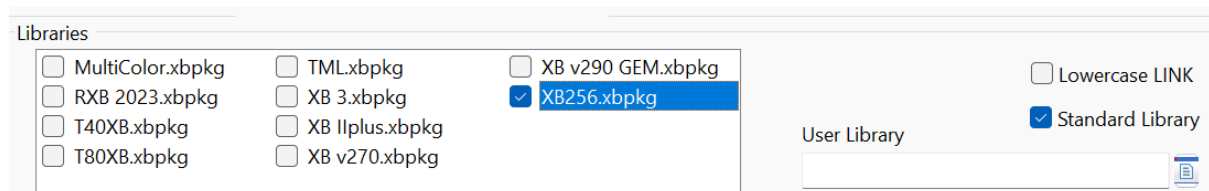
```

GOSUB GameInit
REPEAT
  A=A+INT(RND*5)-2 :: IF A<2 THEN A=2 ELSE IF A>8 THEN A=8
  B=B+INT(RND*5)-2 :: IF B<2 THEN B=2 ELSE IF B>8 THEN B=8
  CALL SCROLLF
  CALL VCHAR(1,32,1,A) :: CALL VCHAR(21-B,32,1,B)
  CALL KEY(3,K,S)
  D = 8 *((K=69)-(K=88))
  CALL MOTION(#1,D,0)
UNTIL K=13
END

GameInit:
  CALL SCRN2
  CALL SCREEN2(2)
  CALL COLOR2(81,12,1)
  CALL CHAR(124,"0000C0F3FF3F3FFF3F3FFFF3C0000000000000000C0FCFFFC")
  CALL CHAR2(1,"FF818181818181FF")
  CALL HCHAR(1,1,1,32)::CALL HCHAR(20,1,1,32)
  CALL MAGNIFY(3) :: RANDOMIZE
  CALL SPRITE(#1,124,10,100,16)
  A,B = 3
RETURN

```

Go to the Project Tab and select XB256.xbpkg from the Extension Package Library list:



Now let's take a look at the program. XB256 offers some very nice graphic features, especially a second screen with 256 additional characters and nice scrolling routines. The GameInit routine first switches to the second screen with CALL SCRN2. This gets translated to the CALL LINK("SCRN2") you will find in the XB256 documentation. SCREEN2 is used for the background color of the second screen and COLOR2 for the characters on screen 2. 81 is a synonym for "all character sets", just as described in the XB256 manual. CHAR2 defines character 1 of the second screen. No need to redefine the standard characters 32 to 127 anymore, there are plenty of characters left outside that range. CALL HCHAR works on the current screen, which we set to the second screen in the first statement of the routine.

A and B are variables for the upper and lower "wall" of the tunnel we are about to build. Now for the actual loop (not a game loop yet). First we change A and B

randomly but make sure they stay within the bounds of 2 to 8. We use CALL SCROLL for "Scroll Left" one column. Now we use CALL VCHAR to fill the new column on the right side.

We then query the keyboard in mode 3, which gives us uppercase letters independent of Alpha-Lock or Shift, and calculate the vertical motion of our spaceship.

$$D = 8 * ((K=69) - (K=88))$$

That's a nifty shortcut only few people know and use. Boolean expressions are evaluated to either 0 for false and -1 for true. So the first inner parentheses calculates -1 if we press "E" (ASCII 69) and 0 if it is not. The second does the same with "X" (ASCII 88), but is subtracted, resulting in +1 or 0. Now multiplied with 8 you get:

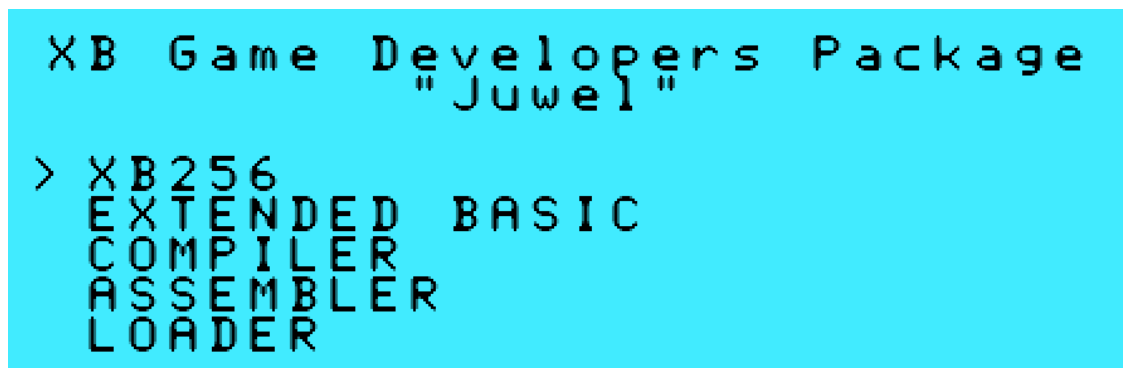
K=69 → D=-8

K=88 → D=8

K other → D=0

Just what we need to control our sprite vertically in the CALL MOTION. You may even omit the variable and enter the boolean term directly in the CALL MOTION. This is standard XB and remains unchanged by TiCodEd.

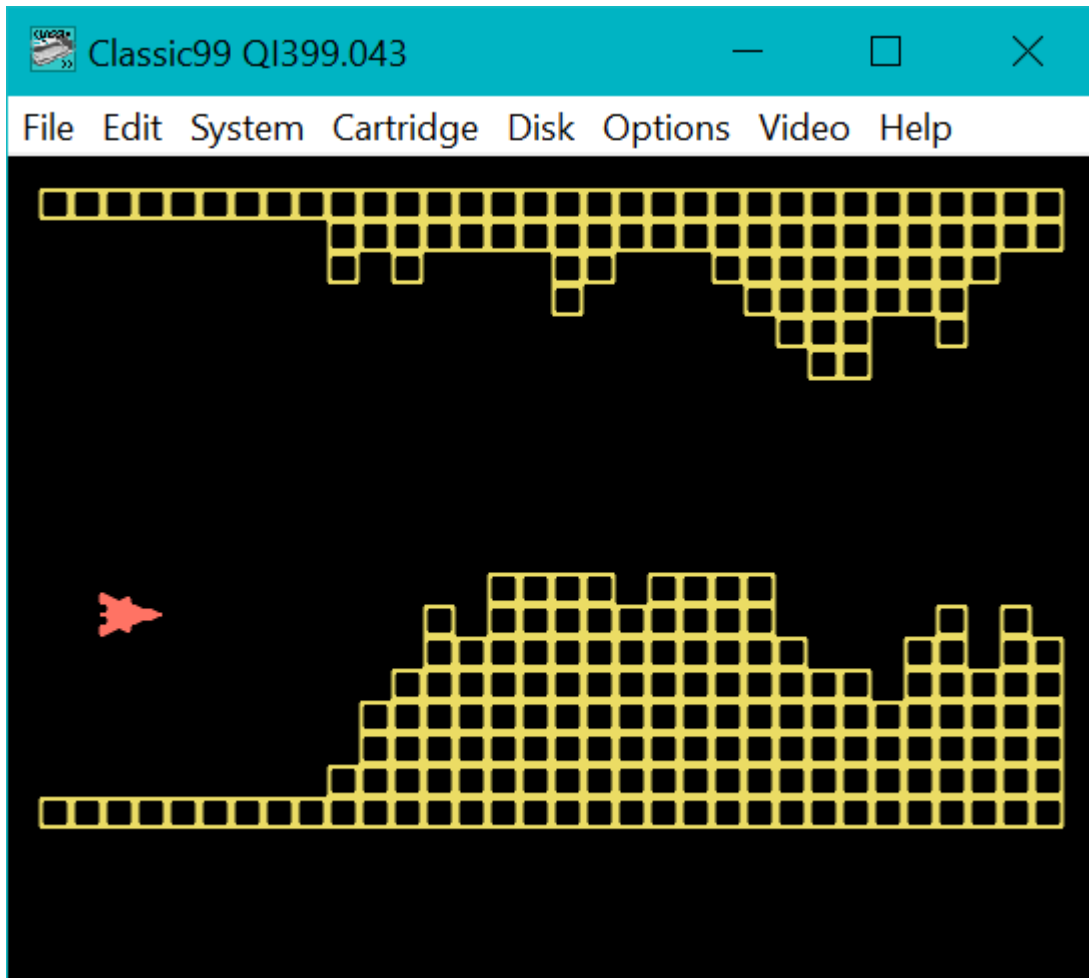
Let's have a look at the program in the emulator. We need to load the XB256 routines first so make sure that the compiler is still in DSK1.



Confirm the XB256 selection with Return. You will get Extended BASIC with a grey background as a reminder that XB256 has been loaded. Harry was so kind to pre-fill the prompt line with "OLD DSK" for you to complete:



Make it OLD DSK4.LSCROLL to load our program, then type RUN.



Better press "E" to pull up!

Press Enter to end the demo. You will notice that the program changes back to the unchanged primary screen where you entered RUN.

Finally, we will have a look at the inner workings of this package. It simply contains rules for transferring the simplified SXB name to the original XB CALL LINK statement in a very easy and simple format. Whenever a subroutine has parameters, they are universally referred to as 'P'. All parameters are passed as-is to the LINK call.

When a subroutine has optional parameters it is required to have two translation lines in the XBP Package file, first the line with parameters, followed by the line without parameters, i.e. SCRLUP in XB256.

```
CALL SCRLUP(P) -> CALL LINK("SCRLUP",P)
```

```
CALL SCRLUP -> CALL LINK("SCRLUP")
```

Each line starts with the SXB simplified code, followed by ' -> ' and the resulting LINK call. Be careful to avoid potential existing name conflicts, i.e. CALL LINK("SCREEN",P) must not be referenced as CALL SCREEN, as this is already taken by XB.

Here are the first lines of the XB256.xbpkg to demonstrate this:

```
// Translates simplified XB256 Subroutines into CALL LINK Statements

// SELECTING THE SCREEN

CALL SCR1 -> CALL LINK("SCR1")
CALL SCR2 -> CALL LINK("SCR2")
CALL SCREEN2(P) -> CALL LINK("SCREEN",P)


// COLOR AND CHARACTER PATTERNS IN SCREEN 2

CALL COLOR2(P) -> CALL LINK("COLOR2",P)
CALL CHAR2(P) -> CALL LINK("CHAR2",P)
CALL CHPAT2(P) -> CALL LINK("CHPAT2",P)
CALL CHSET2 -> CALL LINK("CHSET2")
CALL CHSETL -> CALL LINK("CHSETL")
CALL CHSETD -> CALL LINK("CHSETD")
(...)
```

For XB256 all routines have been named as in the original CALL LINK, except

- CALL LINK("SCREEN",P) is to be used as CALL SCREEN2(P) in SXB
- CALL LOAD(-1,N) may be used as CALL SYNC(N) to set the interval

With this knowledge and the existing XB256 documentation you will be able to use this library in your programs.

XB256 is also special because it can be used by the JEWEL compiler. The compiled program includes the XB256 routines and does not require to load it first.

Want to try it? Start the menu with RUN "DSK1.LOAD" and select the compiler. As you entered the filename at the OLD DSK prompt the compiler should already know what you want to compile, otherwise specify DSK4.LSCROLL-M manually.

```
*****
* EXTENDED BASIC COMPILER *
* Harry Wilhelm 2023 *
*****
```

```
XB merge file to compile?
DSK4.LSCROLL-M
```

```
Assembly file to create?
DSK4.LSCROLL.TXT
```

```
Assembling with Asm994a?    Y
Put runtime in low memory?  N
Proceed?                      Y
```

After the compiler has finished, fire up Asm994a.exe and add source file LSCROLL.TXT. The settings should be unchanged from the last usage. Start Assembly and check for Errors and Warnings in the Log.

Complete the Loader and try the compiled program after going back to normal CPU speed:

LOAD COMPILED PROGRAM

```
Using Assembly Support? N
Enter filename to be loaded:
DSK4.LSCROLL.OBJ
Runtime in high memory
16416 bytes remaining
File is Loaded
Press Enter (F4 will bypass)

>CALL LINK("EA5", "DSK4.LSCROLL-E")
>SAVE DSK4.LSCROLL-X
>RUN
```

Your spaceship is a little sluggish now, compared to horizontal speed, isn't it?

Here are some suggestions for you to experiment with:

- Increase the vertical speed from 8 to ...?
- Remove the safe corridor in the middle, but still make sure that there is always at least 4 characters of free way
- Check if you hit the wall.
- ...

Yepp. It's your turn now to play around.

Writing a fast action game for the TI-99/4A has never been easier!

How to start on your own

Start from Scratch

Always remember, SXB is just a small extension to Extended BASIC. Continue to think in the CALL's, DISPLAY AT, ACCEPT and PRINTs. Think new when it comes to program flow. Top-Down thinking like in *SteveB52* is a good start. Give your program a fundamental structure and then fill the subroutines with life. Remember to build some kind of loop when you want to come back. Try to dump the use of GOTO, at least to jump back. Use indentation to visualize the structure of your program. Index-variables and often used variables may be short. All others should receive intuitive readable names, explaining purposes such as "Score", "Hiscore", "Level". When you are short on memory, use the variable tab to assign memory-saving short names.

Continue an existing project

To continue an existing project you'll need a text file containing your code. All of my old projects from back in the 80s were stored on cassette tape. I used [TAPE994A](#) version 3.1 to "extract" them, as it worked better for me than [CS1er](#).

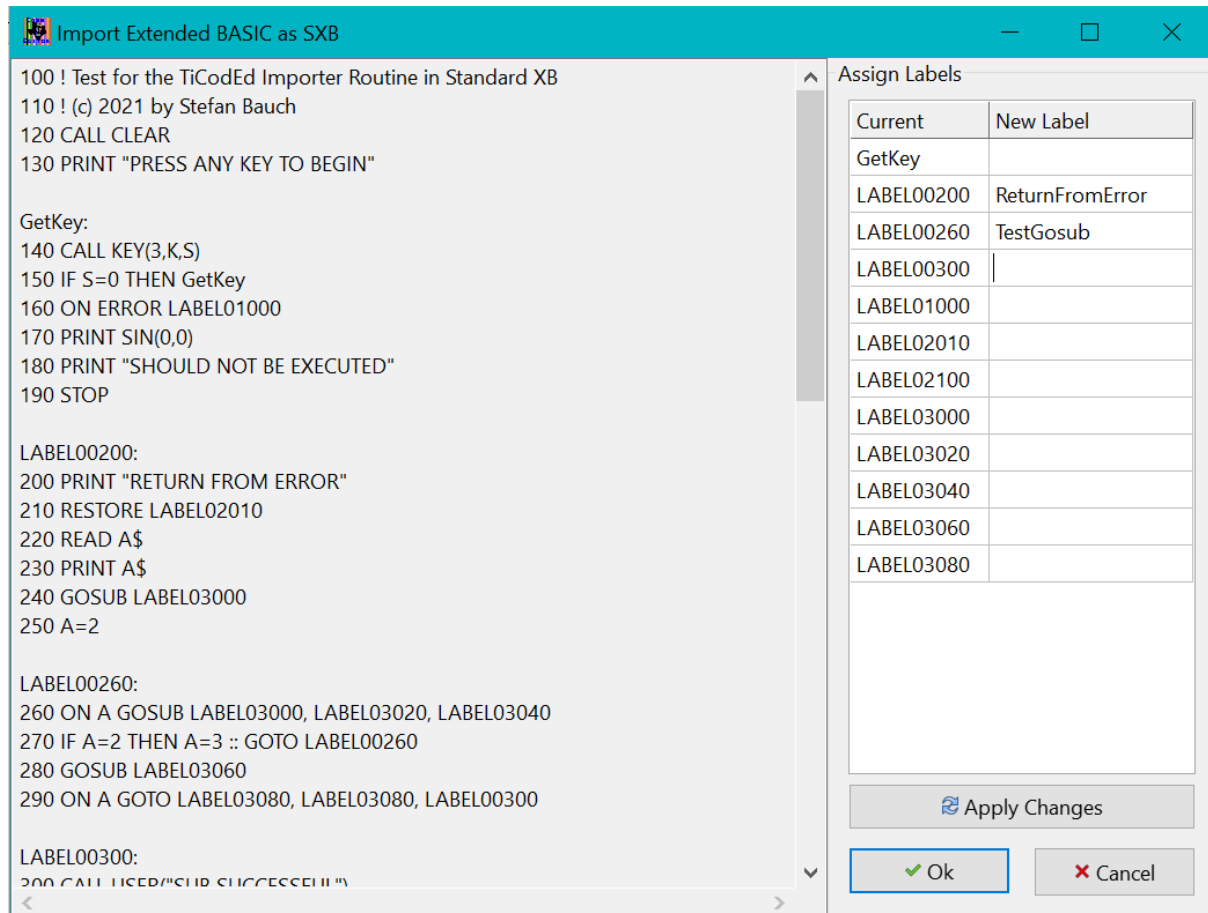
These two helpful Windows programs are able to read and record a digital program from any TI cassette tape. They then convert and save the program files first in WAV format and then again to a tokenized digital format for use in an emulator. You will need to connect a cassette player to your soundcard line-in port. Both of these wonderful tools come with manuals giving you the exact handling and format to be used.

I recorded a full side of a tape (30/45 min) in the required 8bit WAV format after trying different volume and tone settings with a single program first. Next you need to convert the tokenized file to an editable text file. [TI99Dir](#) can convert the file for you. Select your program in DI99Dir and press F3 or Files / View File. From there you can save the program as text.

It depends on the grade of completion how you should proceed. If it is merely a demo with some nice graphics you should consider starting new and copy the elements over.

If you have an advanced project you just want to touch up and finish you may want to use TiCodEd only as a convenient editor able to save tokenized files. If you decide, like I did, to at least drop the line numbers the following approach may work for you as well.

You may use the File/Import function to convert your XB program as a text file to SXB. The importer can't bring any structure to your code, but it can find all line numbers that are used in GOTO, GOSUB, RESTORE or other statements and assign a label to them. These labels are initially LABEL00100 for line 100, but can be renamed interactively in the Import-Assistant or manually later on.



When you click on a label in the right table the label will be selected in your code. You may keep the default name or assign a meaningful name in the "New Label" Column. By pressing the Tab key you can move to the next label and get it highlighted. When done, press "Apply Changes" and the labels will be updated in your code and move to the left and might be changed again. Press okay when done and close the Import-Assistant. The line numbers will be removed and your program is shown on the SXB tab with initial indentions applied.

```
1  ! Test for the TiCodEd Importer Routine in Standard XB
2  ! (c) 2021 by Stefan Bauch
3  CALL CLEAR
4  PRINT "PRESS ANY KEY TO BEGIN"
5
6  KeyPressed:
7      CALL KEY(3,K,S)
8      IF S=0 THEN KeyPressed
9      ON ERROR ErrorHandler
10     PRINT SIN(0,0)
11     PRINT "SHOULD NOT BE EXECUTED"
12     STOP
13
14  ErrorReturn:
15     PRINT "RETURN FROM ERROR"
16     RESTORE RightData
17     READ A$
18     PRINT A$
19     GOSUB GoSub1
20     A=2
```

If you prefer the manual conversion you may start with the first line and watch for any reference to a line-number. Check the destination and think of a descriptive name for this jump-destination. Enter it as a label (with a trailing colon) in an extra line above, but leave the line number intact for now, as there might be more references to this line in your code. Replace the line number in the reference with the label and proceed with the next reference just the same. Only when you have reached the last line, and your references no longer rely on line-numbers, should the line numbers be removed. Do so safely as the use of labels has removed the line number's last remaining purpose in your code.

