FVM Factory overview

Rev 0.4 dated 10/2/2006

FVM Factory is a set of Win32 programs supporting Forth Virtual Machine (FVM) development for embedded applications.

Why FVM?

FVM is very simple, compact, fast and powerful. These qualities ensure that FVM is well suited to be an interpreter engine in embedded applications, where smaller code size is an advantage.

FVM is a virtual stack processor. It employs two stacks, Data Stack and Return Stack. All calculations made on Data Stack (or simply Stack). Return Stack is more specific, mainly used for holding return addresses of subroutines.

Since a stack does not need explicit addressing, a stack machine is a 0-address processor. This means that opcodes for the stack processor do not include the addresses of operands. As a result, opcodes are short, and programs written for FVM are very compact. To learn more about stack computers please refer to the excellent book of Philip J. Koopman, Jr. Stack Computers: the new wave.

Is it yet another Forth?

No.

Typically Forth is a self-contained system. Once Forth is ported to a target, it can do anything. It serves you as compiler, interpreter, debugger, or even as an operating system. However, in return Forth requires that you write your programs in Forth language.

FVM is just a virtual processor, or a bytecode interpreter. Forth usually uses FVM, however not all FVMs may be compatible with Forth.

FVM Factory uses a cross-compiler. Compiler functions for the target FVM are not included into the target. The compiler for a given FVM exists as a PC console application.

FVM Factory helps create application-specific FVMs. It is not concerned with Forth standards compatibility. Many Forth features are not implemented by FVM Factory, because its goal is to make the FVM code as small and as simple as possible.

The created FVM is written in ANSI C language. It can be included as an interpreter engine into an embedded application written in C. Tokens or pre-compiled functions (scripts) can be invoked from C code.

FVM Factory uses some elements of C language for scripts, such as #define. Also, both Forth style and C/C++ style comments are allowed in scripts.

What kind of FVM will be created?

FVM Factory creates 16-bit FVMs with indirect threading code. FVM bytecodes can be 1-byte, 2-bytes and 3-bytes long.

FVM console is case insensitive. Thus, the scripts for FVM will also not be case sensitive.

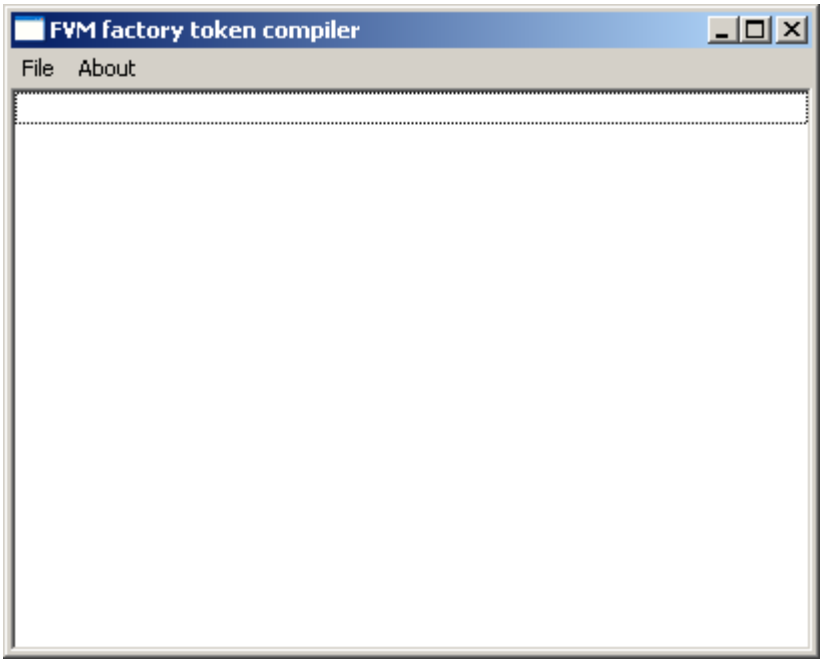
How to use it?

