Virtual eXecuter – The VX platform

A virtual machine for virtually any microcontroller

Table of contents

Preface

The VX virtual machine is designed for small embedded systems originally intended for 8 bit microcontrollers.
When building the virtual machine (VM) core a user provided configuration file is used to configure the system. This configuration file specifies which type of program memory is available, how (if at all) errors must be presented and handled and all the specifics of the hardware.

The preliminary focus is to define and implement a working VM. Execution speed and core size is not the focus of the first version and so it’s not expected to outperform any existing controller.

The instruction set is designed to allow for a short leaning period and will therefore look fairly similar to that of a standard 8 bit microcontroller. Two exceptions from this are that the instructions are stack based and that threading is supported directly on the lowest level (threading is not supported yet!).

Inspiration for the VX has been taken from the AVR, the PIC, the 8051, the x86, and the ARM hardcore microcontrollers, and the MicroBlaze and PicoBlaze softcore microcontrollers. Java byte code has also contributed to the development.

Revision track

0.01	Jan 30 2008	Document started. Created from the first bits and pieces of scribbled down notes and ideas. This is written before any code has been implemented and so represents preliminary thoughts and I-think-this-is-how-it-should-be.
0.02	Feb 12 2008	First published version of this document. Instruction set seems to be complete... Threading has been left out for now.
0.03	Feb 28 2008	Decided to change the way the VM works. It’s now a little more complex than the average 8 bit microcontroller in the way functions works. The tool chain has been expanded quite a bit and a compiler is now under development (thanks to Arild Boes). The new assembler (still under development) produces linkable .part files instead of directly generating executables (.vxx files). This is really the way it should have been made from the start.

Data types

The VX instruction set operates on the following data types

- Single. A single byte representing an integer number.
- Double. Two bytes representing an integer number.
- Quad. Four bytes representing an integer number.
- Float. Four bytes representing a floating point number in the IEEE 754 format.

Programs

VX applications stored on offline storage are called programs. Programs consist of three sections

- Meta
- Code
- Const

The Meta section holds information on the program that enables the program loader to load the program into memory. The code and const sections are the resulting sections from the linking process.

When a program has been loaded into memory and

To run a program it must first be loaded into memory.

The VX machine operates with a number of memory sections. The meaning of these depend on which level they appear in.

- Code. Non volatile. Read only. This is the only section that can be executed.
- Const. Non volatile. Read only.
- Data. Volatile. Both readable and writeable.
- Stack. Readable and writeable through push and pop instructions.

The status register

The status register holds a number of flags that reflects the results and status of the previous instructions. The status register is not directly readable or writeable but reflects and affects the results of instructions.

The following flags is are placed in the status register

- Zero (Z)
- Carry (C)
- Two’s complement overflow (O)
- Negative (N)
- Sign (S)

The interpretation of these flags depends on the instruction being executed and will therefore be described together with the individual instructions.

Program execution

When a program is loaded four base pointers are initialized. Each pointer points to the start of each section of the program. At the same time four size variables are loaded with the size of each section. This way sections can be placed freely in memory by the loaded as the base pointers are added to all memory location values in the program. The size variables are used to detect out-of-range access attempts. If out-of-range access is attempted the program is terminated prior to access.

Instruction set

Each instruction is defined by a unique number in the range 0 through 255. I.e. each instruction op code takes up exactly one byte in the program storage media (not including the constants required by some instructions).
Not all op codes are currently in use. Executing an unused op code terminates the program.

Values passed to the instruction directly from program memory are called constants. Far from all instructions require constants as most use arguments pushed from the stack.
In assembly language constants are placed right after the instruction mnemonic separated by a space.
Constants are placed on the memory location(s) immediately following the op code – little endian style where appropriate.