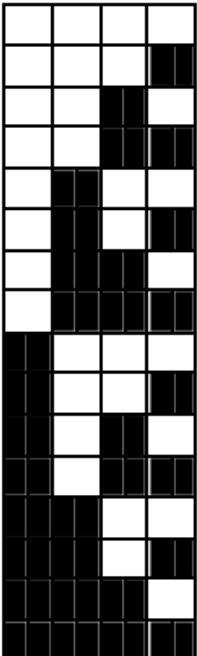
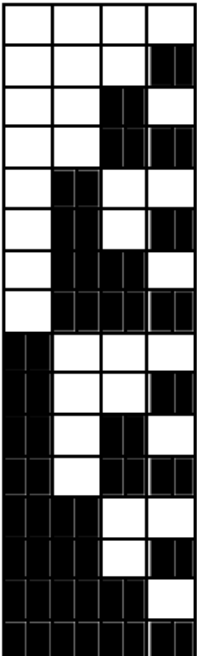
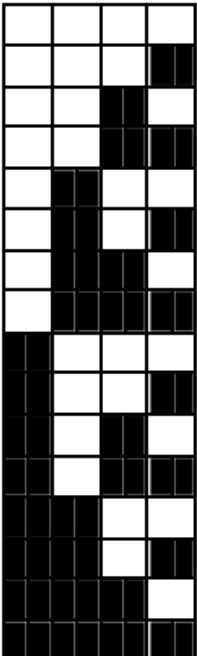
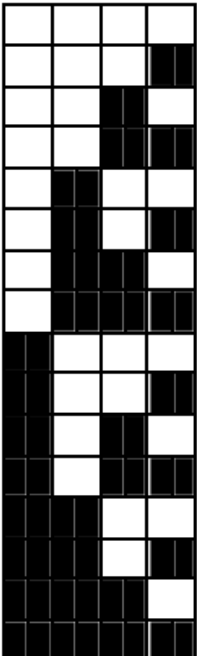
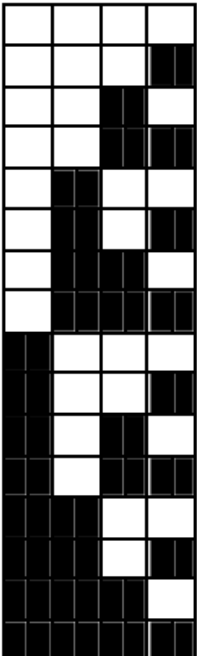
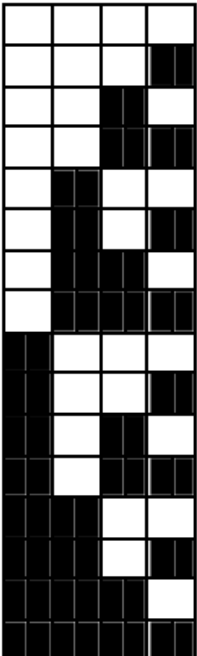
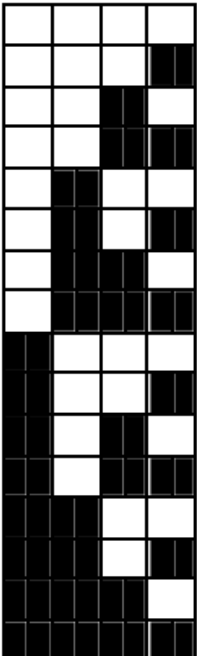
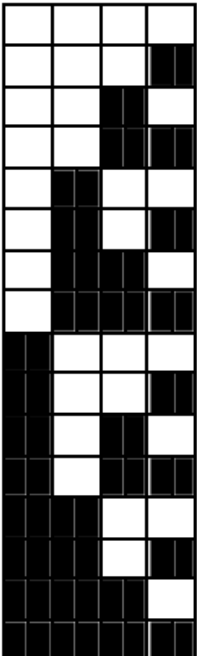
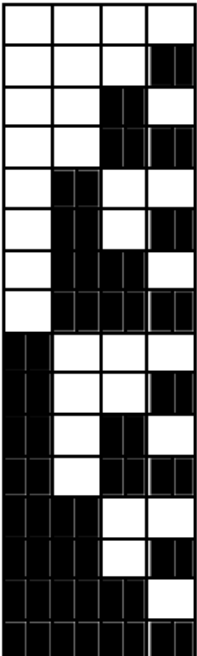
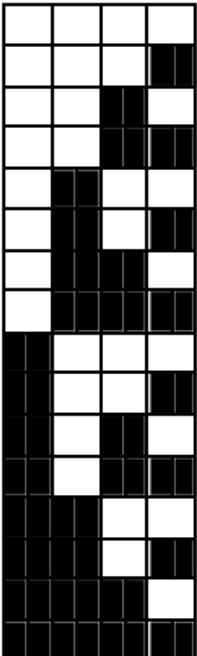
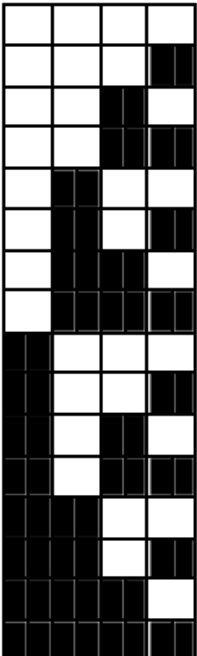
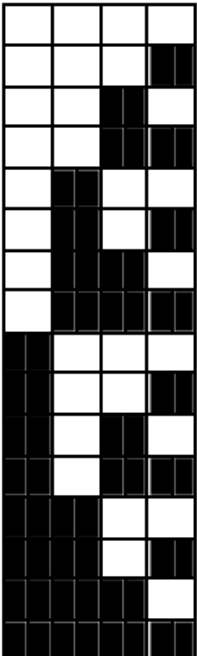
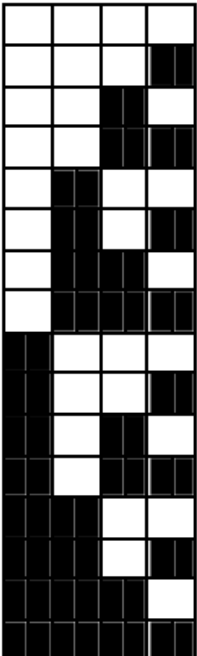
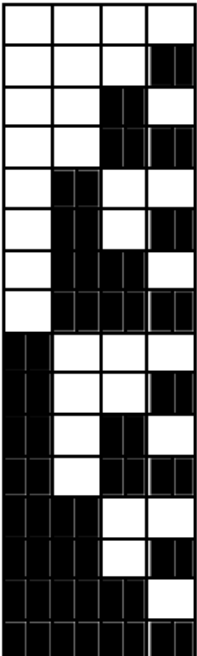
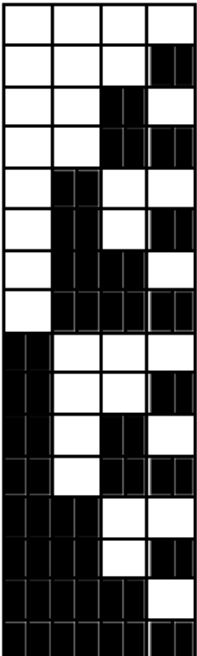
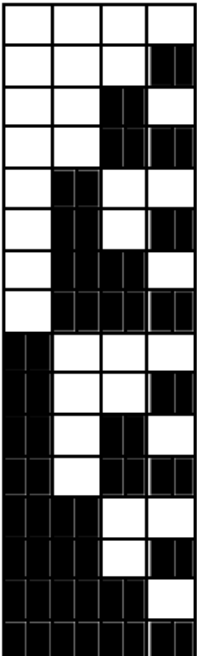
Next you can check for loops done by GOTO-Back's, and see if they can be replaced with REPEAT or WHILE loops. Once you have refactored your program, test it. If you plan to compile it, check for floating point operations. You may have used IF RND<0.1 THEN ... Change it to IF RND*10=1. Also check if variables like scores may exceed 32767. Adjust the program accordingly.

I suggest completing this conversion first before extending the program.

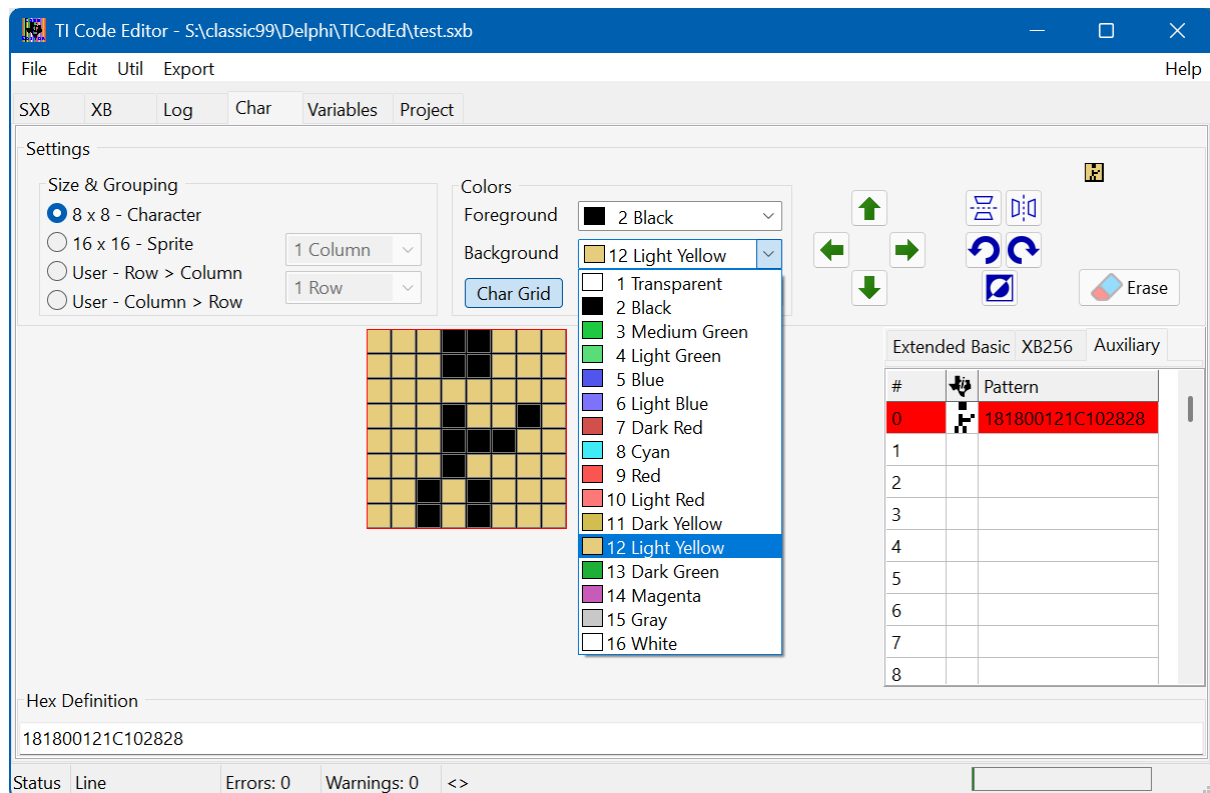
Advanced Topics

The Character Definition Tab

One of the most distinctive features of the TI-99/4a is the capability to redefine characters from BASIC without “poking” in the internals of the computer. With Extended BASIC you are also able to define sprites as we have seen in some of our examples. At this point we used hex-strings to define the shapes of characters without further explaining. Each hexadecimal digit represents four bits, or in a graphic way, four pixels.

BLOCKS	Binary Code 0=Off: 1=On	Hexadecimal Code
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
	0111	7
	1000	8
	1001	9
	1010	A
	1011	B
	1100	C
	1101	D
	1110	E
	1111	F

A character has 8x8 pixels, two digits per pixel-line, gives 16 hex-digits per character. You may use squared paper or a computer tool such as [Magellan](#) or the online tool [Raphael](#) to draw your graphics. The much easier to master tool named Char Tab is built right into TiCodEd! How great is that? You can toggle single squares with the left mouse button, or draw multiple squares with the right mouse button. The Hex Definition is updated automatically.



The color options are just included for a better evaluation of your images. Colors do not become part of the character (shape-only) definition. You may copy the Hex String out of the definition field to use in your program, but this also works the other way around. Take the following line from our last example:

```
CALL CHAR(124, "0000C0F3FF3F3FFF3F3FFFF3C0000000000000000C0FCFFFCC")
```

Click on the radio button for 16 x 16 pixel, select the hex string in quotes in the line above, copy it with Ctrl-C and paste it to the Hex Definition.

The Erase-Button resets the visible field to blank and the green arrows to shift the pattern one pixel. The blue buttons mirror horizontally/vertically, turn 90 degrees left or right and invert all pixels.

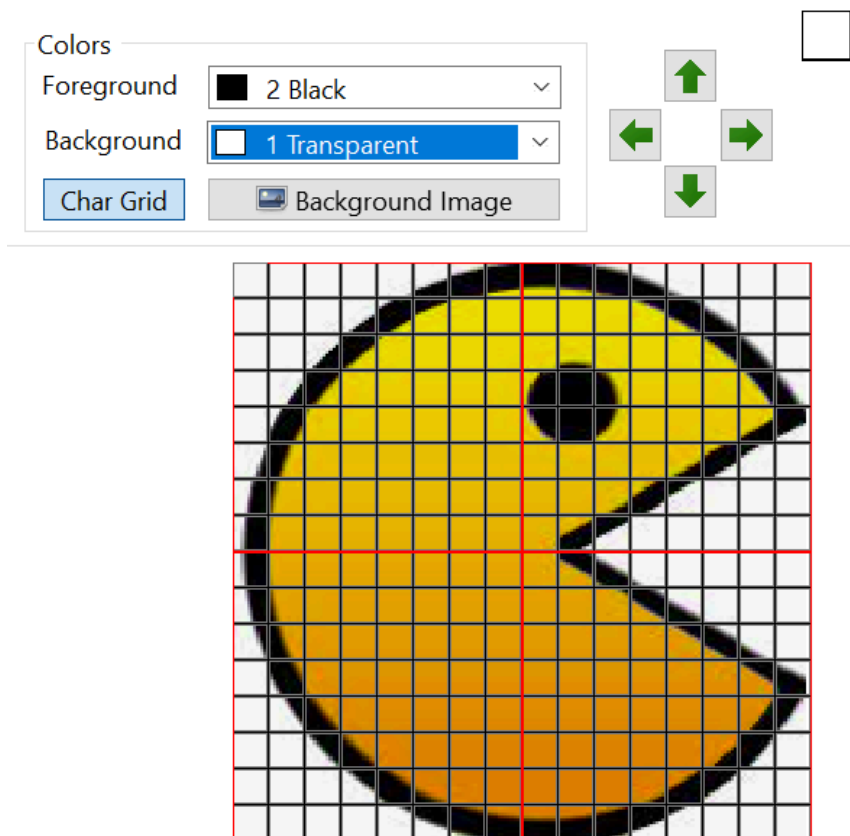
On the right side of the screen you see a charset table with three tabs:

#	Page Name	Characters	Reference
1	Extended Basic (or XB on Mac)	32 to 159	#CHAR1xxx
2	XB256	0 to 255	#CHAR2xxx
3	Auxiliary	0 to 255	#CHAR3xxx

The first column is the character number and the printable character when available, the second is the preview and the third is the Hex-Definition. You may edit or enter valid hex codes directly in this table.

The “**Size & Grouping**” setting allows you to group chars as needed. Groups can be selected by clicking on any char within the group. The first line of the group is marked in red, the remaining lines in yellow. Inactive groups are indicated with a bold character number and the first line of a group has a gray background. Regrouping is always possible and does not change any pattern.

By selecting “Background Image” you may load a graphic in the transparent background as a pattern to copy.



The charset editor is linked to your SXB program in two ways:

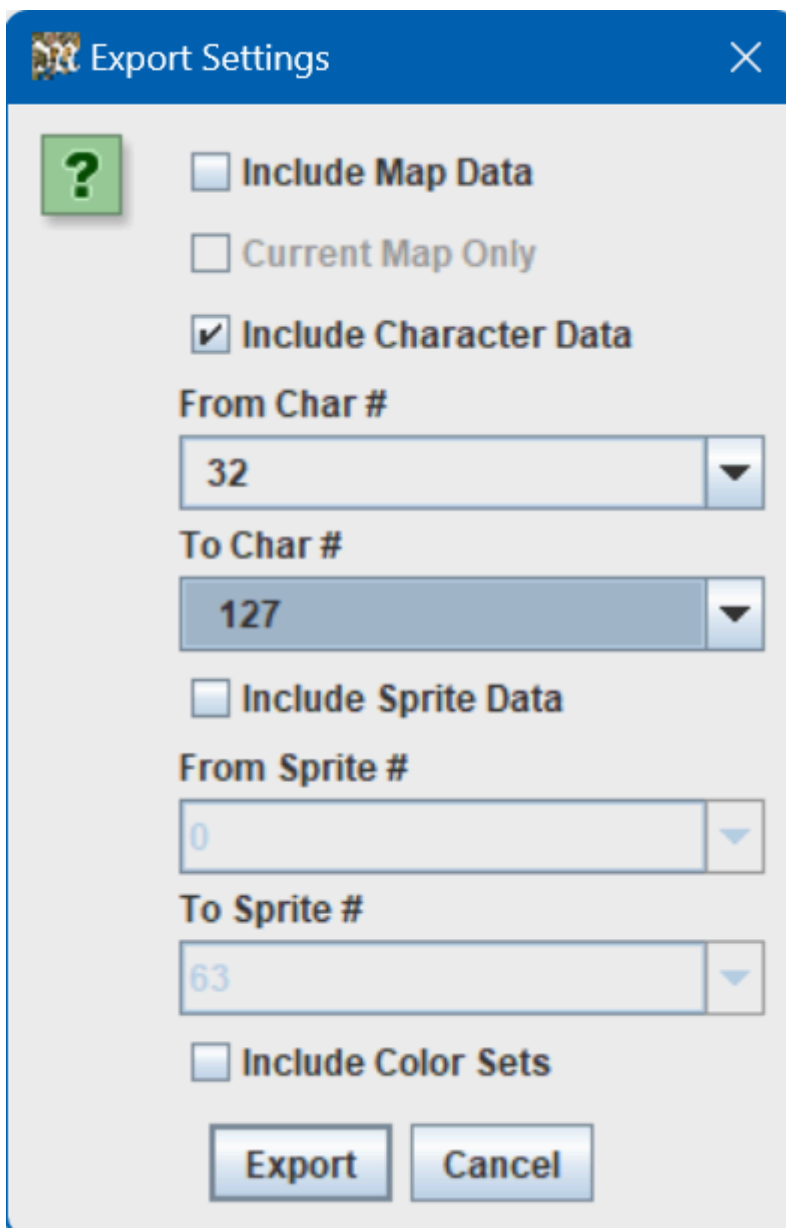
CALL AUTOCHAR – Use this subroutine in your program to dynamically create a subroutine defining all chars (re-)defined in the “Extended Basic” and “XB256” tab, the first by using CALL CHAR, the second by using CALL LINK(“CHAR2”,...) when building your project.

You can **refer to any char** in the three tabs with the literal #CHARnxxx or #CHARnxxx:z , for example as CALL CHAR(132,#CHAR3012:4) which will refer to chars 12 to 15 from the third tab “Auxiliary”.

With a click with the right mouse button you get a context-menu:

- Load Font
- Load Charset
- Dump Font
- Dump Charset

You may load and font files (starting at character 32 "Space", up to character 127) or charsets (starting at character 0, up to character 255, honoring the limitation of XB on the XB page). The files must contain 8 bytes per character. [Magellan](#) and TIFILES header are automatically ignored. Export from Magellan with Export/Binary Data with all other options deselected:

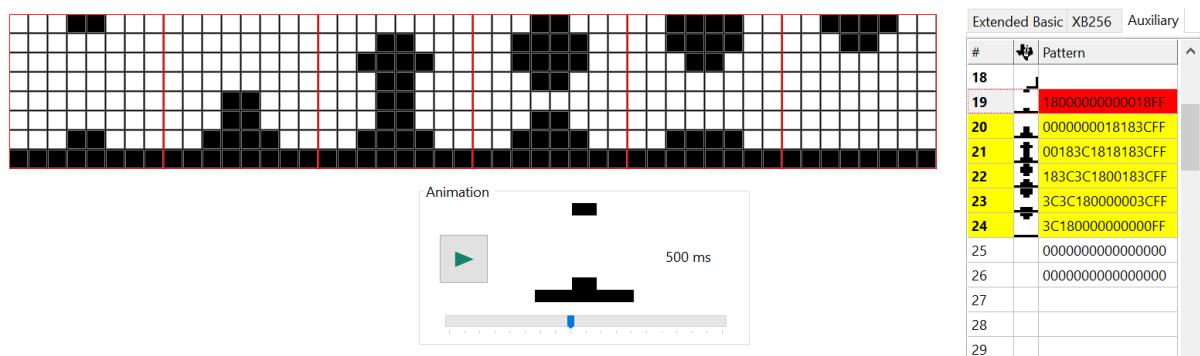


The image shows a dialog box titled "Export Settings" with a blue header bar containing a small icon and a close button. The dialog has a light gray background. On the left side, there is a green square button with a white question mark. To the right of this button are four checkboxes: "Include Map Data" (unchecked), "Current Map Only" (unchecked), "Include Character Data" (checked), and "Include Sprite Data" (unchecked). Below the "Include Character Data" checkbox are two dropdown menus: "From Char #" with the value "32" and "To Char #" with the value "127". Below the "Include Sprite Data" checkbox are two more dropdown menus: "From Sprite #" with the value "0" and "To Sprite #" with the value "63". At the bottom of the dialog, there are two buttons: "Export" and "Cancel".

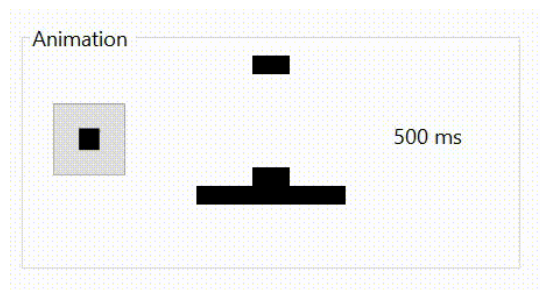
You may also dump (save) charsets and fonts for later use.

This should explain the use of the three charsets. The first is used to redefine your Extended Basic characters (also always used for sprites), the second the additional characters from XB256 and the last one for any dynamic assignment in your program, when characters are redefined for movements for example.

For sizes 1xN or 2x2N an additional section for **animations** appears:



You can play the animation while editing to see the result immediately.



CALL SUB or GOSUB?

Extended BASIC knows two ways to invoke a sub-routine, CALL and GOSUB. But what is the difference? Which is better? When should I use which one?