Unzip FVM_factory.zip into a working folder. There are the following subfolders in the archive:

-- **Console**
-- **Doc**
-- **Target**
-- **Tok_compiler**

Step 1. Compile FVM_factory from the supplied source code

Go to the **Tok_compiler** folder. Using Lazarus http://www.lazarus.freepascal.org/ compile the supplied FVM_factory project. The resulting program is FVM_factory.exe. It is a token compiler; it converts a token's description file into a set of C template files. When FVM_factory.exe is started it should look as follows:



Step 2. Edit your token's description file

Using any text editor, edit your token's description file. There is a sample file **tokens.bcd** in the folder **Console**; it can be used as a template. This file should list all the tokens for your virtual machine. Your token description file should be in the **Console** folder as well.

There is a minimum set of tokens required for compiler to operate properly; those tokens are listed in the file **required_tokens.c** in the folder **Console**.

Your token's description should obey the following simple rules:

- Only line comments are permitted. The line comment should start with double slash, //.
- Directives #1, #2 and #3 specify that following tokens will have length of 1, 2 or 3 bytes respectively. Those directives must be the first word of a line; the rest of that line is treated as comment.
- Every token description should occupy one line.
- The first word in the line should be a unique token name complying with the rules for function names in C language. Chars like ".,<' etc are not permitted. The FVMFactory compiler will convert this name into lower case and then create a C function prototype,

adding the letters "fvm" in front. All created FVM functions are void (void).
For example, if a token name is DUP, then FVMFactory will create the following definition:

```
void fvmdup(void)
{
} /* end */
```

- A second word in the line is optional. This word is a token name complying with the rules of Forth language. If the second word does not exist, the FVMFactory will use the first word as the token name.
  For example, if the required Forth word is >R, then the C-name can be "TOR", and the token definition can be as follows:

```
TOR     >R
```

- Stack comments start with a stand-alone left brace. The rest of the line after the left brace is considered to be a meaningful comment for the token, and it will be included into the embedded help system. Working with the console application you will see this comment when you type "help MyToken", where "MyToken" is the token name. Please note that the comment will be printed by the standard C printf function, and you can use printf formatting symbols.
  For example, here is a definition of token DUP including help information:

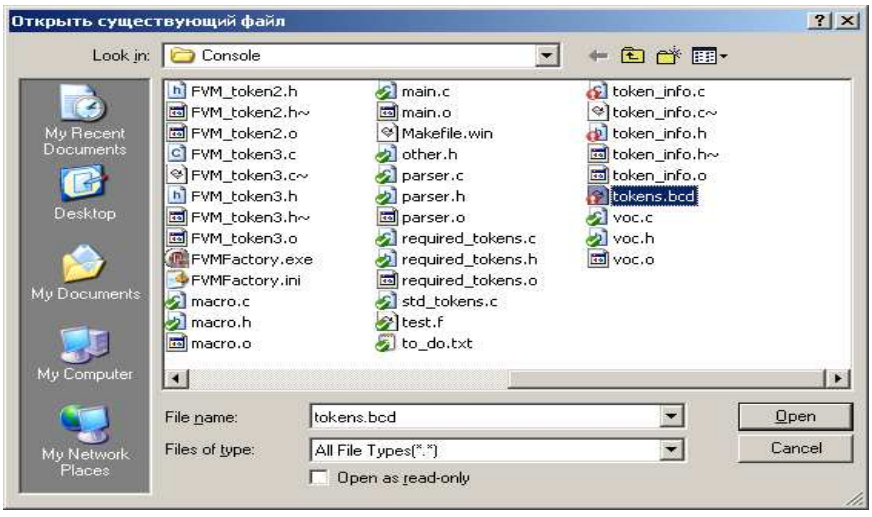
```
DUP       ( x -- x x )\nDuplicate x
```

Step 3. Compile tokens description

Start FVMFactory.exe.

Click **File/Target directory** and select a C file in your target folder where the embedded FVM should be. This tells FVMFactory where to put a second copy of the compiled files.