As the VX VM is stack based all arguments for the instructions must be pushed onto the stack prior to executing the instruction.
Each instruction requires a specific number of values being pushed onto the stack. This number is referred to as the instructions argument count.
The argument with the lowest number is the first one to be pushed onto the stack.
When the instruction has completed all arguments will have been popped from the stack and return values, if any, will have been pushed in place. As before, the first value pushed will be number one.

How to read the instruction set

The instructions are grouped together depending on the nature of the action they perform.

The description of each instruction has the following fields

- **Name**. The name of the instruction. This is not the same as the mnemonic for the instruction. The actual mnemonic is an abbreviated version of the name with options appended to indicate data type etc.
- **Description**. A description of the instruction and how it works.
- **Variants**. The variants of the instruction. The number in brackets indicates the variants op code.

- **Argument count**. The number of arguments popped from the stack by the instruction and the number pushed by the instruction.
- **Flags**. The flags used and/or affected by the instruction.
- **Size**. The number of bytes used to represent the instruction and constants if any.
- **Usage.** A demonstration of how to use the instruction. For instructions with several options only one variant is demonstrated.

The assembler is case insensitive and the mnemonics are only written with mixed case letters to increase readability (label names ARE however case sensitive).

Arithmetic

Add

Add the first argument to the second.
Carry may be ignored or included (indicated by the 'C' in the mnemonic) when performing the operation but it will always be effected by it.

Variants	AddS [], AddD [], AddQ [], AddF [], AddSC [], AddDC [], AddQC []
Arg. count	2 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushLD 102    ; arg 1 (the double at memory location 102)
PushCD 20000 ; arg 2 (a constant)
AddD          ; perform an addition
PopLD 102     ; place the result back at memory location 102
```

Subtract

Subtract the second argument from the first.
Carry may be ignored or included (indicated by the 'C' in the mnemonic) when performing the operation but it will always be effected by it.

Variants	SubS [], SubD [], SubQ [], SubF [], SubSC [], SubDC [], SubQC []
Arg. count	2 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushGQ 100 ; arg 1 (the quad at memory location 100)
PushLQ 104 ; arg 2 (the quad at memory location 104)
SubQ       ; sub 'em
PopLQ 108  ; put result at location 108
```

Multiply

Multiply the first argument with the second.
When multiplying integers the output will always be the double width of the input values.

Variants	MulS [], MulD [], MulQ [], MulF []
Arg. count	2 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushLQ 100 ; arg 1 (the quad at memory location 100)
PushLQ 104 ; arg 2 (the quad at memory location 104)
MulQ       ; Multiply them
PopLQ 108  ; put result at location 108
PopQ 112  ; ...and 112
```

Divide

Divide the first argument by the second.
When dividing integers the output will always be the double width of the input values (integer part pushed first followed by the fractional part).

Variants	DivS [],DivD [],DivQ [], DivF []
Arg. count	2 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushGQ 100 ; arg 1 (the quad at memory location 100)
PushGQ 104 ; arg 2 (the quad at memory location 104)
DivQ       ; Div ‘em
PopLQ 108  ; put result at location 108
PopLQ 112  ; ...and 112
```

Increment

Increment the value by one.

Variants	IncS [], IncD [], IncQ [], IncF []
Arg. count	1 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushLS 100 ; push the value (the single at memory location 100)
IncS       ; add one
PopLS 100  ; and pop it back
```

Decrement
Decrement the value by one.

Variants	DecS [], DecD [], DecQ [], DecF []
Arg. count	1 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushLQ 100 ; push the value (the quad at memory location 100)
DecQ       ; subtract one
PopLQ 100  ; and pop it back
```

Compare
Compare the two arguments by subtracting the second from the first. This instruction is basically identical to subtract but with the single difference that the result is not pushed onto the stack. The flags are still modified and so this instruction is suitable for comparing values.