GOSUB is supported by nearly all BASIC Variants. You jump to a line and the RETURN statement brings you back to the next statement after the GOSUB. This way you can use this routine from different parts of your code and the computer knows where to continue the execution. With SXB you can use a label instead of a line-number, making it even more intuitive and portable. The appropriately named subroutine may be moved around in your program listing free from the shackles of sequential line numbering.

Extended BASIC also offers a very modern feature for its time, the ability to define your own CALL routines with the keyword SUB. The main difference to GOSUB is that you can pass parameters to the sub-routine, just like with traditional Extended BASIC, when you do a CALL HCHAR(12,1,88,32):

```
SUB MYADD(Param1, Param2, Param3)
    Param3 = Param1 + Param2
ENDSUB
```

The very special and modern concept is that the parameters are *local* to the routine. Within the routine you have no access to the variables outside the routine (global variables), which makes sure you do not change a value accidentally. The variables get their values from the CALL statement and return them if changed back to the calling program.

```
CALL MYADD(3,4,SUM)
```

The call is executed by assigning PARAM1=3, PARAM2=4 and PARAM3=SUM. When the line "Param3 = Param1 + Param2" is executed the Param3 is changed to 7 (=3+4). When ENDSUB is reached the variables are assigned back to the caller, so SUM will be set to the 7.

You may use variables with the same name as on the main program, without changing their value. These variables will remain active between subsequent calls. This isolation is a great feature, but a problem as well, as there is no way to declare any variable to be global, meaning that all needed values need to be passed to the routine as parameters, which can be difficult and time-consuming.

Please note that TI wants you to place all SUB commands at the end of a program. For more details please check the Extended BASIC documentation. TiCodEd will use the line-number you configure on the Project Tab for the first SUB statement, default is 30000.

Which form of subroutine calling is better? It depends. If you have a function which is independent of global variables and has a limited set of parameters, the CALL SUB is much more elegant. It is also perfect for the TiCodEd libraries, as they do not use any variables from your own programs. GOSUB is a good choice if you act on many global variables or just want to structure your program into smaller manageable blocks as the experts do.

Using BEGIN-END for code blocks

One annoying limit in the standard Extended BASIC is the maximal length of a codeline, especially in IF-THEN-ELSE. If you can't fit your statements in one line you need to use a subroutine or jump around a code-block. In SXB you may group statements in a "Block", starting with BEGIN, ending with END and spreading across multiple source lines. Please note, that END is already a statement in XB and without a leading BEGIN will end the program.

With Structured Extended BASIC there is a way around this limit, but it comes at a price. Take a look at the [The BEGIN/END-Block](#) of the SteveB52 example.

The BEGIN/END block gets replaced by a GOSUB and the code gets moved into its own section, ending with a RETURN.

Now you see the price to pay

- Runtime increases with the additional GOSUB / RETURN
- Your SXB and your XB program will differ in structure with this Out-of-Sequence-Translation

Caution: The created GOSUB needs a RETURN to clear the stack. If you want to abort a BEGIN/END block you may use RETURN. When jumping with GOTO outside the BEGIN/END block make sure to use RETURN to end the task, otherwise the stack will not be cleared and you may run out of memory.

The start line-number for the generated subroutines can be configured on the Project Tab and defaults to 20000. You may use BEGIN-END after THEN, ELSE and in the CASE-Statement introduced in the next chapter.

BEGIN-END Blocks may be nested, allowing complex logic, like in this following example.

```
input x
IF x=1 then begin
    y=1
    z=1
end else begin
    if y=1 then begin
        print "inner block"
    end
    y=2
    z=2
    print "outer block"
end
Print y
```

This out-of-sequence processing makes it a little more complicated finding the original line if XB gives you an error, but understanding the logic helps a lot. Look for the first line of the block, either 20000 or whatever your default is, or the line after the previous return. Looking for a GOSUB to this line leads you to the IF block.

Note the END in line 130 being automatically inserted to prevent the code at line 20000 executed after finishing the code-block (Print y).

```
100 input x
110 IF x=1 then GOSUB 20000 else GOSUB 20030
120 Print y
130 END
20000 y=1
20010 z=1
20020 RETURN
20030 if y=1 then GOSUB 20080
20040 y=2
20050 z=2
20060 print "outer block"
20070 RETURN
20080 print "inner block"
20090 RETURN
```

Note: When you use BEGIN/END within a SUB/SUBEND routine, the generated GOSUB will remain within this routine and not relocated to 20000. Instead, the SUBEND will be replaced by a SUBEXIT for the program flow, the generated code will be appended there, closing the last routine with a RETURN followed by a SUBEND.

The CASE Statement

When talking about complex logic with multiple IF-THEN lines, sometimes a CASE statement can simplify the program when you test one variable on different values, like the key pressed in a CALL KEY.

Take a look at the following example:


```

Print "Press H for Help"
repeat
  repeat
    call key(3,k,s)
  until s<>0
CASE k OF
  88 : Print "Down"
  69 : Print "Up"
  83 : Print "Left"
  68 : Print "Right"
  72 : BEGIN
      Print "Help:"
      Print "Press S,D,E or X"
      Print "Press Enter to quit"
    END
  13 : Print "Exiting..."
ELSE BEGIN
  Print "Wrong Key"
  Print "Try again"
END
ENDCASE
UNTIL K=13
END

```

The CASE statement line defines which variable is used. It might be numeric or string. Each following line defines one branch of the CASE. Multiple values are separated by comma. If one line is too short for the branch you may use BEGIN-END blocks (see previous chapter). The ELSE branch is optional and is activated when none of the other conditions apply.

The CASE statement is closed with the ENDCASE Statement. Failing to do so may crash the translation to standard XB.

The CASE Statement is translated to a complex ON GOSUB and multiple Routines:

```

100 Print "Press H for Help"
110 call key(3,k,s)
120 IF NOT (s<>0) THEN 110
130 ON 1-(k=88)-2*(k=69)-3*(k=83)-4*(k=68)-5*(k=72)-6*(k=13) GOSUB
      20090, 20000, 20010, 20020, 20030, 20040, 20080
140 IF NOT (K=13) THEN 110
150 END
160 END
20000 Print "Down":: RETURN
20010 Print "Up":: RETURN
20020 Print "Left":: RETURN
20030 Print "Right":: RETURN
20040 Print "Help:"
20050 Print "Press S,D,E or X"
20060 Print "Press Enter to quit"
20070 RETURN
20080 Print "Exiting...":: RETURN
20090 Print "Wrong Key"
20100 Print "Try again"
20110 RETURN

```

This code is based on the boolean expression values -1 for true and 0 for false, calculating the index for the GOSUB. The 1 is the ELSE part, if none of the other conditions are true. All expressions are multiplied by 2,3,4,... to calculate the index. It is obvious that no two expressions should be true, leading to a wrong index and possibly a runtime-error. If ELSE is omitted, the calculation starts with the first expression.

The branches may be a single line or a BEGIN/END block. Single lines are terminated by a RETURN, BEGIN/END blocks are similar to the usage in IF-THEN-ELSE.

If you want the ELSE branch to be without any action, use "ELSE RETURN".

IN SET Condition

Sometimes you don't want to do different things on different values like in the CASE statement, but the same thing on a set of values, you want to test if your variable is in a set of values. Here comes the IN[...] function handy:

```
IF V$ IN["A","E","I","O","U"] THEN PRINT "Vowel"
```

You may use string or numeric values, separated by comma, enclosed in square-brackets. This function can be used anywhere where a condition is used, like in IF, REPEAT or WHILE statements.

Above example will be translated to

```
100 IF V$="A" OR V$="E" OR V$="I" OR V$="O" OR V$="U" THEN PRINT "Vowel"
```

You may also use "V\$ NOT IN[...]" but be aware that there must be exactly one space between NOT and IN, as this is used as a fixed expression.

Binary data with BIN\$

When you need some unprintable characters in a string you can use the BIN\$ feature of TiCodEd. It is not a runtime function, but is executed on writing the tokenized file to disk. It converts the provided hex.string to the binary string.

```
A$ = BIN$( "2A4021" )
```

In this example all the characters are printable, so when loading this line on the TI it would read

```
100 A$="*@!"
```

As it is converted while tokenizing, the hex-string needs to be a constant in quotes, variables are not allowed.

Embedded Assembly ASM ... ASMEND

Just like SUB/SUBEND, you can now include assembly language subroutines directly in your BASIC program, at least for compiling. This will not work when running interpreted Extended BASIC. The idea of having a similar construct to SUB/SUBEND is to create a program to be interpreted in CPU Overdrive until it is mature enough to be compiled. If you still discover performance issues, you may then substitute critical Extended BASIC subroutine 1:1 with assembly language. This works best if you create those routines already to the specification.

The ASM blocks were made possible by an extension Harry introduced in [Jewel 7](#). Please check the version of your installation and update to Jewel 7 or above before proceeding.

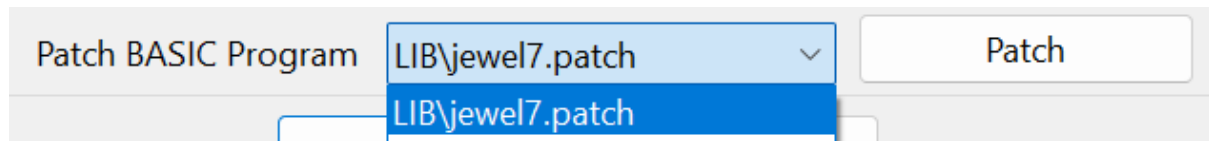
This version enabled adding user defined runtime files and extended the internal subroutine table with 32 additional slots for custom routines. These routines become part of the compiler. It is described in detail in the section "ADDING ASSEMBLY SUBROUTINES DIRECTLY TO THE COMPILER" of the compiler manual.

TiCodEd builds up on this functionality and partly automates it.

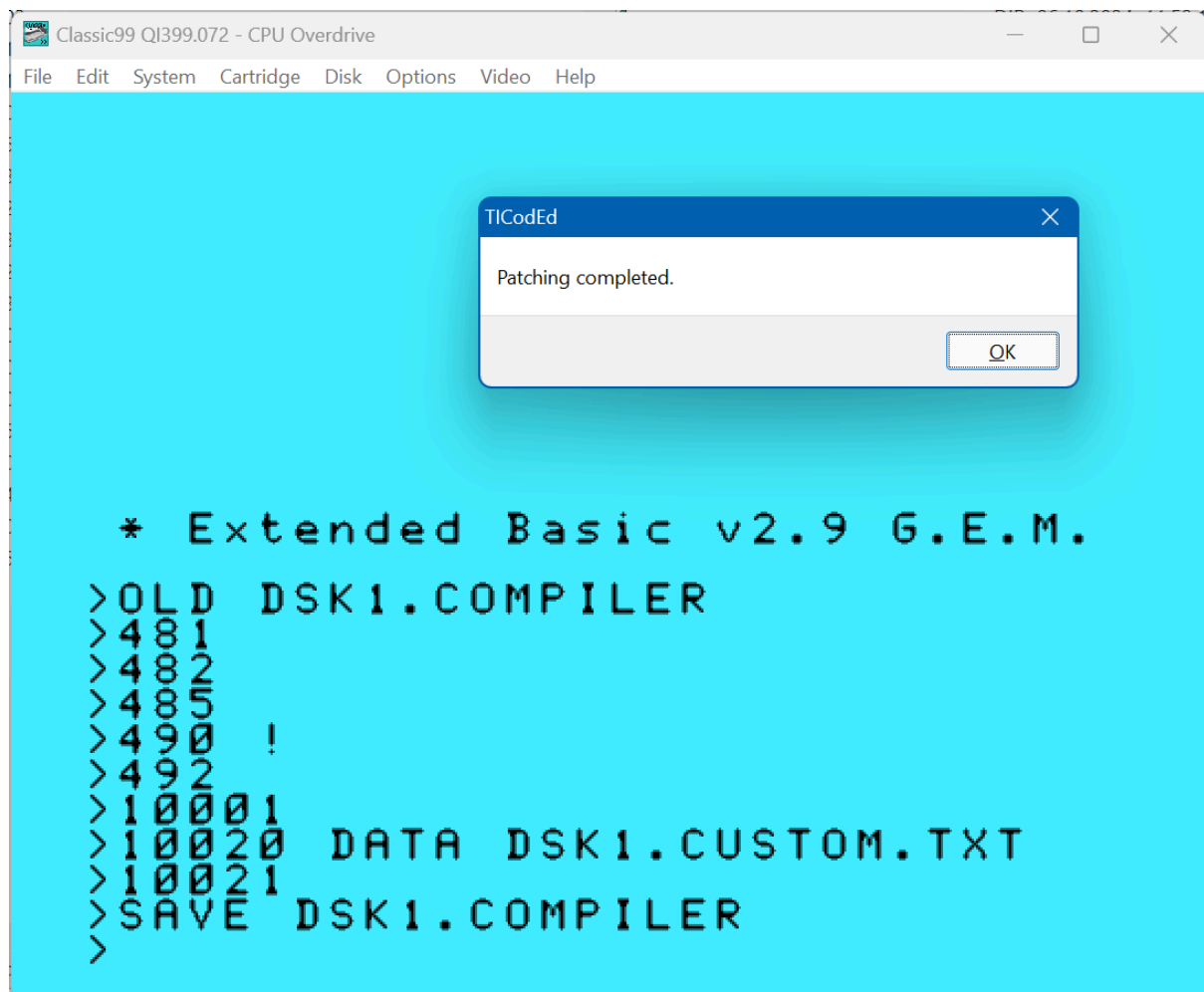
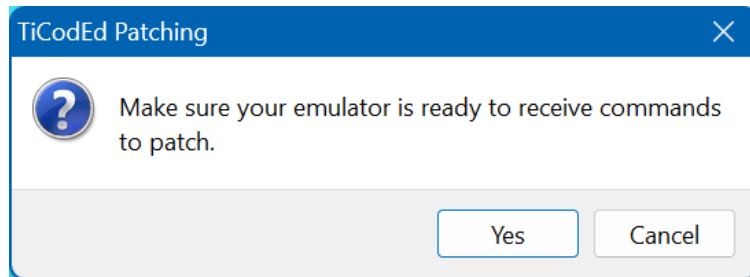
Prepare the Compiler

In order to use this feature, you have to patch the compiler. To apply the patch, go to the Preferences and select the patch you want to apply in "Patch BASIC Program". Please be sure to have the Compiler package available or a backup of

the compiler in case you need to start fresh or want to revert to the original behaviour, which is more interactive.



Select jewel7.patch if you have Jewel 7 installed, then press the "Patch" Button. Make sure that Classic99 is running, the Compiler in DSK1 and the XB prompt ready to accept commands:



This patch loads the compiler, deletes line 481, 482, 485, 492 and makes 490 a remark, to activate the functionality and remove the interactive part, which will be handled by TiCodEd now. Line 10000 and above keep the names of the

custom assembler subprograms and the name of the runtime file(s). TiCodEd will update the DATA line 10000 and put all runtime assembler into DSK1.CUSTOM.TXT, removing superfluous lines.

A program with embedded assembly language may look like this:

```

1 | A=12
2 | B=31
3 | CALL MPY(A,B,C)
4 | PRINT C
5 | END
6 |
7 | ASM MPY(A,B,C)
8 | * CALL LINK("MPY",A,B,C....) FOR the compiler
9 | * Multiplies the parameters A x B AND returns product TO C
10 | * The only purpose OF this is TO show how TO retrieve AND assign NUMERIC values
11 |   MOV *R4,R1      get A (equivalent TO NUMREF)
12 |   MOV *R5,R2      get B (equivalent TO NUMREF)
13 |   MPY R1,R2       multiply them
14 |   MOV R3,*R6      store the product IN C (equivalent TO NUMASG)
15 |   C *R13,R15      See IF NEXT word is a number OR an instruction
16 |   JLT MPY         do it again IF there are more numbers
17 |   B @RTN          AND GO back
18 | ASMEND
  
```

The line "ASM MPY(A,B,C)" will create an assembly line

MPY	BL @GET3	Get parameter: MPY(A,B,C)
-----	----------	---------------------------

On return by "B @RTN" the values are returned to the caller.

The syntax highlighting will switch to the assembly language when

- there is a blank in column 7
- there is a known assembly mnemonic starting in column 8

This is the reason why "B" is red in lines 2, 3 and 7, but bold black in line 17 ("Branch"). The following arguments are processed until the next blank, where the beginning of the assembler comment is expected.

Writing an ASM Routine

Writing assembly language is hard. Writing assembly language to become a custom routine within the compiler is even harder. Harry explains in the compiler manual the specific requirements for writing a custom compiler routine.

Here is a short recap, but make sure to read the compiler manual.

Parameter passing

While you can do everything described in the compiler manual, you get two automated functions. Up to four parameters will be store to R6 downward, where

- One parameter in R6
- Two parameter in R5 and R6
- Three parameter in R4, R5 and R6

- Four parameter in R3, R4, R5 and R6

Additional parameters may be read using the "BL @GETx" function as described in the compiler manual.

For numerical parameters, the register points to the number, for strings the pointer to a pointer to the string value.

This means, you just have the addresses of numeric values in your register. They are passed by reference. You can read and write to them using the * like *R6.

For a string parameter you get a pointer to a pointer, so you need to de-reference it first: MOV *R6,R6 lets R6 point to the length-byte of the string. INC R6 brings you to the address of the first character of the string.

You may manipulate the string directly. If you use a buffer, you may write the buffer back to your string variable using the @STRSTR routine.

Provided, you left the register pointing to your string untouched, you may automate the return of the string using ASMEND V\$(BUFFER). In this case, the BUFFER will be transferred automatically to the variable V\$, provided the corresponding register has been left untouched. See the example "FLIP.SXB" in the examples folder.

The compiler environment provides a general purpose buffer GPBUFF with 280 bytes (>0118) and lets you use registers R0 to R10 at your discretion. R11 is as always the return address and R13 to R15 are reserved for the compiler.

Debugging an ASM Routine

It can be tricky to create an assembly routine as part of the compiler runtime. If you experience problems, you may use the Classic99 debugger to see what is actually going on in your routine. But how to find the right code?

Classic99 offers "Additional debug opcodes", most important >0113 which is ignored by the real hardware, but sets a breakpoint in Classic99. To start debugging, just enter a pseudo mnemonic:

BREAK	Classic99 Breakpoint
-------	----------------------

This gets translated to:

DATA >0113	Classic99 Breakpoint
------------	----------------------

in your code. Make sure, the additional opcodes are enabled in Classic99.ini:

<pre>[debug] ScrambleRam=0 CorruptDSKRAM=0 enableDebugOpcodes=1</pre>
--

This way it is even easier to debug assembly code than Extended BASIC!

Check chapter 7.9 of the Classic99 manual for more information on debug opcodes in Classic99.

Assembly libraries

You may collect assembly language routines in a library file, store it in the LIB directory under with the extension .xbpkg. It will show up in the Project tab. TiCodEd uses smart linking, only used subroutines will be included in custom compiler subroutines. You may check which routines are in use in the Assembly tab after building the project.

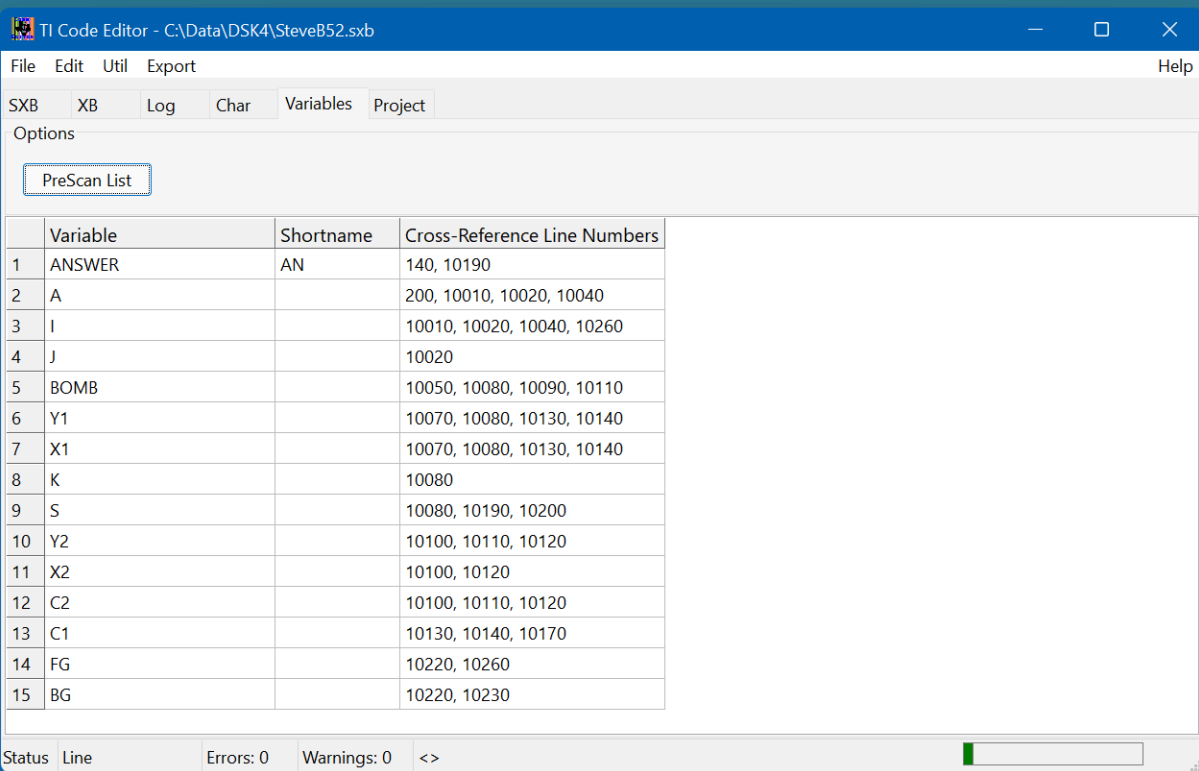
See the example [Classic99 Debug Library](#).