Click **File/Open** and select your token description file in the **Console** folder.



FVMFactory will compile a description file and will create/update two sets of C files.

In the local folder (in our case - in the **Console** folder) it will create/update the following files:

-- FVM_token1.c, FVM_token1.h
-- FVM_token2.c, FVM_token2.h
-- FVM_token3.c, FVM_token3.h
-- FVM_exe.c, FVM_exe.h
-- token_info.c, token_info.h

In the target folder it will create/update the following files:

-- FVM_token1.c, FVM_token1.h
-- FVM_token2.c, FVM_token2.h
-- FVM_token3.c, FVM_token3.h
-- FVM_exe.c, FVM_exe.h

If those files already exist in their respective folders, FVMFactory will make backup copies. It will also parse the existing FVM files, trying to find any already written C code for the specified tokens. If the code exists, FVMFactory will copy the content of C functions into the files being updated. This supports incremental code development.

FVMFactory also checks if a file called **std_tokens.c** is present in the local folder (eg in the **Console** folder). If this file is present, FVMFactory will parse it trying to find C functions matching tokens' definitions there. If successful, FVMFactory will copy the body of the matching C functions directly into the FVM switch function situated in the **FVM_exe.c** file. In this case, a C function in the respective file FVM_tokenX will not be created. In other words, FVMFactory considers the file **std_tokens.c** as a kind of repository storing already created and debugged FVM tokens.

Step 4. Add C code to your custom tokens

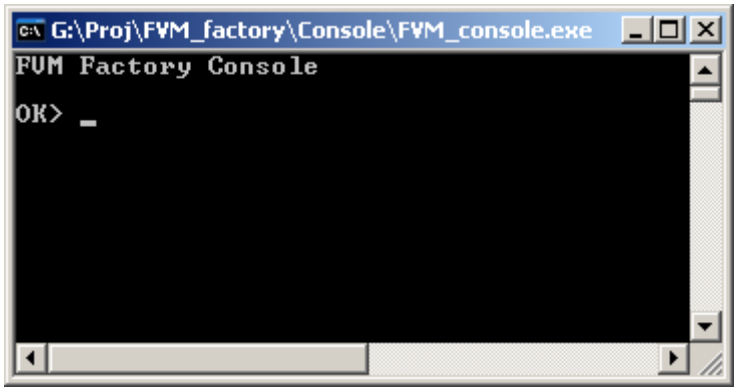
Edit the files created by the FVMFactory in the **Console** folder. To access stacks use FVM primitives defined in the **FVM.c** file. Please refer to the code in the **std_tokens.c** file to see how those primitives can be used.

Step 5. Compile console application and debug your tokens

FVM Factory console code has been developed using Dev-C++ 4.9.9.2. You can use other C compilers and IDE of your choice. The following procedure is described for Dev-C++.

Download and install Dev-C++ from SourceForge, http://sourceforge.net/projects/dev-cpp

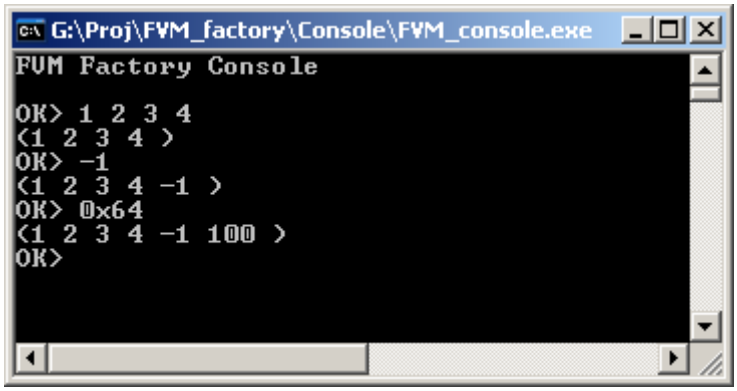
In the **Console** folder, open the Dev-C++ project file **FVM_console.dev** and compile the project. The resulting console application should look as follows:



Test and debug your tokens. To execute a token just type its Forth name, as FVM console is

already aware of your custom tokens.