Variants	CmpS [], CmpD [], CmpQ [], CmpF []
Arg. count	2 / 0
Flags	Z, C, O, N, S
Size	1

Usage

```
PushLQ 100 ; arg 1 (the quad at memory location 100)
PushLQ 104 ; arg 2 (the quad at memory location 104)
CmpQ       ; compare them
JmpNZ 120  ; jump if the arguments where unequal
```

Logical

And
Perform a bitwise and of the two values.

Variants	AndS [], AndD [], AndQ []
Arg. count	2 / 1
Flags	Z, O, N, S
Size	1

Usage

```
PushLS 100   ; arg 1 (the single at memory location 100)
PushCS 0x80  ; arg 2 (the constant 0x80)
AndS         ; do the and
PopLS 100    ; and back with it
```

Or
Perform a bitwise or of the two values.

Variants	OrS [], OrD [], OrQ []
Arg. count	2 / 1
Flags	Z, O, N, S
Size	1

Usage
```
PushLS 100   ; arg 1 (the single at memory location 100)
PushCS 0x80  ; arg 2 (the constant 0x80)
OrS          ; or ‘em
PopLS 100    ; and back with it
```

XOr
Perform a bitwise exclusive or of the two values.

Variants	XOrS [], XOrD [], XOrQ []
Arg. count	2 / 1
Flags	Z, O, N, S
Size	1

Usage
```
PushLS 100   ; arg 1 (the single at memory location 100)
PushCS 0x80  ; arg 2 (the constant 0x80)
XOrS         ; do the xor thing
PopLS 100    ; and back with it
```

Complement
Perform one’s complement of the value (i.e. flips all bits).

Variants	ComS [], ComD [], ComQ []
Arg. count	1 / 1
Flags	Z, O, N, S
Size	1

Usage
```
PushLD 100   ; arg 1 (the double at memory location 100)
ComD         ; complement it
PopLD 100    ; and put it back
```

Negate
Negate the value. Effectively subtracts the value from zero.

Variants	NegS [], NegD [], NegQ []

Arg. count	1 / 1
Flags	Z, O, N, S
Size	1

Usage

```
PushGD 100   ; arg 1 (the double at memory location 100)
NegD         ; negate the value
PopGD 100    ; and put it back
```

Shift

Shift the value one bit location left or right.
The bit being shifted in is always a 0. The bit being shifted out is placed in the carry flag.

Variants	ShfLS [], ShfLD [], ShfLQ [], ShfRS [], ShfRD [], ShfRQ []
Arg. count	1 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushLD 100   ; arg 1 (the double at memory location 100)
ShfLD        ; shift it left
PopLD 100    ; and put it back
```

Rotate

Rotate the value one bit location left or right.
The carry flag is rotated in and the bit being rotated out is placed in the carry flag after the rotation.

Variants	RotLS [], RotLD [], RotLQ [], RotRS [], RotRD [], RotRQ []
Arg. count	1 / 1
Flags	Z, C, O, N, S
Size	1

Usage

```
PushCS 0    ; arg 1 – IO pin 0
BiI         ; copy pin state to carry
PushCS 0    ; clear the value
RotLS       ; rotate it in as the LSB of value
```

Carry

Clears or sets the carry flag.

Variants	CarS [], CarC []
Arg. count	0 / 0
Flags	C

Size	1
Usage	`CarC     ; clear the carry flag`

Transfer

Load local
Load the constant into the local memory location.

Variants	LoadLS [], LoadLD [], LoadLQ [], LoadLF []
Arg. count	0 / 0
Flags	-
Size	4 / 5 / 7 / 7
Usage	`LoadLD 10 12345 ; put the double 12345 at local location 10`

Load global
Load the constant into the global memory location.

Variants	LoadGS [], LoadGD [], LoadGQ [], LoadGF []
Arg. count	0 / 0
Flags	-
Size	6 /7 / 9 / 9
Usage	`LoadGD 100 12345 ; put the double 12345 at global location 100`

Stack

Push
Push the value onto the stack.

Variants	PushS [], PushD [], PushQ [], PushF []
Arg. count	0 / 1
Flags	-
Size	2 / 3 / 5 / 5
Usage	`PushD 10000  ; place the value 10000 on the stack`

Push from local
Push the contents of the local memory location onto the stack.

Variants	PushLS [], PushLD [], PushLQ [], PushLF []

Arg. count	0 / 1
Flags	-
Size	3

Usage