The Variable Tab

The Variable Tab serves two purposes. For one it gives you a cross-reference of the variables, listing all line-numbers in the XB tab, where a variable is used. If you find your program doing something wrong because a variable contains a wrong value, you may locate the error by checking these lines. You can also check if a variable name is already in use before introducing a new variable. You can sort the list by clicking the header column.

The second purpose is the definition of variable short-names. When memory is valuable we tend to use short names for variables as every byte counts. But this makes programs hard to read. Now you can have both, meaningful variable names and still save memory on the TI, as variable names get exchanged for their short version on exporting to the tokenized file.

Have a look at the SteveB52 Variables tab:



The screenshot shows the TI Code Editor window with the 'Variables' tab selected. The 'Options' section has 'PreScan List' checked. The main area displays a table of variables and their usage.

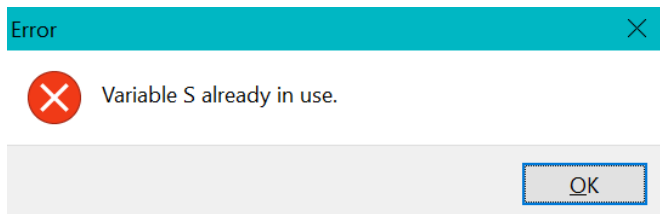
	Variable	Shortname	Cross-Reference Line Numbers
1	ANSWER	AN	140, 10190
2	A		200, 10010, 10020, 10040
3	I		10010, 10020, 10040, 10260
4	J		10020
5	BOMB		10050, 10080, 10090, 10110
6	Y1		10070, 10080, 10130, 10140
7	X1		10070, 10080, 10130, 10140
8	K		10080
9	S		10080, 10190, 10200
10	Y2		10100, 10110, 10120
11	X2		10100, 10120
12	C2		10100, 10110, 10120
13	C1		10130, 10140, 10170
14	FG		10220, 10260
15	BG		10220, 10230

The status bar at the bottom shows: Status Line Errors: 0 Warnings: 0 <>

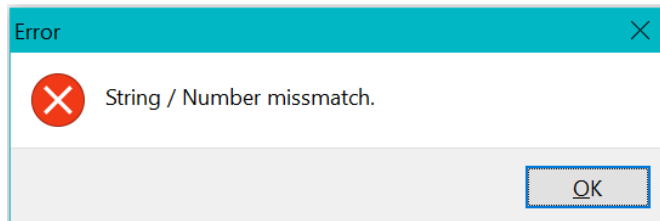
You see all used variables in the program and the lines they are used in one or more times. The variable "K" is only used in line 10080, but multiple times. The variable "ANSWER" will be exported as "AN" to the TI, all others will remain unchanged.

There are two checks performed when entering a short-name.

1. Is the variable already in use (long or short)?



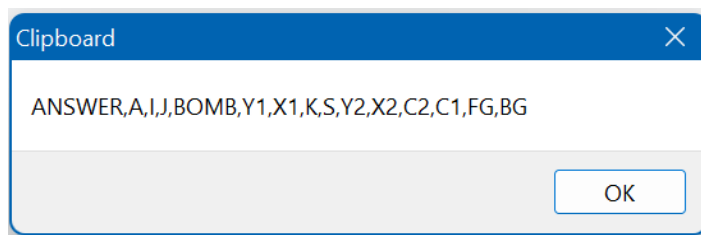
2. Is there a Number/String mismatch (both or none with \$ sign)



Make sure that you do not introduce a new variable in your program that has already been used as a short-name.

The list on this tab will be updated every time a build process is started. A newly introduced variable is available for the assignment of a short-name after the next build. The variable table is stored in the VXR-File specified on the Project tab and read when the Variable tab is displayed or you start a build process.

You may use the “PreScan List” Button to copy all used variables into the keyboard, separated by comma, to use the PreScan feature to speed-up the start of not compiled programs.

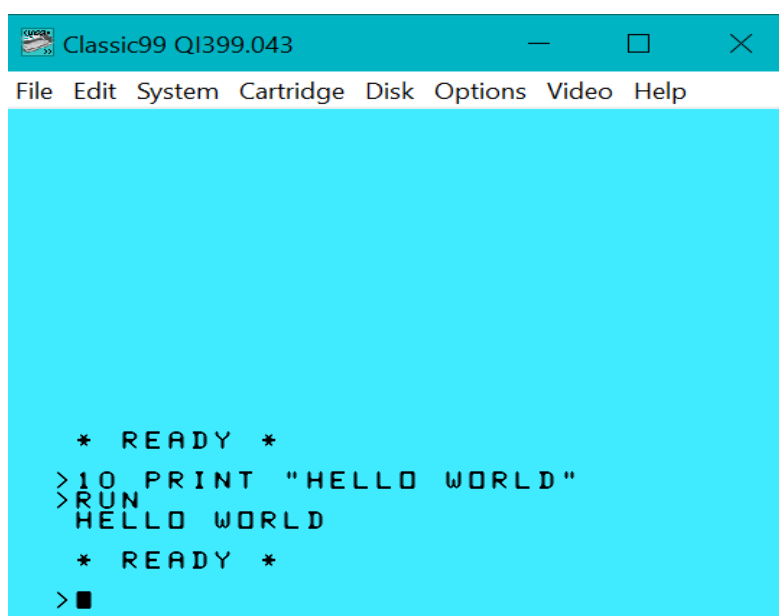


A daunting look at TIFILES

One of the most powerful and truly unique advantages of TiCodEd is the ability to effortlessly write program files in the internal, tokenized format, ready to be used by an emulator or the real TI-99/4A computer itself.

Not surprisingly, the TI file system works a little differently from today's PC systems. The TI had no internal clock, so file-date and time are unknown. But the TI stored file-type information of files, where Windows only recognizes flat “byte-stream” files. The TI-99/4A files system has fixed (DF = “Display/Fixed”), and variable record length files (DV = “Display/Variable”), or program files (“Internal”). This additional information is stored in specific areas of the file known as file headers. Two kinds of headers are popular in different emulators: TIFILES and V9T9 file header. Classic99 knows and handles both natively.

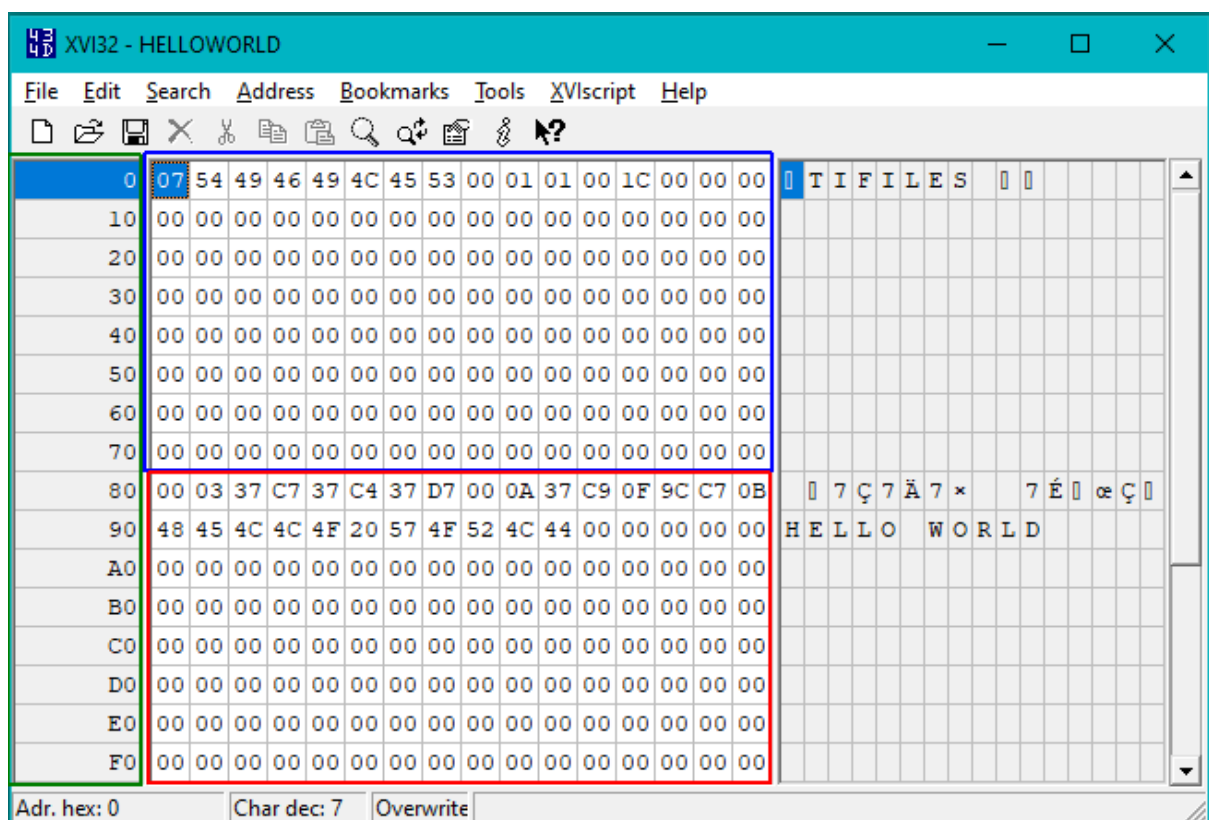
Do you remember the first “Hello World” program we did in the emulator and saved it to DSK4? It is time to have a closer look at the file.



When you use Notepad to look at this file you may notice that what the emulator saved to disk is not at all what you typed in:



Using a hex-editor like [XVI32](#) you will see some more details:



This editor shows the file in a split view of 16 bytes per line, left is the hexadecimal view of the data, right is the ASCII view. Hexadecimal means that each byte (value between 0 and 255) is shown as two characters from 0 to F, where A is 10, B is 11, C is 12, D is 13, E is 14 and F is 15. We talked about it when we looked at [graphic patterns](#) for CALL CHAR..

This is also the reason why the second "line-number" in the green square is "10": This is hexadecimal for 16. The reason why hexadecimal is so popular with computers is that 4 bits (0 or 1) can be written as one hexadecimal number, so two hex numbers are 8 bit equals one byte.

You can calculate the byte value by multiplying the first character by 16 and add the second, so >A2 is 10*16+2 = 162. We use the > sign to indicate a hex value to distinguish hex 80 as >80 (=128 dec) from decimal 80.

Back to our file...you see that the first 128 bytes (Lines >00 to >70 in the blue square) are started with "?TIFILES" as an identifier and then contain some internal values, followed by many >00. This part is due to the emulator and the settings to use the TIFILES header. At line >80 the actual program starts. You can see your text Hello World, but no quotes and no print statement. Why? There is just some gibberish before the Hello World.

The TI stores programs not as text, but as so-called tokens.

NOTE: Remember that *Tokens* are used in old computers to save precious memory. The practice of saving programs in a compressed *tokenized format* disappeared with affordable mass storage and cheap memory chips.

A TI BASIC/Extended BASIC token is a byte larger than >80 (128 decimal) which helps the (old-school slow as molasses) BASIC interpreter decide what to do. The first bytes are address information for loading the program.

In this example we start with line number >000A (=10 decimal, red), the memory address >C937 (reverse order, blue).

80	00	03	37	C7	37	C4	37	D7	00	0A	37	C9	0F	9C	C7	0B	7	Ç	7	Ä	7	*	7	É	æ	Ç
90	48	45	4C	4C	4F	20	57	4F	52	4C	44	00	00	00	00	00	H	E	L	L	O	W	O	R	L	D

the length of code line >0F (15 dec, green), the token for the PRINT statement >9C (pink),

80	00	03	37	C7	37	C4	37	D7	00	0A	37	C9	0F	9C	C7	0B	7	Ç	7	Ä	7	*	7	É	æ	Ç
90	48	45	4C	4C	4F	20	57	4F	52	4C	44	00	00	00	00	00	H	E	L	L	O	W	O	R	L	D

<C7 (blue) stands for a quoted string, >0B (11 dec, red) is the length of this string, >48 is H, >45 is E until all of "HELLO WORLD" is done (yellow, 11 characters), terminated by a >00 (green).

80	00	03	37	C7	37	C4	37	D7	00	0A	37	C9	0F	9C	C7	0B	7	Ç	7	Ä	7	*	7	É	æ	Ç
90	48	45	4C	4C	4F	20	57	4F	52	4C	44	00	00	00	00	00	H	E	L	L	O	W	O	R	L	D

If you are interested in the eight magic bytes before the line number, they are described under "Header" in a more comprehensive description of the file format on [Ninerpedia](#), there is also a list of all [tokens](#).

I have convinced you that you really do not want to bother with this, right?

Good news, I spent some evenings with that so that you don't have to.

Only keep in mind that a text-file with an Extended BASIC program on your PC can't be read by the TI or the emulator, it needs to be translated so that the TI can understand it, or as we call it - Tokenized.

Understanding the Log

The default for the log-level is 3 "Information". This is a reasonable setting if everything works as expected, as all information usually nicely fits on the screen. There 6 levels available:

1. Error - Only Errors are shown
2. Warning - Errors and Warnings are shown
3. Information - Errors, Warnings and Information are shown
4. Verbose - Also included debug information
5. Very Verbose - More Debug information
6. Unbelievable Verbose - Debug down to the bits

With level six the log file is about eight times as big as the SXB file. As there are several steps in building the project there are also multiple sections in the log file. The following examples are excerpts from the level 6 Log of SteveB52.

Analyzing the SXB file

In the first step the SXB file is read and line by line gets analyzed and classified.

```
Line-nr: 10000
Statement:
Label: PAINTSCREEN
Statement:
Statement: CALL ScrInit(16,2) :: DISPLAY AT(23,10):"SteveB52"
Found CALL SCRINIT
Found User Subroutine SCRINIT
Statement: FOR I=1 to 26 :: A(I)=0 :: NEXT I
```

Here you see that a line number of 10000 has been found, an empty line, a label called "PAINTSCREEN", a Call to "SCRINIT" which is not a standard XB subroutine, but somehow a user subroutine, either in the program or in a library. TiCodEd will look for it later in the process.

When the input has been read a list of found Non-XB subroutines is shown:

```
List of SUB-Routines:
  SCRINIT
Looking in Library for:
  SCRINIT
Read Standard-libfile S:\classic99\Delphi\TiCodEd\lib\StdLib.xbib
Found Sub-Routine SCRINIT
```

The SCRINIT routine could be found in the standard library, so everything is fine. After this the SCRINIT routine gets read and analyzed as well.

REPEAT-UNTIL, WHILE-WEND and line-numbers

In the next phase the new loop types get resolved and line-numbers assigned.

```
REPEAT: REPEAT001:
UNTIL [Answer=78 OR Answer=110]
IF NOT (Answer=78 OR Answer=110) THEN REPEAT001
REPEAT: REPEAT002:
UNTIL [(Y1>148 AND X1>200) OR C1<>32]
IF NOT ((Y1>148 AND X1>200) OR C1<>32) THEN REPEAT002
REPEAT: REPEAT003:
UNTIL [S<>0]
IF NOT (S<>0) THEN REPEAT003
Overwriting Line-Number 220 with 10000
```

The REPEAT Statement becomes a REPEATnnn Label first, as at the end of the loop you might want to return here UNTIL your condition is met. So the next thing is isolating this condition (shown in []) and constructing the IF statement by negating it with a NOT operator.

When a line-number is found in the SXB file it is checked that the current, automatic number is less or equal before overwriting the value.

Resolving Labels

Whenever a label is encountered it is noted with the assigned line-number. This table will first be dumped, then the labels will be replaced with the assigned line-numbers. A line may appear several times if it uses more than one label, i.e. in independent GOSUBs, in IF-THEN-ELSE or ON A GOSUB.

```
Line Number Table
  PAINTSCREEN -> 10000
  REPEAT001 -> 110
  REPEAT002 -> 10070
  REPEAT003 -> 10190
  GAMEINIT -> 170
  PLAYGAME -> 10070
  GAMEOVER -> 10160
  NOBOMB -> 10130
Label replace
  GOSUB GameInit
-> GOSUB 170
Label replace
  GOSUB PaintScreen
-> GOSUB 10000
Label replace
  GOSUB PlayGame
-> GOSUB 10070
```

Dumping the Extended BASIC file

Now the SXB has been converted to standard Extended BASIC.

```
Standard Extended BASIC:
```

```
100 GOSUB 170
110 GOSUB 10000
120 GOSUB 10070
130 GOSUB 10160
```

Prepare to tokenize

Next, the real magic happens... In Step #2 of the Build Phase the freshly created XB file is read into memory. The log shows you where the two token files should be written to and which XB file to tokenize.

It also reads the Variable X-REF file for assigned short-names.

Step 2: Save Standard Extended BASIC in TI internal format.

Save C:\Data\DSK4\SteveB52.xb and tokenize to ...

Token File: C:\Data\DSK4\SB52.

Merge File: C:\Data\DSK4\SB52-M.

Read file C:\Data\DSK4\SteveB52.xb into memory
40 lines read.

Read variable X-Ref file C:\Data\DSK4\SteveB52.vxr for shortnames.

ANSWER -> AN

1 variable shortnames read.

We see that one variable "ANSWER" has a short-name "AN" assigned.

Atomize into Tokens

Next the XB lines get "atomized", split into the very basic elements, the tokens.

Please note that the lines get both long and column-wrapped without a carriage return. The line-numbers are marked in bold for easier viewing.

```
130 GOSUB 10160 -> GOSUB|10160| -> 87[GOSUB] C9[Linenumbr] 27 B0[THEN] (??'?)  
140 IF NOT (Answer=78 OR Answer=110) THEN 110 ->  
IF|NOT|(|Answer|=|78|OR|Answer|=|110|)|THEN|110| -> 84[IF] BD[NOT] B7[(] 41 4E  
BE[=] C8[Unquotedstring] 02 37 38 BA[OR] 41 4E BE[=] C8[Unquotedstring] 03 31 31 30  
B6[)] B0[THEN] C9[Linenumbr] 00 6E (???AN???78?AN???110????n)  
150 CALL CLEAR -> CALL|CLEAR| -> 9D[CALL] C8[Unquotedstring] 05 43 4C 45 41 52  
(???CLEAR)  
160 END -> END| -> 8B[END] (?)
```

Each line consists of three parts, separated by " -> " arrows.

1. The original XB line
2. The identified elements on the line, separated by |
3. The token-converted line in hex with hints and raw in square brackets.

So the number three portion is the really important and interesting (nerdy) stuff. TiCodEd checks each element to see if it can be substituted by a token. This token has a direct impact on how the next element is interpreted.

I don't expect you will ever need this! You may skip the rest of this chapter. Read on if you want to learn about the exciting internals of a tokenized Extended BASIC file.

In the hex-output the token meaning is added in [] like in the first line 87[GOSUB]. This function should just spare you the chore of referencing the token table. The program does not know whether this hex value at this position really stands for that token, or if it is perhaps part of a line number. This is already the case on the first line. "27 B0" stands for the line to GOSUB, but B0 is also the token for THEN ($2*16^3 + 7*16^2 + 11*16 + 0 = 8192 + 1792 + 176 = 10160$). So be careful of this token-hint. In parenthesis you will see the actual bytes, all non-printable characters replaced by question marks.

So let us practice a little.

Line 130 consists of two elements: "GOSUB" and "10160". Line numbers are always marked with the token C9, then the line number follows in two bytes. 10160 is >27B0.

Line 140 is quite complex, but piece-by-piece you get it tokenized. There are tokens for IF (>84), NOT (>BD), Left-Parenthesis (>B7). The >41 4E stands for AN. Remember that we asked the Tokenizer to replace ANSWER by AN when doing the export? Here you see that at work. Funny that Texas Instruments decided to not wrap this variable name up like the number after the equal sign (>BE), which is a so-called "unquoted string" (>C8), followed by a length byte of two and then the >37 and >38 ("78") .

Continuing with the OR (>BA) and the comparison of AN= (>414E BE) with the three character unquoted string 110 (>C8 03 31 31 30). THEN is >B0 as seen before, then >C9 followed by the two bytes of a line-number >006E ($6*16+14 = 110$).

Line 150 is easy. CALL is >9D and the name of the subroutine is an unquoted string >C8 with the length of >05. CLEAR is >43 4C 45 41 52.

Line 160 is even easier. END is >8B.

You may wish to check the [full list of tokens](#) and the [BASIC file format](#) on Ninerpedia.

Program Statistics

Once tokenized the size of the program can be calculated.