To put a number on top of the stack, just type it. FVM console compiler will convert the number into appropriate literal tokens.



The number can be decimal or hexadecimal, depending on the base. The default base is decimal. To change the base to hex, type:

`HEX`

To change the base to decimal type

`DECIMAL`

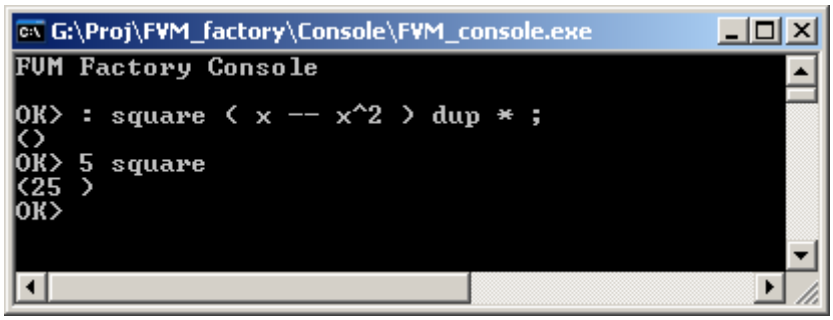
To enter hex numbers while the base is decimal use C conventions. For example, to enter hex number F5, type:

`0xF5`

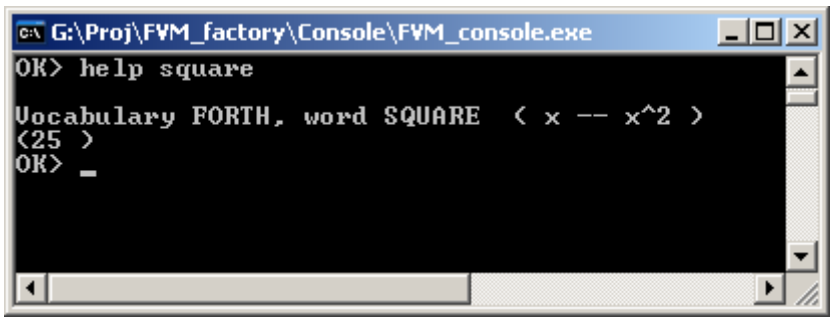
Repeat steps 2 to 5 to add more tokens as required.

Step 6. Create and debug scripts for your FVM

Script words are defined in the same way as in Forth. For example, let's define the word "square" and test it:

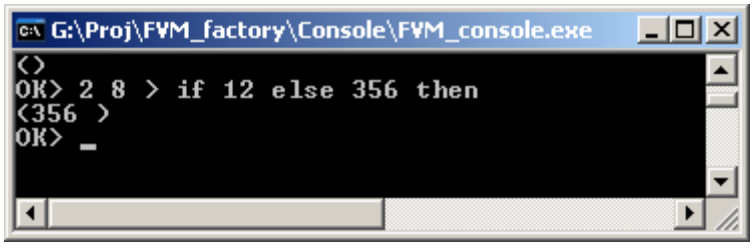


Block comment and line comment are automatically included into the help system:



FVM console compiler is aware of the following Forth control statements:

IF ELSE THEN

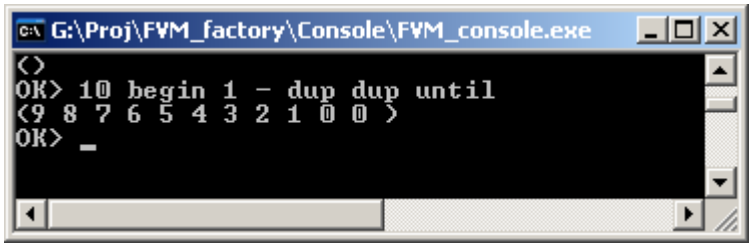


If-else-then statements can be used both in command line and in word definitions. They can be nested.