Program Statistics in Bytes	
Code-Size:	1201
Line-Nr Table:	160 for 40 lines.
Length&Stop Byte:	80
Total Size:	1441

The actual code-size is the sum of all tokenized line sizes. These lines are prefixed with a length byte and a >00 stop byte, so 80 additional bytes for 20 lines. Additionally a line-number table is created, with two bytes for the line-number and two bytes for the memory address of the line.

Variable Cross-Reference

The cross reference is created and dumped here, identical to the VXR file and the Variables tab.

Variable Cross-Reference		
ANSWER	AN	140, 10190
A		200, 10010, 10020, 10040
I		10010, 10020, 10040, 10260
J		10020
BOMB		10050, 10080, 10090, 10110

Export Files

If the export works without a problem then the following three lines are outputted. If errors or warnings occur they are added after the filename.

```
Writing MERGE-File C:\Data\DSK4\SB52-M
Writing PROGRAM-File C:\Data\DSK4\SB52
Writing variable X-Ref file C:\Data\DSK4\SteveB52.vxr
```

Errors and Warnings

If there are errors or warnings a popup-window will indicate so.

```
Errors: 0
Warnings: 0
```

Advanced Project Settings

We covered most of the on the project tab down to the libraries, but skipped three:

Memory Base

This is a drop-down box with the default setting "auto", suitable for simple projects. Here are the options and when to use them:

Auto	For smaller programs the classic VDP RAM layout of the unexpanded console is used. Programs exceeding 11,775 bytes will use the IV254 format of the 32k RAM extension.
IV254	Use this setting to force the use of the 32k RAM extension, especially needed, when not all VDP RAM is available for the use

	of a Graphics Library etc.
Files(3)	This is the default for the VDP RAM, allowing buffers for 3 simultaneously opened files (See XB CALL FILES command)
Files(2)	Saves space by reducing to two buffers (eq. CALL FILES(2))
Files(1)	Saves space by reducing to one buffer (eq. CALL FILES(1))
Files(0)	Saves space by reducing to no buffer - This only works on some emulators, i.e. Classic99, with an external file buffering. Programs with this option will not run on a physical TI-99/4a

File Header

The most commonly used format for TI files on modern computers is the TIFILES header. The tokenized Program- and Merge-File are written with this 128 byte header recognized by most emulators and tools. You may switch this to the less popular V9T9 format.

TI BASIC

This checkmark indicates that the program is not Extended BASIC, but stock console TI BASIC. For now, the only difference is the slightly different handling of the REM statement (Add one space character). This may only be important when bit-identical versions must be created. Any XB program not using only the TI BASIC subset, can be loaded and executed under TI BASIC, even with this minor deviation.

Post Processing Command

If you have another environment than the one suggested here, you may want to trigger a program after you build your project. Perhaps your favorite emulator does not support FIAD and you need to put your program into a disk image. Or you want to commit your code to GIT or or or.

You may enter a command in the Post-Processing box on the Project tab.

Post-Processing

Command

☐ Active

You may activate and deactivate the post processing with the checkmark "Active".

You may use the following variables in your command which will be substituted before the command is handed to the operating system.

- %SXB% - Filename of the SXB file
- %XBT% - Filename of the Extended BASIC Text file
- %XB% - Filename of the Extended BASIC Token file
- %XBM% - Filename of the Extended BASIC Merge file
- %XBP% - Filename of the Project file
- %TIME% - Current time
- %DATE% - Current date

Include Assembler Data in HighMem

If you need reserved upper memory, i.e. you put the runtime in lower memory or have assembler code loaded to lower memory, you may create an **include file** with the same name as your SXB file, but the extension INC.

Here an example from the T.E.R.O game, where the build-in levels are stored in the upper memory through TERO.INC

```
BSS >0D82          reserve 3.5 kB space for levels
AORG >FF9E->0D82   point to the beginning of the reserved space
COPY "S:\classic99\DSK4\TEROLVL.TXT" and put your level-code there
```

The first line reserves a block of memory (>0D82 = 3458 dec), AORG back to the beginning of this block and then, in this example, include code via the copy command. This might be actual assembly code as well. The fixed base is >FF9E.

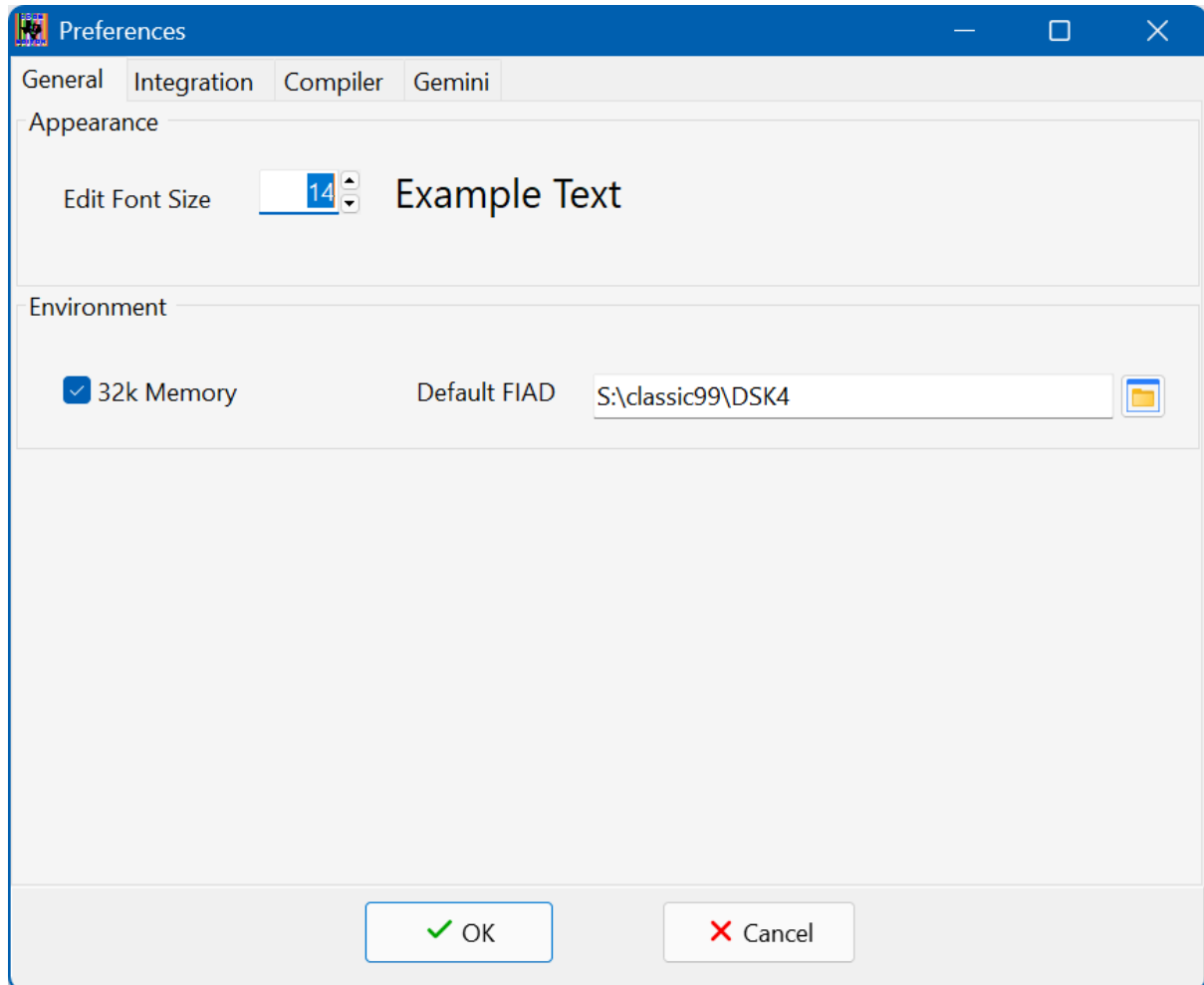
Take a look at the generated code:

```
4081          LASTDT
4082          EVEN
4083          ENDCC
4084 4A04          BSS >0D82          reserve 3.5 kB space for leve
4085 F21C          AORG >FF9E->0D82   point to the beginning of th
4086          COPY "S:\classic99\DSK4\TEROLVL.TXT" and put your lev
4087 F21C  30      TEXT '0020g58Jk2P5cF5Ci8Ea79,'
4088 F233  31      TEXT '1000jF5Ch4N5iWe9CbBPe9M,'
4089 F24B  2C      BYTE 44
```

Your INCLude-File will be injected after the ENDCC and the INCLude works only in the integrated scenario. See the thread "[Free HiMem after compiling XB?](#)" on AtariAge for detailed explanations and ADDING ASSEMBLY SUBROUTINES DIRECTLY TO THE COMPILER of the XB Compiler manual for using code in this memory.

TiCodEd Preferences

The default values for TiCodEd are set for the beginner, but needs to be adjusted to use the more advanced features. The “Integration” section is more complex and will be explained in the following chapter.

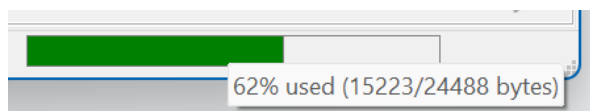


Appearance

The most obvious setting is the font size of the editor.

Environment

The 32k Memory extension flag is mainly used to scale the available memory in your status-line:

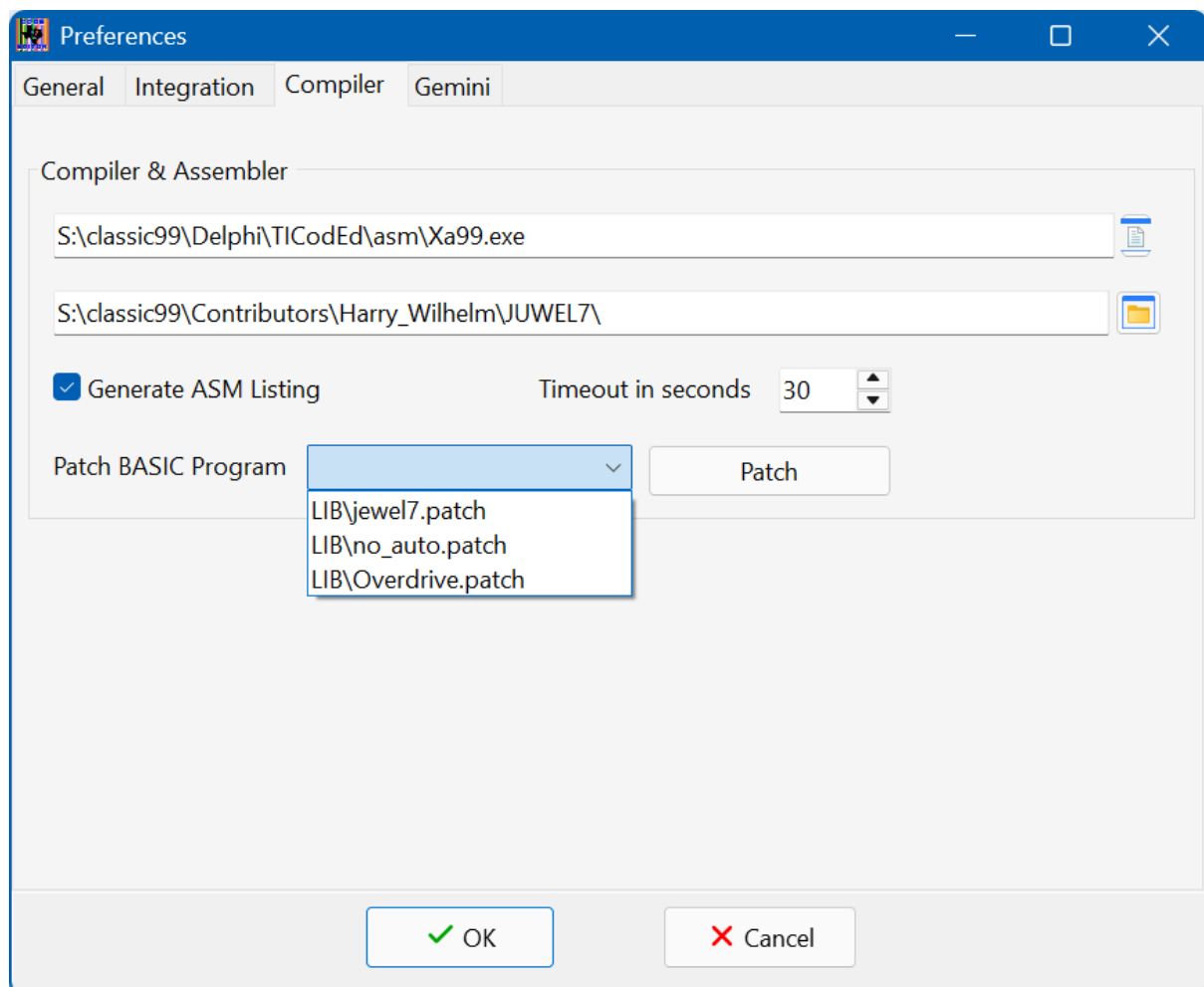


The **Default FIAD** Directory is used for your tokenized Program- and Merge-Files when first saving a file. This helps to separate your project PC files from the generated TI files.

Files	
Structured XB	S:\classic99\PRJ\ScuttleButt\ScuttleButt.sxb
Extended Basic Text	S:\classic99\PRJ\ScuttleButt\ScuttleButt.xb
Extended Basic Token	S:\classic99\DSK4\SB
Extended Basic Merge	S:\classic99\DSK4\SB-M
Variable X-Ref	S:\classic99\PRJ\ScuttleButt\ScuttleButt.vxr
Char Definitions	S:\classic99\PRJ\ScuttleButt\ScuttleButt.cset
TICodED Project	S:\classic99\PRJ\ScuttleButt\ScuttleButt.xbp

S:\classic99\DSK4 is mapped as DSK4 in my Classic99 configuration, whereas my project with all artwork, documentation etc. is in a separate project directory, sometimes a GIT repository.

Compiler & Assembler



In [Get the Compiler ready!](#) we used the compiler very much like Harry explained it in his great manual. Especially the use of the Asm994a.EXE assembler is a time-consuming manual step and the assembler has some known

bugs. There are other assemblers out there. Fred Kaal's Xa99 is a small cross-assembler as a command-line tool, fully compatible with the TI Editor/Assembler. While Windows-Tools with a graphical interface are usually more intuitive to use, command-line tools are easier to use in scripts and automation.

TiCodEd can automate the complete build-process with Classic99 and the BASIC Compiler including the assembly! It needs to know where the Xa99 assembler program is and where to find the compiler runtime. The first should be set automatically, as Xa99.EXE is included through the courtesy of Fred Kaal. TiCodEd, the second needs to be set by you depending on your Classic99 installation.

You may also point to the [XAS99.py](#) assembler by Ralph Benzinger, which is especially handy on Linux and MacOS.

Check the "Generate Assembler Listing" if you want to have a look at the output with all references generated by the assembler. The 30 Seconds time-out should work in most cases when the compiler is run in Classic99 "CPU Overdrive" mode.

Patching the Compiler

Starting version 3.0, TiCodEd is able to patch the compiler to your specific needs. This is done with simple macros by loading, modifying and saving the compiler in Classic99 or any other configured emulator on Windows. We did this already in chapter [Embedded Assembly ASM ... ASMEND](#), but there are more options available.

Start the Preferences dialog in the Edit menu and look for the "Patch BASIC Program" at the bottom:

`jewel7.patch`

This is the main patch to change the compiler to be able to use the TiCodEd ASM/ASMEND embedded assembler feature. It replaces the manual interactive maintenance of the custom routines with those based on READ/DATA, where the DATA line 10000 is maintained by TiCodEd. The custom runtime-file DSK1.CUSTOM.TXT is updated by TiCodEd as well

`no-auto.patch`

This patch changes line 180 as described in the Appendix B of the XBGDP manual to not use the autocomplete feature, which sometimes conflicts with the automation coming from TiCodEd.

overdrive.patch

This patch adds a line 1 to the compiler switching on the Classic99 CPU Overdrive mode via the Classic99 Debug OpCodes described in chapter 7.9 in the Classic99 manual.

Hint: If you want to have your compiled programs always switch to “normal speed mode”, add DATA >0110 in RUNTIME1.TXT as shown:

```
*****
RUNV  LI R0,1      RUNV will run program but NOT clear out screen2
      JMP RUN1
*****
RUN   SET0 R0
      DATA >0110   Switch Classic99 to normal speed
RUN1  CLR @XBEA5
      LI R1,CLRSCN
```

You need to enable the Debug OpCodes by setting **enableDebugOpCodes=1** in the Classic99.ini file. The OpCode is ignored by the TI and other emulators.

Integration and automation

TiCodED can be configured to send key-stroke messages to an emulator, especially to Classic99. This can be used to load, run or compile the program automatically in the emulator. In the Preferences in the menu Edit you find:

Preferences

General Integration Compiler Gemini

Target

☐ None

☒ Classic99

☐ Class Name <ID>

☐ Window Title <ID>

☐ Named Pipe <ID>

Target-ID /tmp/emul_in

Integration

Delay (ms) 20

Device DSK4

☐ XB to Clipboard

Filename Convention

☒ Use dots in filenames (old Classic99 behaviour)

☐ Toggle dots and slashes (TI-99/4a compatible)

OK Cancel

Select Classic99 and the Device you use in the emulator for loading the FIAD programs you create with TiCodEd. The Delay in milliseconds is the delay between keys sent to the emulator, the typing speed. 20 should work in most environments. Other programs might be identified by Windows Class Name or by exact window title. The Named Pipe can be used with [Emul99](#) under Linux instead of the keyboard emulation.

The Device is used to generate the full path to your files from the TI side. Specify the device you configured in your emulator.

By selecting "XB to Clipboard" the generated XB code is automatically copied to the clipboard and may be pasted to an editor or emulator.

If anything but "None" is selected, an additional section appears on the project tab:

Integration

☐ Auto-Load

OLD DSK4.SB52:RET.

☐ Auto-Run

RUN :S+:OEM7:S-DSK4.SB52:S+:OEM7:S-:RET.

☐ Auto-Compile

RUN :S+:OEM7:S-DSK1.COMPILER:S+:OEM7:S-:RET.
:WAIT4:F3.
DSK4.SB52-M:RET.
:RET.
Y:RET.
M:RET.

If the edit fields are empty they can be populated with a default by double-clicking in the edit box. Use this to get back to the default settings.

When you select one of the Auto-Checkboxes, this action gets executed after a successful build (no errors) automatically.

There is a fine, but important difference between sending characters or emulating keyboard entries. There is for example no dedicated key for the quote-character. For typing a quote you have to press the Shift-Key, press the key in the middle between Return and L (Labeled differently in each country setting), and release the Shift-Key. This can be constructed as:

`:S+:OEM7.:S-`

`:S+` is pressing Shift, `OEM7` is the internal Microsoft name of the mentioned key and `:S-` releases Shift.

When a build is successful, which means without errors, three buttons appear below the log. You can press Load, Run or Compile or hit the keys L,R or C to send the specified key-sequence to the emulator. The compile script will start the compiler, fill in the file-name of the merge file and set the standard settings. In the edit-boxes you see and may adjust the code sent to the emulator.

```
RUN :S+:OEM7.:S-DSK1.COMPILER:S+:OEM7.:S- :RET.
:WAIT3.:F3.
DSK4.SB52-M :RET.
:RET.
Y:RET.
N:RET.
Y:RET.
:WAITFILE. C:\Data\DSK4\SB52.txt
:ASSEMBLER. C:\Data\DSK4\SB52.txt
:RET.:WAIT1.
N:RET.:WAIT1.:RET.:WAIT2.
:RET.:WAIT1.:RET.
```

These keyboard messages are sent "blind", without any feedback. So TiCodEd does not know if the emulator is ready in the beginning or the compiler has finished. Make sure the emulator is ready to receive the commands and has enough time to execute the commands by adjusting the **:WAITx.** commands.

We replaced Asm994a with Xa99, but still check Y for the Asm994a question to not use the slow local TI assembler. For larger projects you may want to put the Runtime in low memory and adjust the first **N:RET..** to be a **Y:RET.** .

This script relies on the fact that the compiler generates the assembler file `C:\Data\DSK4\SB52.txt` , but the emulator only writes it to disk once finished (file CLOSE command). The `:WAITFILE.` will wait for the file to have more than 0 byte, then the Xa99 assembler is started locally on the Windows environment. The output can be redirected to the Log Tab by selecting a higher Log-Level in

the Project Tab. After successful assembly, the script starts the linker/loader and holds short of the RUN statement suggested by the loader.

You may edit the integration commands to your needs. For "Compile" the default expects the compiler in DSK1, but this can be changed. The command sequence can have up to 10 lines. You may restore the default by deleting all characters (including spaces!) and then double-click the edit field.

Please visit the Microsoft page for [virtual key codes](#). Many of these keys can be used in the TiCodEd Keyboard Emulation.

- Standard Characters A-Z and 0-9
- :RET. Return Key
- :TAB. Tabulator
- :ESC. Escape
- :F1. to :F24. Function Keys (i.e. :F3. for "erase")
- + - , .
- :S+ :S- Shift on / Shift off
- :C+ :C- Control on / Control off
- :A+ :A- Alt on / Alt off
- :OEM1. to :OEM8., :OEM102. See Microsoft virtual key codes for details

Special commands :

:WAITx. waits for x seconds before continuing (1-9).

:WAITFILE. <Filename> waits to the file to have more than zero bytes

:DIALOG. <Message> Displays a message-dialog with "Yes" and "Cancel"

:ASSEMBLER. <Source> Assembles the given source

You may use up to 20 lines. Note that for each line in the script:

- The focus is set to the specified program (Classic99)
- Before the first character 20 times the delay-rate will be waited

Increase the Delay in the Preferences if characters are missing.

You may use a "bigger" sign in the first column to send a string through the Clipboard as written under Windows without the need for key-sequences. Please note, that this will override your Windows Clipboard.

```
>RUN "DSK1.DEMO"
```

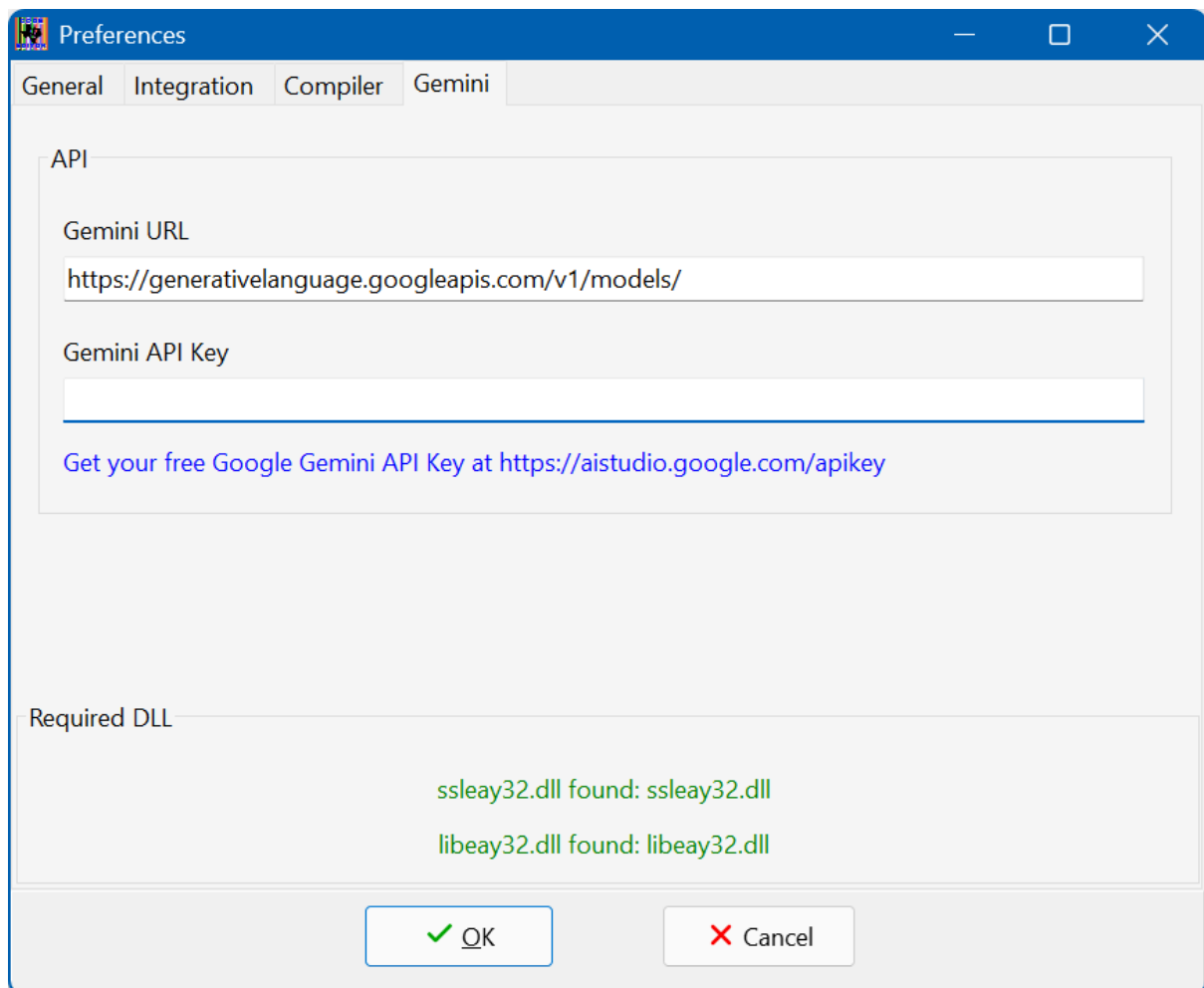
When not using Classic99 you may configure your emulator by using the Windows Class Name or the Window Title. Class Names for emulators are

- Classic99: TIWndClass

- Ti994w : Ti994w
- Win994a: n/a (use Window Title 'Win994a Simulator - v3.010 (x64)')

Gemini

TiCodEd 3.5 introduces an online-interface to the Google Gemini AI for coding. The Gemini URL default should work for you, but may be changed. You need to have an API key to use this feature, which requires a Google account. Click on the blue line or copy the URL to your browser.



The required DLL are checked under windows, under Linux libssl.so is checked to be installed by the libssl-dev package.

Libraries

Library reading sequence

In case you wondered what would happen if there are multiple routines with the same name in different libraries, you are really into the advanced stuff. While reading the SXB program TiCodEd builds a list of CALL routines not internal to Extended BASIC and not declared in a package as internal.

1. If the SXB program itself contains a matching SUB, this one is taken.
2. Secondly, the User-Library is searched
3. Then the selected Library Package-Files are read. Yes, packages may contain specific additional SUB Routines. See XB256 for an example.
4. At last the Standard-Library is searched.

As soon as a matching SUB routine is found it is added to the program and deleted from the list. This prevents double insertions and ensures the above priorities.

A warning is issued if the list of SUBroutines is not empty in the end, that means unresolved references remain.

Extended BASIC 2.9 Graphics Enhancement Module

The standard Extended BASIC is commonly version 1.1 from 1982 (PHM3026). There have been some improved versions since then. For later using the compiler, XB 2.9 GEM offers some advantages and is included in the compiler distribution. The loader for compiled object-code has been replaced with the much faster version from Mini Memory. It also introduces some statements not included in the original Extended BASIC, but available in the compiler:

- CALL MOVE(mode, start address, target address, # of bytes) moves a block of memory. The four modes available are: 1 - from VDP RAM to VDP RAM, 2 - from VDP RAM to CPU RAM, 3 - from CPU RAM to VDP RAM, 4 - from CPU RAM to CPU RAM
- CALL PEEKV(address, numeric variable list) reads values from VDP RAM
- CALL POKEV(address, numeric variable list) writes values to VDP RAM
- CALL LDCR(# bits,cruaddress,numeric value) Loads 1 to 16 bits contained in the numeric value into the CRU starting at cruaddress.
- CALL STCR(# bits,cruaddress,numeric-variable) Reads 1 to 16 bits from the CRU into a variable.

CALL MOVE is most important in the combination with the embedded utility "[Cart Patching](#)" to load screens, fonts or sound-lists.

The GDP module not only contains the Extended BASIC V2.9, but also the most common libraries.



TEXAS INSTRUMENTS HOME COMPUTER

PRESS

- 1 FOR TI BASIC
- 2 FOR EXTENDED BASIC V2.9
- 3 FOR XB256
- 4 FOR T40XB
- 5 FOR T80XB
- 6 FOR THE MISSING LINK
- 7 FOR TML GRAPHIC ADVENTURE
- 8 FOR MULTICOLOR XB

XB 2.9 uses the TI BASIC and not the Extended BASIC 1.1 pseudo number generator. The compiler does not support the setting of a seed like XB does, but this can be resembled by CALL LOAD(-31808,N1,N2) to get identical random numbers in each run. This can be used for testing and to generate reproducible level data in games.

XB256 Library

XB256 greatly enhances Extended BASIC for game development, like shown in the demo in chapter [Packages](#). It provides a secondary screen with 256 characters to be freely designed in 32 color groups. Sprites still get their character definition from the primary screen. HCHAR, VCHAR, GCHAR, PRINT, INPUTm DISPLAY and ACCEPT are working on the active screen without changes.

There are x groups of subroutines available with XB256

- Screen 2 characters and colors
- Character and pixel scrolling
- Soundlists, VDPRAM and sprite routines

Check the included documentation for details. When sing the XB256.xbpkg Package, you may use the subroutines directly, without the CALL LINK, i.e.

- CALL SCRN2
- CALL CHAR2(234,"0000FF55AAFF0000")
- CALL COLOR2(27,13,1)
- CALL SCRLRT(3)

- ...

There are two exceptions to this 1:1 translation:

- CALL SCREEN2(c) instead of CALL LINK("SCREEN",2) as CALL SCREEN is already in use for screen 1.
- CALL SYNC(cyc) instead of LOAD(-1,cyc) as a more intuitive name to set the numbers of v-sync cycles to wait (60 for NTSC, 50 for PAL per second)

RXB - Rich Extended BASIC

This enhanced BASIC is especially useful when not planning to compile, as it replaced many GPL routines with faster native assembler and added a lot of additional functions. RXB can access SAMS memory from BASIC.

There is a package file to declare all additional subroutines to be used with TiCodEd, so there are no unresolved references reported.

Extended BASIC 3

[XB 3](#) by Asgard Software is an enhancement to XB 1.1 by Winfried Winkler. It introduced multiple new functions, which are additional tokens not supported by TiCodEd: DATE\$, HEX\$, LALPHA, LWRC\$, OFF, TIME\$ and UPRC\$.

Without the "OFF" token CALL SCREEN OFF and CALL QUIT OFF are not correctly tokenized.

The additional subroutines are available through the XB 3.xbpkg package though:

BYE ALL ALLSET ALOCK BEEP CLRS FIND GOSUB GOTO GPEEK GPOKE HONK
KEYS MLOAD MOVE MSAVE PRNTPAT QUIT RESTORE RNDVPEEK VPOKE WAIT

Forty Columns Text-Mode T40XB

The T40XB package by Harry Wilhelm is part of the XB v2.9 G.E.M. package and offers access to the 40 columns text mode available in the TI-99/4a. This mode, other than the standard "G32 Graphics Mode". The video processor is only capable of 256 pixels in a row, so each character is only 6 pixels wide.

The standard graphics routines are not available, so this library offers alternatives for PRINT, INPUT, HCHAR, CHAR, COLOR and others.

Eighty Columns Text-Mode T80XB

The T80XB package by Harry Wilhelm is part of the XB v2.9 G.E.M. package and offers access to the 80 columns text mode when a 80 character card is available.

Check the T80XB.xbpkg file for the translation of the CALL LINK to direct calls.

The Missing Link

The Missing Link 2.0 offers 32 assembly routines to use high resolution graphics with 192x240 pixels in 2 or 16 colors. The 16 color mode is limited to an 1x4 pixel area can only hold one foreground and one background color.

Check the TML.xbpkg file for the translation of the CALL LINK to direct calls.

Multicolor

The TI-99/4a has a rarely used graphics mode with 64 x 48 pixels in 16 colors. A pixel has the quarter size of a regular character. It supports direct drawing and buffered drawing and "shapes", predefined graphics to be painted.

Check the MultiColor.xbpkg file for the translation of the CALL LINK to direct calls.

Classic99 Debug Library

Classic99 contains some additional CPU opcodes for debugging, which are ignored by the real hardware. The library Classic99Dbg.xbpkg makes them available in compiled Extended Basic. **enableDebugOpcodes=1** must be set in Classic99.ini to use these features.

CALL C99NORM	Switches Classic99 to normal speed
CALL C99OVRD	Switches Classic99 to CPU Overdrive
CALL C99SMAX	Switches Classic99 to System Maximum
CALL C99BREAK	Starts the debugger and halts execution
CALL C99QUIT	Shuts down Classic99
CALL C99DBG(T\$)	Outputs the string T\$ to the debug log, up to 128 bytes

You may use [DebugView](#) instead of the debug-window. Add a CHR\$(0) to your debug string as it expects a NUL terminated string. CALL C99DBG gives you the option to output debugging information to another window, either the Debug Screen or DebugView.

See chapter 7.9 of the Classic99 manual for details.

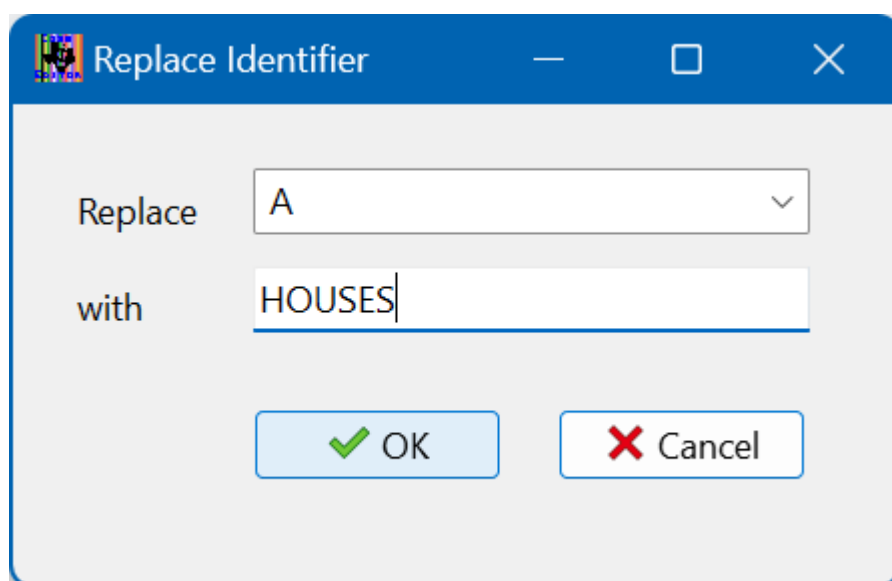
Embedded Utilities

Edit Utilities

Beside the common edit functionalities Cut&Paste, Find and replace, TiCodEd offers some specific utilities for programming SXB.

Replace Identifier

This function not only offers a list of identifiers from the last Build, but also takes care that no sub-strings are changed accidentally



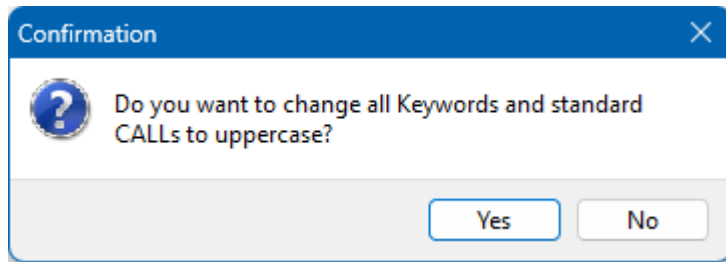
If you replace the identifier A with HOUSES in SteveB52, CALL HCHAR should not become CHOUSESLL HCHHOUSESR:

```
18 PaintScreen:
19 CALL ScrInit(16,2) :: DISPLAY AT(23,10):"SteveB52"
20 FOR I=1 to 26 :: HOUSES(I)=0 :: NEXT I
21 FOR I=1 to 50 :: J=INT(RND*26)+1 :: HOUSES(J)=HOUSES(J)+1 :: NEXT I
22 CALL HCHAR(21,1,133,32)
23 FOR I=1 to 26 :: CALL VCHAR(21-HOUSES(I),I+3,132,HOUSES(I)) :: NEXT I
24 CALL SPRITE(#1,124,13,1,1,0,16) :: BOMB=0
25 RETURN
```

Uppercase Beautifier

You wondered, why are all statements and standard-routines so nicely written in uppercase? Now you know.

This can't be reversed, so use with caution.



Toggle Comment

Sometimes you want to exclude some statements for a test. It is easy for one or two lines to enter a `//` in front of them ... for larger blocks select the lines and select `Edit/Toggle Comment` or simply press `Ctrl-T`. As the word “toggle” suggests, remove the comment the same way.

Advanced Editor Features

The used Lazarus editor offers some special functions

- Indent: Shift-Ctrl I
- Un-Indent: Shift-Ctrl U
- Undo: Alt Backspace
- Redo: Shift-Alt Backspace
- Column Select: Shift-Ctrl C
- Line Select: Shift-Ctrl L
- Normal Select: Shift-Ctrl N

You may hide a code-block by selecting it:

```
7 CALL CLEAR
8 END
9
10 GameInit:
11     CALL CHAR(124,"000000080C0E070FFFFF070F1C38000000000000000")
12     CALL CHAR(128,"28103838381000000000000000000000000000000000")
13     CALL CHAR(132,"FE929292FE929292FFFFFFFF")
14     CALL MAGNIFY(3) :: DIM A(26) :: RANDOMIZE
15 RETURN
16
17 10000
18 PaintScreen:
19     CALL ScrInit(16,2) :: DISPLAY AT(23,10): "SteveB52"
```

and click on the small square in front of the first selected line

```

7  CALL CLEAR
8  END
9  |  ...
16
17  10000
18  PaintScreen:
19  CALL ScrInit(16,2) :: DISPLAY AT(23,10):"SteveB52"

```

Clicking on the square (now with a plus sign) again will expand and display the lines again. Code foldings are temporary and will not be saved.

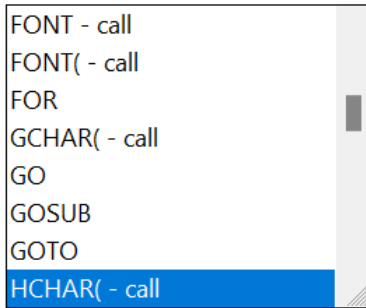
Code Completion

TiCodEd has a simple form of code completion. Just type the beginning of a subroutine and press **Control-Space**.

```

26  HC

```



A window will appear with known expressions, the first matching highlighted. After typing HC and pressing Control-Space, "HCHAR(- call" is highlighted. Pressing Enter will give you:

```

26  CALL HCHAR(

```

You may scroll in that window to select the correct entry.

The following expressions are presented:

- All original XB 1.1 commands and routines
- All routines from the Standard-Library when checked
- All routines from the selected Package (XB256, RXB,...)
- All routines from your program (from last build-run)
- All labels from your program (from last build-run)
- All variables with at least 3 characters from your program (from last build-run)

For subroutines a CALL is added if not already present. The result will be in lowercase when the entry itself is all lowercase. Include at least one uppercase character to get an uppercase completion.

Forward-Navigation

When double-clicking a word it usually gets selected. If the word is a label or a custom subroutine in your program, the editor screen jumps to the label definition or subroutine. It uses the label/subroutine list generated at the last build of your project, therefore not immediately after creating a label or subroutine.

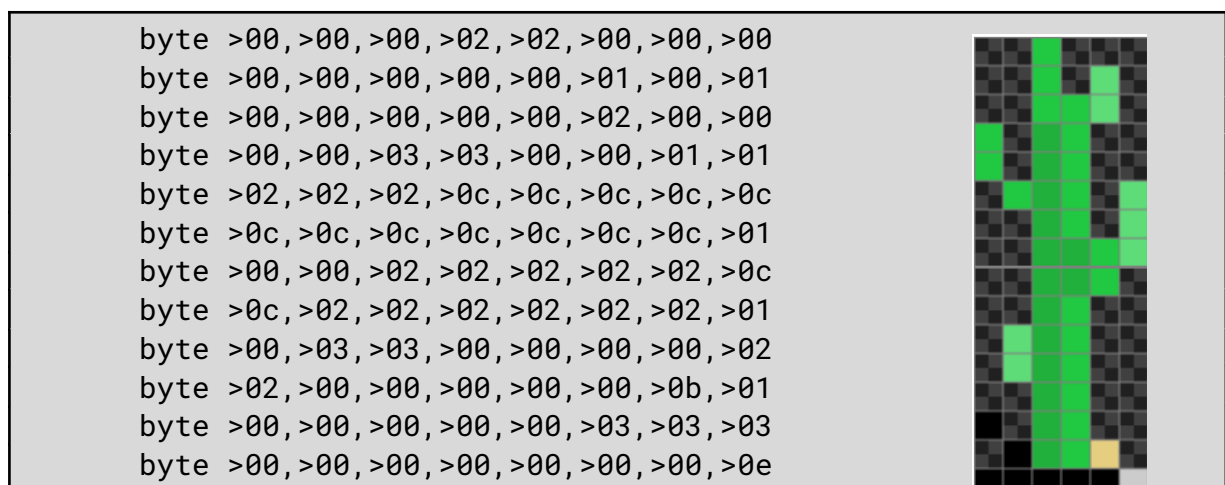
No more searching and scrolling manually.