DO LOOP

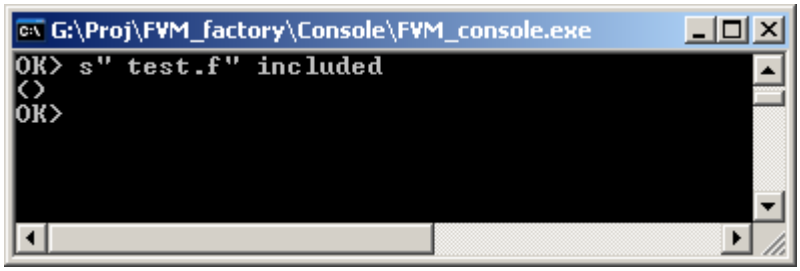
Do-loop is a loop with a counter. FVM console also is aware of related Forth words I, J, LEAVE.

BEGIN UNTIL



Another loop is the begin-until loop with end condition checking.

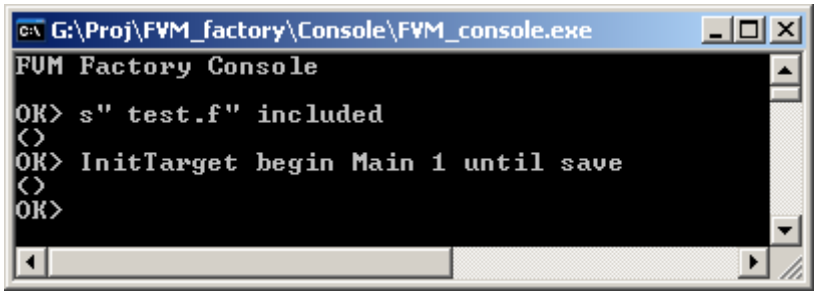
Script definitions can be stored in an external text file. To compile an external file use the word **INCLUDED** as shown in the following example:



Files can be nested; the nesting depth limit is 100.

Step 7. Cast compiled scripts into target application

When all scripts have been compiled and debugged, the resulting bytecodes can be cast as a C source file (array). To cast your code type **SAVE** at the end of the line.



The result is stored in the file **FVM_code.c** in the **Console** folder. For example, the content of that file might look as follows:

```
/* FVM Factory */
/* Bytecodes compiled by FVM_console */
uchar FVM_code[] = {
75, 0, 21, 70, 87, 70, 128, 70, 255, 70, 144, 60, 77, 0, 3, 60,
6, 7, 45, 18, 60, 77, 0, 12, 77, 0, 16, 2, 73, 250, 255 };
```

The first three bytes in that example constitute a 3-byte token, _LGOTO 21. It is generated automatically by the FVM_console compiler. Address 21 is the address of the first compiled command related to the line "**InitTarget begin Main 1 until**".

Addresses 3 to 20 are filled by bytecodes compiled when the file **test.f** was included. The file stores a user's definitions, including the words "**InitTarget**"and "**Main**". At address 21 you can find the 3-byte token _LCALL 12. This token calls the "**InitTarget**" word; its definition starts at address 12 and ends at address 15 by a 1-byte token, RETURN.

There is a 3-byte token, _LCALL 16, at address 24. It calls the "**Main**" word, which is at address 16. Next is a 1-byte token, LIT1. Finally, a 2-byte token, _SJNZ -5, relates to the word "**until**".

As you can see, the compiled code is quite compact. Unlike Forth, it does not store any link fields or word names. FVM_console takes care of those issues.

Actual bytecode values depend on the order in which tokens are listed in the description file (see Step 2). For your own FVM, these codes might differ from the given example.

Step 8. Compile your target application

Copy all source files whose names start with FVM_ from the **Console** folder into your target folder and compile your target application. Here is the list of the required files:
-- FVM_token1.c, FVM_token1.h
-- FVM_token2.c, FVM_token2.h
-- FVM_token3.c, FVM_token3.h
-- FVM_exe.c, FVM_exe.h
-- FVM.c, FVM.h
-- FVM_code.c

The target folder also should have the following files:
-- other.h
-- config.h