Hex to BIN\$

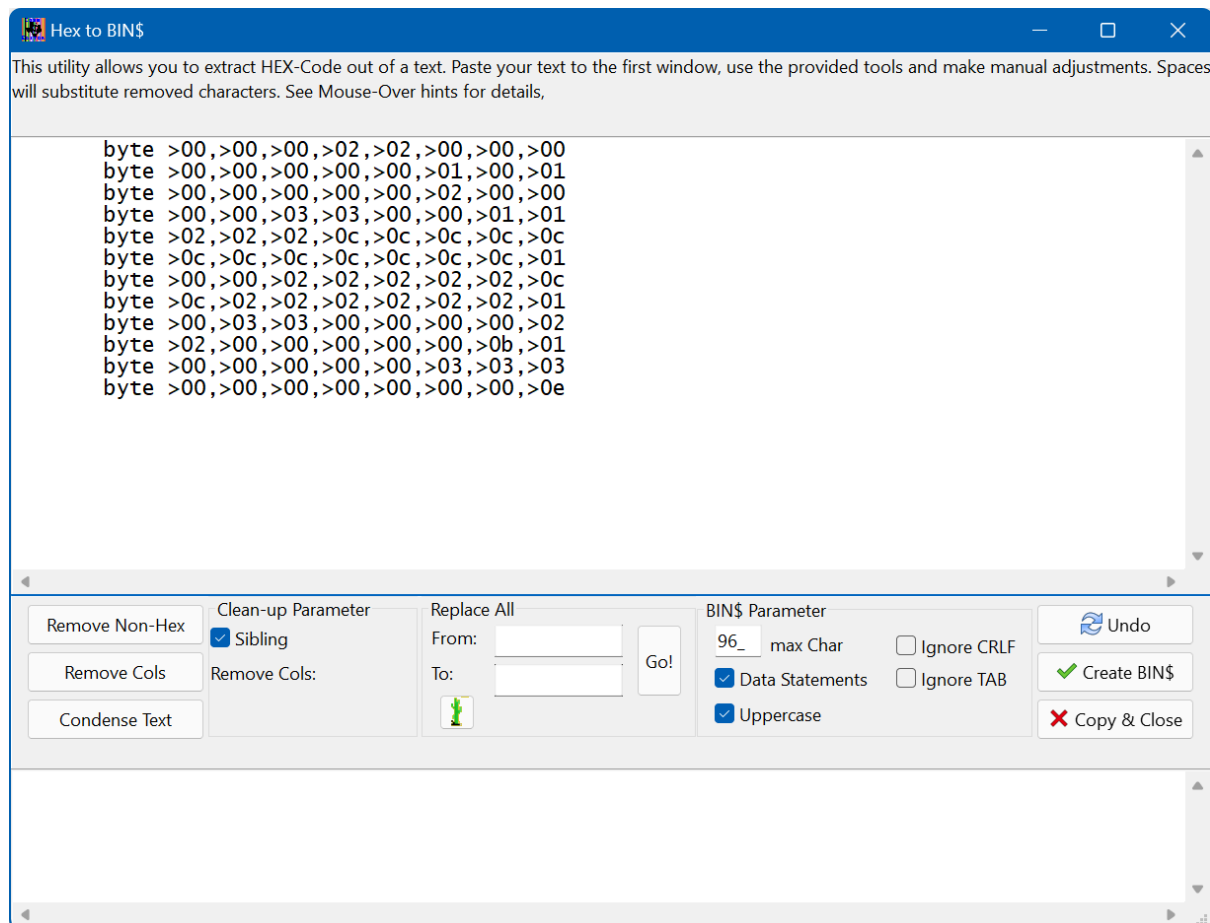
This is a specialized editor to extract code from text.

Sometimes you have a listing file from a tool or scan ... and need the embedded hex-code ... just paste your code in the upper editor and start playing around. Use the Undo if something does not work out (sorry, only one level of undo).

Copy this Assembly export from <https://raphael.js99er.net/>

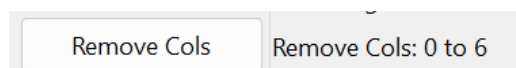


over to TiCodEd Util/Hex to BIN\$ dialog:



You want to get a single hex-string out of this. You may do so with any editor manually and might be faster with this small example than learning the functions of this dialog, but this might change the more or the larger the files are.

So let's start ... first press "Condense Text" to remove space and empty lines. Then select "byte >" in the first line and "Remove Cols" get displayed:



Press "Remove Cols" to remove those columns. This example is simple and there are multiple ways to remove the unwanted characters:

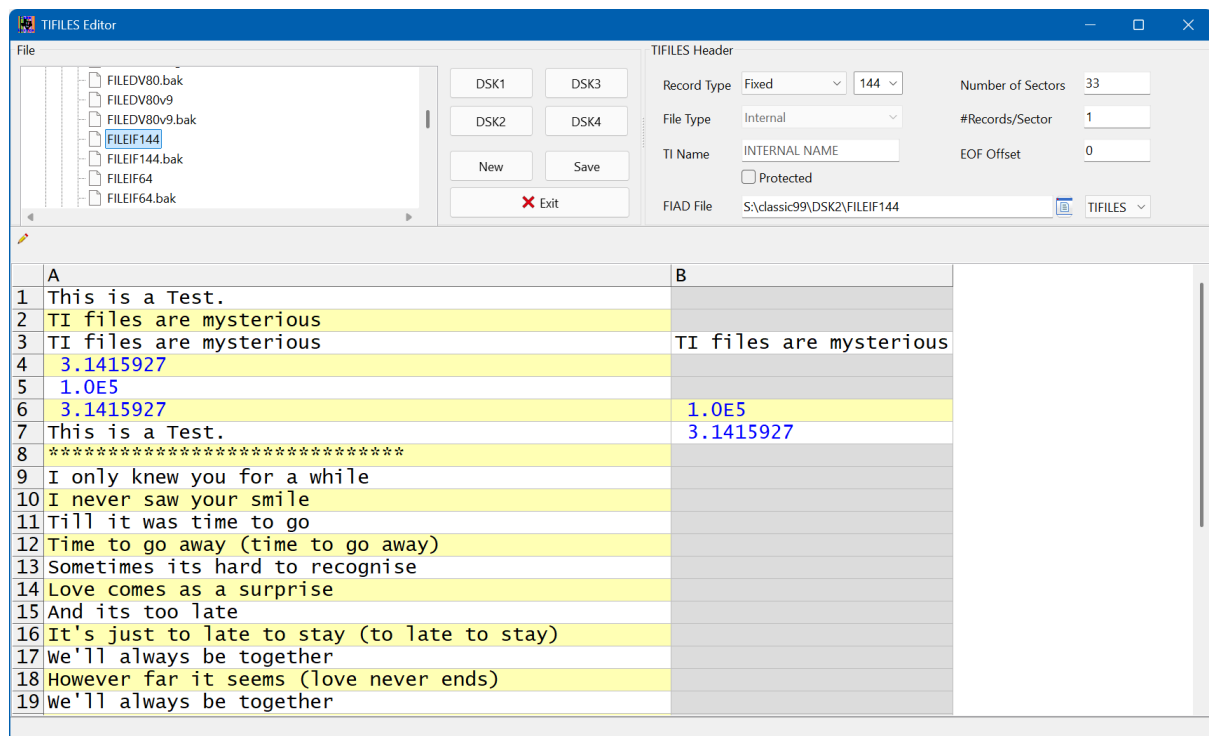
- Select chars in the first row and press "Remove Cols" to delete the marked columns in all rows.
- Use "Replace all" to get rid of repeating patterns .. "To" may be empty.
- "Remove Non-Hex" removes all characters other than 0..9, A..F, a..f;
Check "Sibling" to require at least two of them together.
- Condense Text removes spaces
- Export remaining characters with "Create BIN\$"
 - Ignore CRLF combines multiple lines
 - Uppercase creates only uppercase hex numbers

- you may limit the number of characters in a line
- Data Statements can be added automatically
- When done, leave with "Copy & Close"

The Quick-Action "Saguaro" performs a combination of those steps to extract the shape definition from <https://raphael.js99er.net/> exported by "Assembly data / By column 8 bpp" to be used with the Multicolor library, by also removing the leading 0 from all bytes.

Edit TIFILES

With this editor you are able to open FIAD (File in a Directory) data files created with (Extended) BASIC in Variable/Fixed and Display/Internal format. Type "Internal" files are opened in a grid where each line is a record, each cell a variable, whereas type "Display" files are less structured and opened in a generic editor control.



You may adjust all settings like Fixed/Variable, record length, internal name, file name and header (TIFILES/V9T9) before saving, only the File Type Internal/Display remains fixed as the formats differ too much.

There is no meta-information except the length of a field. Therefore some knowledge of the data is required, especially for 8 character wide fields in an Internal format file. They may represent a string with 8 characters or a RADIX-100 coded floating point number, i.e. "ABCDEFGH" may also be read as 6667.6869707172 resp. 6.6676869707172E3.

The following heuristic is applied: If an 8 byte field is a valid RADIX-100 encoding, it is interpreted as RADIX-100. If it contains bytes below 32, it is considered safe to be a RADIX-100 and shown in blue, if it consists only of displayable characters 32 to 99, it is marked in purple and should be checked, like the above example.

Checks and conversions can be done by a right-mouse button context-menu. You may always convert from Text to RADIX-100 and back. If a field is set to RADIX-100 and does not contain a valid number, the value is shown in red.

You can not save the file with red entries, you need to decide and correct first.

Unused fields are gray and might be populated simply by typing, and deleted (inactivated) with the context menu, where also whole lines (records) may be added or deleted.

Be sure to have a backup before editing any file!

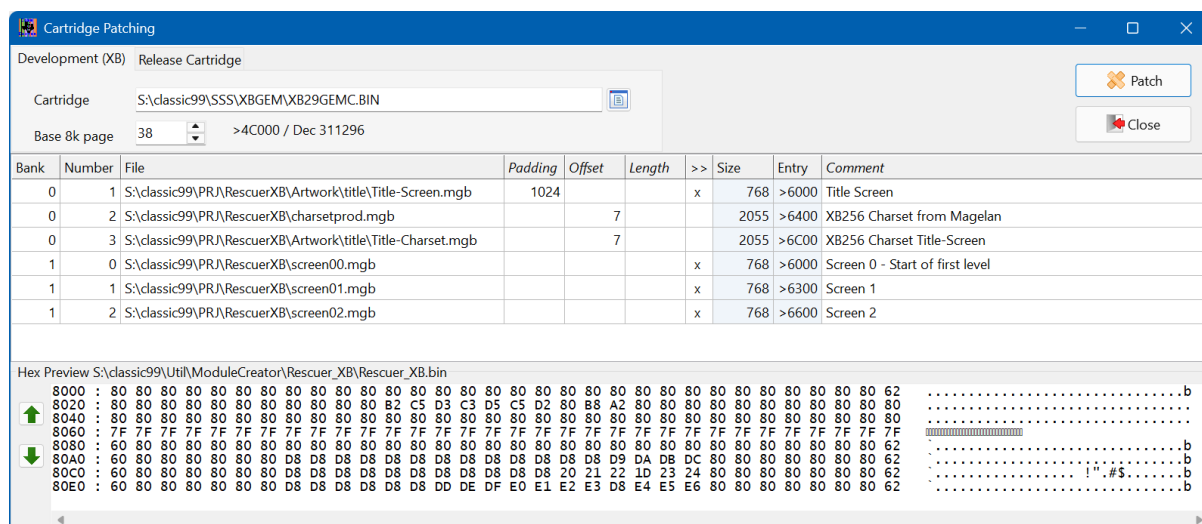
Cart Patching

You may duly ask why you would want to patch a cart and why not doing so with existing tools like a hex-editor. This utility is designed to support ROM bank-switching in your programs. This frees up your 24k + 8k of CPU RAM by moving static data to one or more ROM pages in the module. This may be charsets, music, screens or other level-data.

As Extended BASIC itself uses bank-switching, this is not possible when running interpreted code, only compiled code can safely switch banks.

This gives you the first problem: How to develop something, when you need to build a cartridge and patch it in each and every development cycle? Well, you don't have to. You may patch the Extended BASIC module! You may add additional pages to the TI Extended BASIC 1.1 or use unused pages of a newer Extended BASIC. XB 2.9 G.E.M. comes with a 512kB ROM, but only uses about 300kB. In version 2.920250111 it is safe to use page 38 and above ($38 \times 8 = 304$). For other versions check with a hex-editor where the free space begins. The Cart Patching utility even offers a preview of either the ROM cartridge or the file to be copied into the cartridge.

The dialog has two tabs, one for the XB used for development, and one for the final release cartridge. Both have their own base page setting. A regular ROM cartridge created with the MAKECART8 tool from the BASIC Compiler or the Module Maker by Fred Kaal has 32kB ROM, banks 0-3, so additional pages start at 4.



This is an example of my upcoming game "Rescuer XB", a sequel to my 1984 TI BASIC helicopter game "Rescuer". I currently use two pages. The first page (bank 0) contains the title-screen (file number 1) with the charset for this screen (file #3) and the tiles for the actual game (file #2). The second page (bank 1) contains the first three screens.

While the bank has an actual meaning for addressing (base page + bank gives the page to be updated), the number gives just the sequence of the files in the page. You may number the files 10,20,30 if this suits you better.

The third column gives the source file-name. A single-click allows you to enter a filename, double-click opens a file dialog to select a file (you must not be in edit-mode for the file name to do so!).

The following columns need some more explanation.

Padding specifies to which length the input file will be padded, meaning filled up with zero bytes. Even though the title screen has only 768 bytes (32 x 24 screen), there will be 1024 bytes written in the first line.

Offset is used to skip the first bytes of a file. Charsets created with Magellan have a 7 byte header which needs to be skipped. When a file has a TIFILES header, skip 128 bytes.

Length is the number of bytes written to the cart. If omitted, the file will be used to the end of the file.

>> (Shift) is an important flag, but a little difficult to explain. The TI graphics chip has limited options to where tables can be put in video RAM. For Extended BASIC TI decided to use some overlapping areas for the screen and the character definition table, using only chars 30 to 142. This results in a shift of 96 for all characters in video RAM. If you want to display an asterisk * with ASCII code 42 on the screen, you need to put a 42+96=138 into the video RAM. Screen 2 of XB256 follows the same shift-logic.

If you want to use a binary file where characters are represented by their ASCII values, check this column with an "x" to get the data shifted by 96. When you

enter a number, this value will be used instead. Screens exported from Magellan with the option "Export/Binary Map (current)" can directly be moved to VRAM using CALL MOVE with this option.

Size displays the size of the input file, not the size of the copied chunk. In the second and third line you see the size of 2055, which is 2048 plus 7 byte header.

Entry shows the resulting entry address of this file within the ROM page. It will be filled/updated when patching a cartridge, not while editing the table. The ROM pages are always mapped in the address-space >6000 to >6FFF. Selecting a page from Extended BASIC is simple and will be demonstrated later in this chapter.

Comment may be used to give some more information about the file and the parameters used. Commenting is always good, right?

When clicking on a line or in the cartridge field, the preview on the bottom of the dialog gets updated and shows the first 256 byte in hexadecimal and ASCII of the file or base page of the cartridge. You may move 256 up or down using the arrows on the left side.

Your settings will be stored when leaving the dialog with "Close" in a file named like your project with the extension .cart, which is a comma-separated text file.

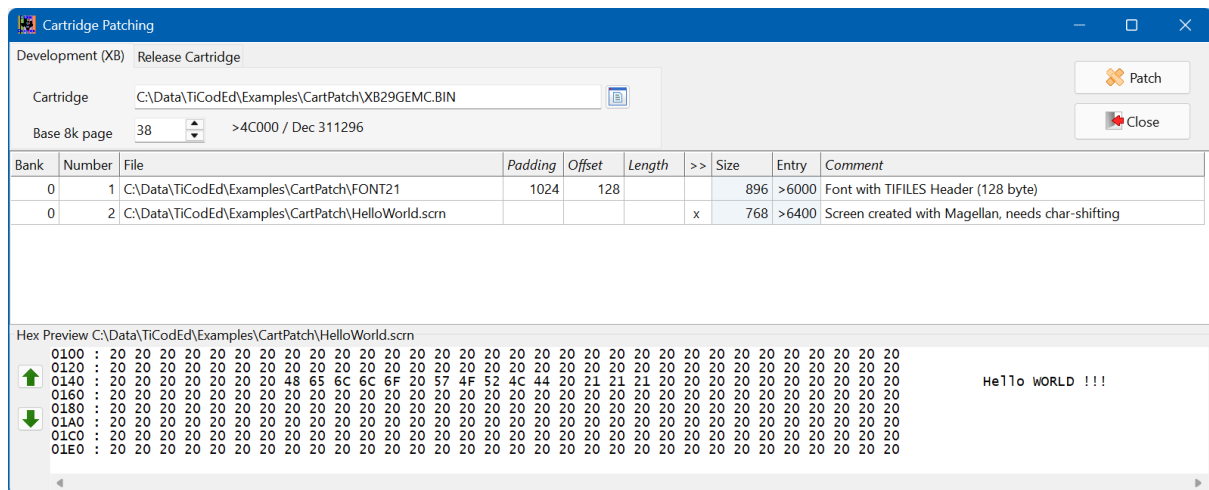
Use the "**Patch**" button to apply the patch to the selected active cartridge. The results will be shown in the log tab of TiCodEd.

Let's use the provided example in the subdirectory CartPatch:

Open the file CartPatch.sxb with TiCodEd. Go to the Cart Patching utility by selecting Util/Cart Patching.

The two files, a font and a screen-dump should already be configured from the CartPatch.cart file.

Update the Cartridge path to your Extended Basic cartridge or create a project specific Extended BASIC by copying both cartridge files (GROM and ROM) to a project location.

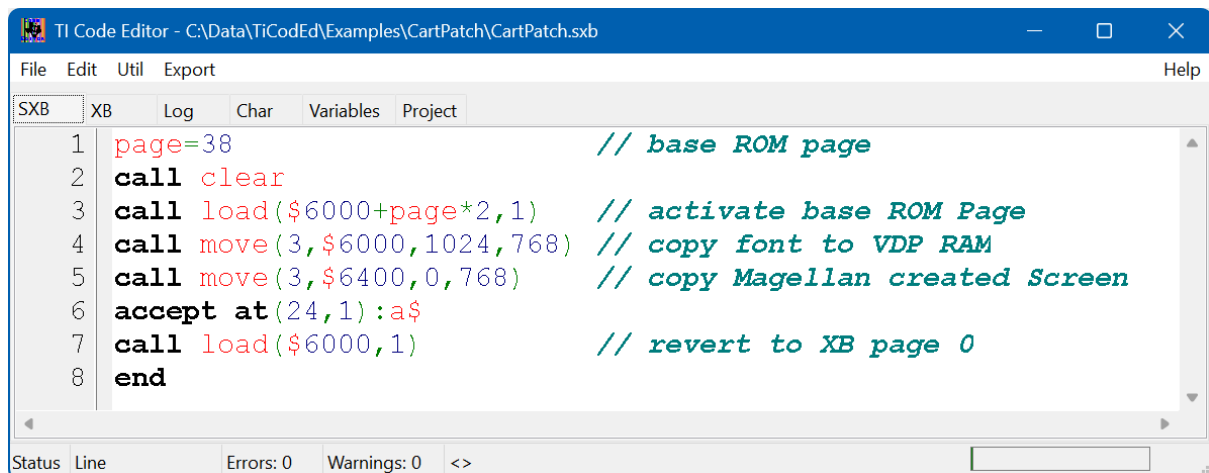


Check the values and hit the "Patch" Button. You need to reset Classic99 by selecting File/Cold Reset to reload the modified Extended BASIC cartridge after each patching.

Now we come to the usage of these two files from Extended BASIC. I suggest always using a variable "page" or "basepage" to store the base page, as the word "base" is a keyword in Extended BASIC. This variable can easily be changed to compile for a cartridge creation to another value, most likely "4".

You may select a ROM page with a simple CALL LOAD to the ROM area starting at >6000. Just add the desired page times two: >6000, >6002, >6004, etc for Page 0,1,2 and write any value to this address.

Then you can access this 8k area with CALL PEEK and very efficiently with CALL MOVE.



Before returning to Extended BASIC make sure to reset the ROM page with a CALL LOAD(\$6000,1).

Check the XB256 manual Page 11 for the addresses. When not using XB256 use only the screen 1 addresses:

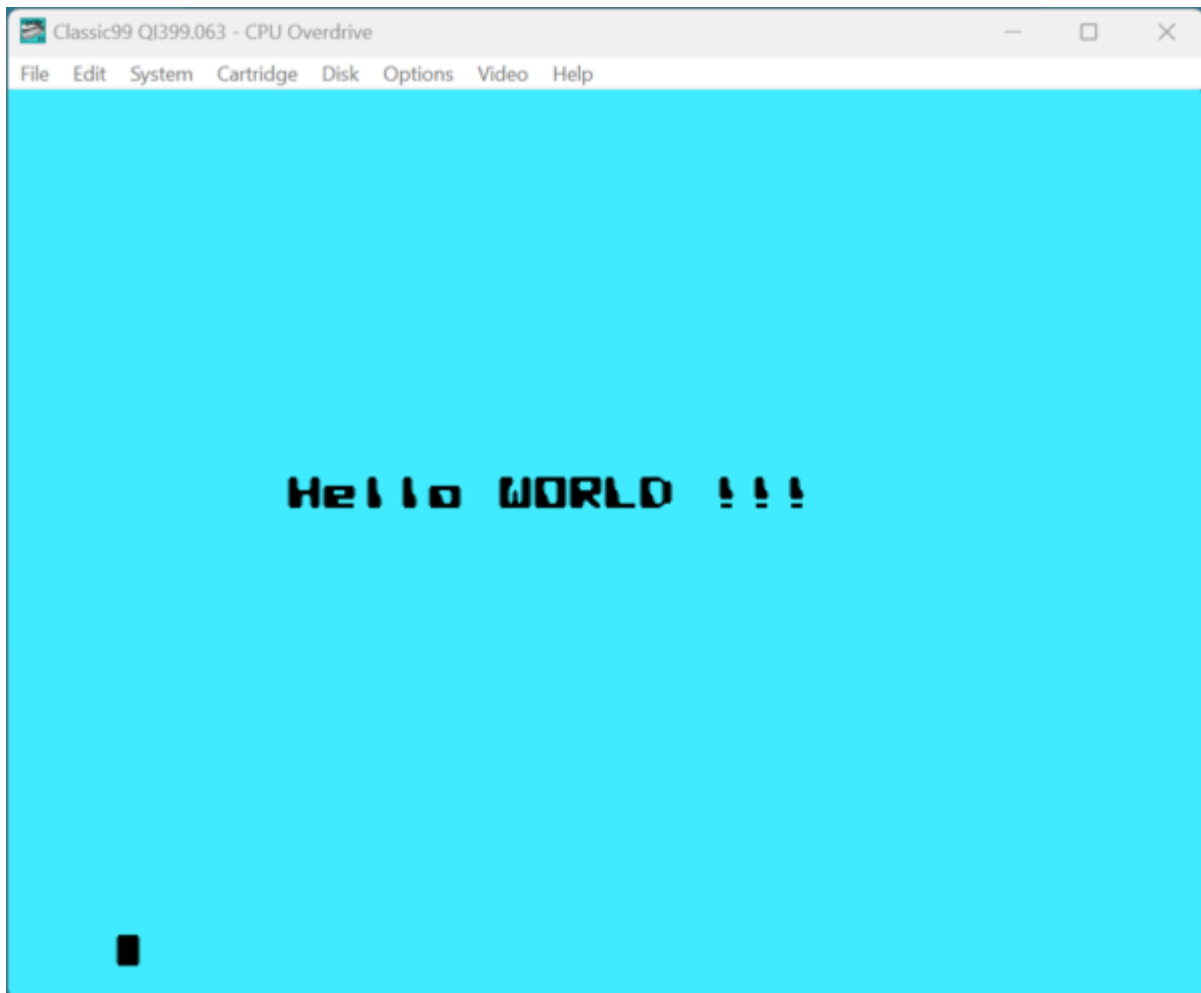
- The character pattern table for screen 1 starts at 1008 for char 30 (the cursor), the font-file starts with character 32. Each character needs 8 bytes. Therefore the font is loaded at address 1024 (1008+2*8).
- The screen starts at VDP RAM address 0. The Magellan file will get the 96 shift during patching, so it can be moved directly to VDP RAM.

From the XB 2.9 G.E.M. manual:

CALL MOVE(mode, start address, target address, # of bytes) moves a block of memory. The four modes available are:

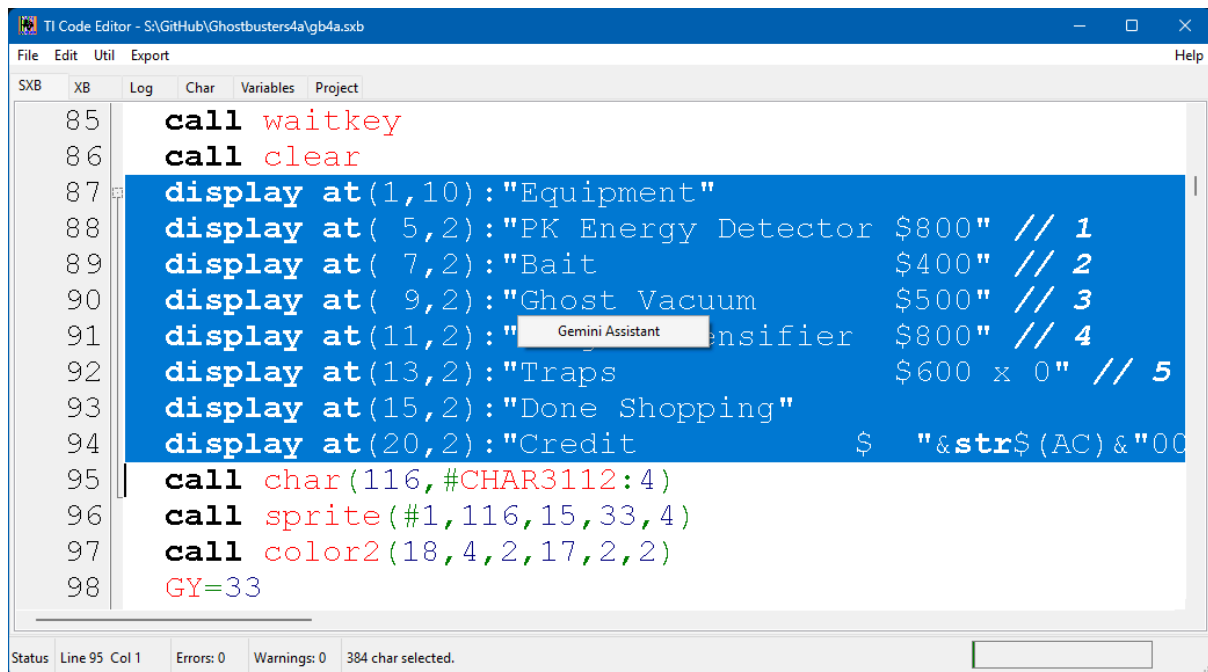
- 1 - from VDP RAM to VDP RAM,
- 2 - from VDP RAM to CPU RAM,
- 3 - from CPU RAM to VDP RAM,
- 4 - from CPU RAM to CPU RAM

Compiling and running the program gets you:

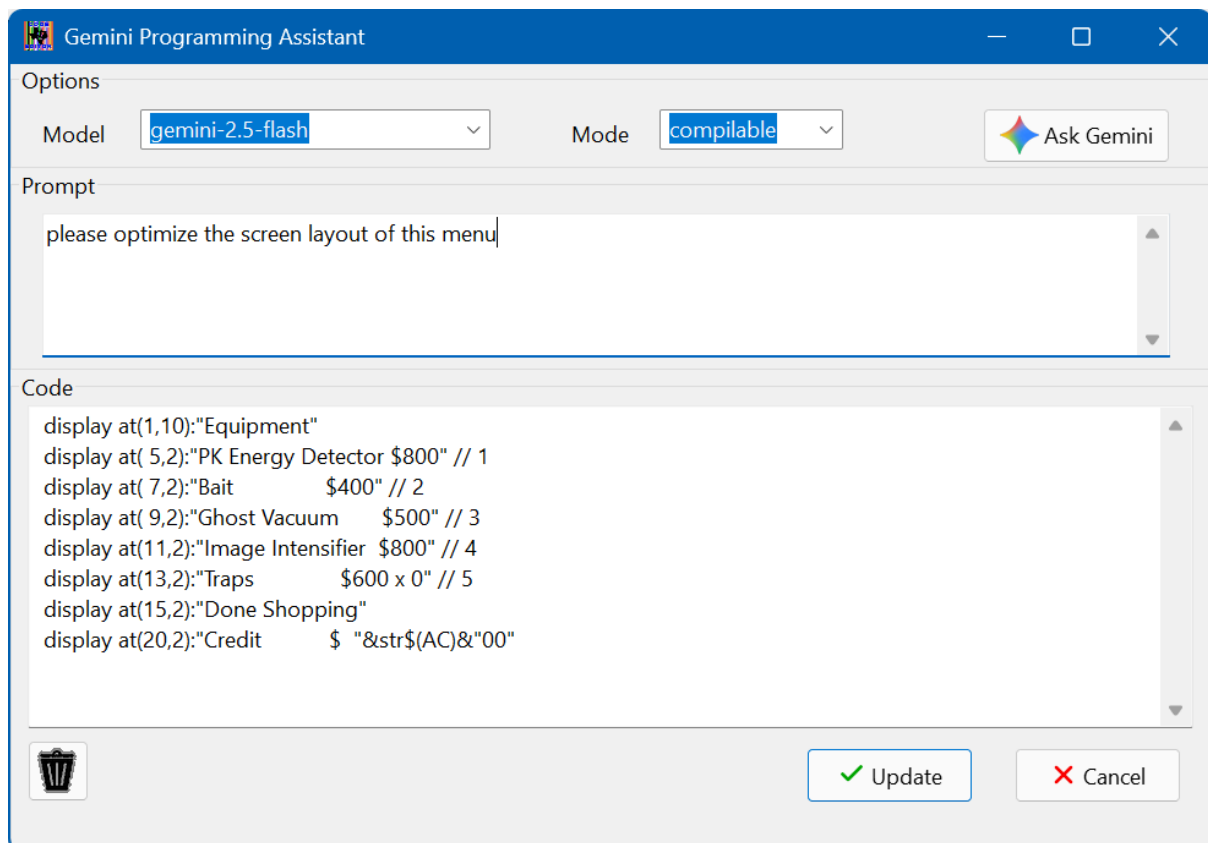


Gemini AI

Once configured, you can simply select some code, press the right mouse button for the context menu and select "Gemini Assistant".

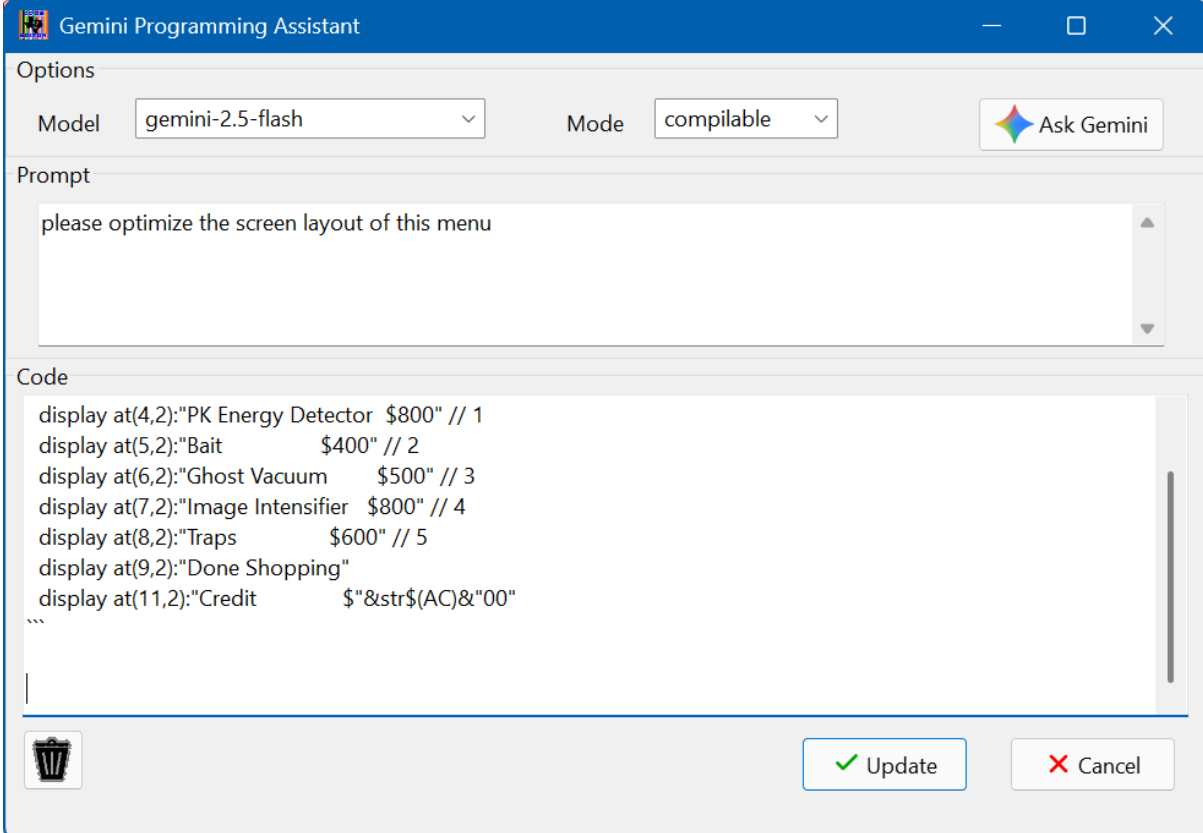


This will bring up the "Gemini Programming Assistant" and your selected code is copied to the Code box.



Select a model to use and a mode: Interpreter, compilable or Compiler only. This will tell the AI which statements may be used, i.e. the DEF statement is not supported by the compiler, but by the XB interpreter.

Enter a prompt (request) and then press the "Ask Gemini" button to call the AI. The result will be shown in the Code box, changed rows in this example.



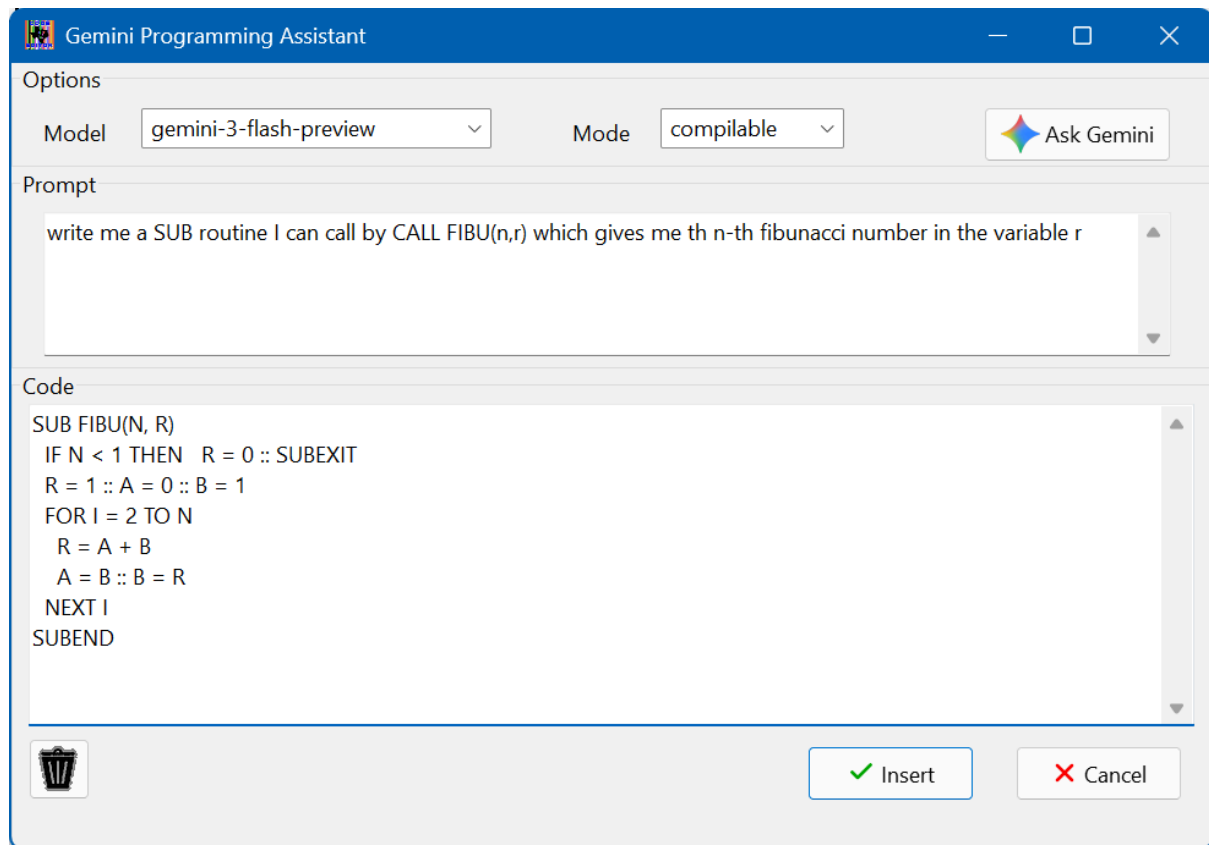
The screenshot shows a window titled "Gemini Programming Assistant". It has a blue header bar with standard window controls. Below the header is a section labeled "Options" containing two dropdown menus: "Model" set to "gemini-2.5-flash" and "Mode" set to "compilable". To the right of these is a button with a Gemini logo and the text "Ask Gemini". Below the "Options" section is a "Prompt" section with a text area containing the text "please optimize the screen layout of this menu". Below the "Prompt" section is a "Code" section with a text area containing the following code:

```
display at(4,2):"PK Energy Detector  $800" // 1
display at(5,2):"Bait          $400" // 2
display at(6,2):"Ghost Vacuum    $500" // 3
display at(7,2):"Image Intensifier  $800" // 4
display at(8,2):"Traps          $600" // 5
display at(9,2):"Done Shopping"
display at(11,2):"Credit        $"&str$(AC)&"00"
...
```

At the bottom of the window, there is a trash can icon on the left, and two buttons on the right: "Update" with a green checkmark and "Cancel" with a red X.

If you are happy with the result press "Update" to replace the selected text with the result. Otherwise start a new prompt or press cancel. The history of your conversion is logged and included in every new request. Press the dustbin icon to delete the chat history.

You may also create new code without selecting something, just using the context menu, and write a prompt:



Why Google Gemini?

There are actually two reasons. For one, Google is constantly improving Gemini with regular updates, enhancing it while being predictable. Secondly, Google is the only supplier offering a free subscription for the API use. Other models may be better at times, but they are behind a paywall.

The free subscription was scaled back in late 2025, but is still sufficient for private, amateur use. The models are changing fast and you can check which models are available and how many calls per day are available in the free subscription, or in your paid subscription in Google's [AI Studio](#) (RPD = Rounds per Day). Beside the well known Gemini models, the Open Source Gemma 4 and the proprietary licensed Gemma 3 are available.

How to tweak

1. Manage the number of calls - Different models have different strengths. The newer models are usually more sophisticated than their predecessors, the Pro models are reasoning more than Flash models, Flash-lite models are the weakest, but very straight-forward, while Pro models have a tendency of "over-thinking" a request. Choose wisely, when Flash and Flash-lite models might be sufficient. Pro models have been excluded from the free tier lately.

2. The context window is important. This parameter describes, how many tokens a model can handle in one request. TiCodEd comes with multiple context files to tell the AI about Extended BASIC, the SXB extension, XB256, the Compiler, RXB etc., depending on the current project settings. Additionally, not only the selected part of the program, but the whole program is included in the prompt, to complete the context of the prompt, as are previous prompts to form a dialog. 32k tokens is the minimum. You can get an overview over the almost daily changing models on the [pricing page](#) and [models page](#).
3. The context files may be found in the `./lib/ai` directory. They are brief manuals, like the TI reference cards, amending to the general programming knowledge of the model. Before calling the API, the prompt is saved to the TiCodEd directory in the file *LatestPrompt.txt*. If you notice, that a statement or subroutine has been used incorrectly, check if the context file may be inaccurate. Please share corrections on AtariAge.
4. The models are maintained in the file `/lib/ai/models.txt` file. You may recognise the names from the [pricing page](#). There are frequent updates to the models so this file gets outdated fast. Updates may be found on AtariAge in the [TiCodEd thread](#).

TiCodEd Built Notes

The Windows Versions

TiCodEd is developed with Lazarus in Pascal, the source code is included. The Windows installation contains two executables. TiCodEd.EXE is the 64 bit version, TiCodEd32.EXE is the 32 bit version provided for compatibility with older Windows installations.

The Mac Version

TiCodEd is developed with Lazarus in Pascal. Lazarus is available for multiple platforms, not only Windows. Early versions have been built for Intel Macs, now the ARM architecture is used. Some (few) functions are using Windows functions and are not portable.

See the Difference to the Windows version:

- The Keyboard-Emulation can't be activated in the Preferences, as it uses the Windows Messaging functions.
- The HTML-Export is broken for Mac in Lazarus, therefore the menu entry has been removed.
- The tabs-widget on the Char Tab differs from the native Windows implementation, leaving an empty row and compact labels, but is fully operational.

- The INI file is not stored beside the executable as in windows, but within the Package-File TiCodEd.app under Contents/Resources.

A major difference is the lack of a Mac emulator supporting FIAD. You may use the Post-Processing command to add the tokenized files to a disk-image and start an emulator.

There has been very little feedback on the ARM versions and I'm not using MacOS myself, so please consider this version "experimental".

The Linux Version (x64)

Just like with the Mac version not all features are available in Linux. The 64bit binary distribution is compiled using Lazarus 4.4 on Linux Mint 22.3. If this does not work for you sources should compile on any Linux (or FreeBSD, ...) version where Lazarus is available, i.e. Raspberry Pi.

The most used TI-99/4a emulator under Linux is MAME, not supporting FIAD. You may use the Post-Processing command to add the tokenized files to a disk-image and start an emulator. MAME unfortunately loads the disk content on start-up, so changes to a disk-image require MAME to be closed and started again.

The Windows-Version is reported to work together with Classic99 under WINE.

The Keyboard-Emulation can't be used under Linux, as it uses the Windows Messaging functions, but a named pipe may be used to interface to Emul99 which supports FIAD (<https://github.com/statpascal/emul99>):