```
PushLD 100  ; arg 1 (the double at local memory location 100)
ShfLD       ; shift it left
PopLD 100   ; and put it back
```

Push from global

Push the contents of the global memory location onto the stack.

Variants	PushGS [], PushGD [], PushGQ [], PushGF []
Arg. count	0 / 1
Flags	-
Size	5 / 5 / 5 / 5

Usage

```
PushGD 100  ; arg 1 (the double at global memory location 100)
ShfLD       ; shift it left
PopGD 100   ; and put it back
```

Push from constant

Push the contents of the constant memory location onto the stack.

Variants	PushCS [], PushCD [], PushCQ [], PushCF []
Arg. count	0 / 1
Flags	-
Size	5 / 5 / 5 / 5

Usage

```
PushCD 20   ; arg 1 (the double at location 20 in constant section)
ShfLD       ; shift it left
PopLD 100   ; and put it in local memory
```

Pop to local

Pop the top most value from the stack in to memory at the specified local memory location.

Variants	PopLS [], PopLD [], PopLQ [], PopLF []
Arg. count	1 / 0
Flags	-
Size	3 / 3 / 3 / 3

Usage	`PushLD 100   ; arg 1 (the double at local memory location 100)`
	`ShfLD        ; shift it left`
	`PopLD 100    ; and put it back`

Pop to global

Pop the top most value from the stack in to memory at the specified global memory location.

Variants	PopGS [], PopGD [], PopGQ [], PopGF []
Arg. count	1 / 0
Flags	-
Size	5 / 5 / 5 / 5

Usage	`PushGD 100   ; arg 1 (the double at global memory location 100)`
	`ShfLD        ; shift it left`
	`PopGD 100    ; and put it back`

Branches

Jump

Unconditional jump to the address specified by the constant. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	Jmp []
Arg. count	0 / 0
Flags	-
Size	3
Usage	`Jmp 100   ; jump to address 100`

Jump if zero

Jump to the address specified by the constant if the zero flag is set. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpZ []
Arg. count	0 / 0
Flags	Z
Size	3

Usage	`PushLD 100  ; arg 1 (the double at memory location 100)`
	`Dec`
	`PopLD 100`
	`JmpZ 230    ; jump to address 230 if the zero flag is set`

Jump if not zero
Jump to the address specified by the constant if the zero flag is cleared. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpNZ []
Arg. count	0 / 0
Flags	Z
Size	3

Usage

```
PushGD 100  ; arg 1 (the double at memory location 100)
Dec
PopGD 100
JmpNZ 325   ; jump to address 325 if the zero flag is cleared
```

Jump if carry
Jump to the address specified by the constant if the carry flag is set. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpC []
Arg. count	0 / 0
Flags	C
Size	3

Usage

```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpC 21     ; jump to address 21 if the carry flag is set
```

Jump if not carry
Jump to the address specified by the constant if the carry flag is cleared. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpNC []
Arg. count	0 / 0
Flags	C
Size	3

Usage

```
PushGD 100  ; arg 1 (the double at memory location 100)
Dec
PopGD 100
JmpNC 20    ; jump to address 20 if the carry flag is cleared
```

Jump if negative

Jump to the address specified by the constant if the negative flag is set. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpN []
Arg. count	0 / 0
Flags	N
Size	3

Usage

```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpN 232    ; jump to address 232 if the negative flag is set
```

Jump if not negative

Jump to the address specified by the constant if the negative flag is cleared. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpNN []
Arg. count	0 / 0
Flags	N
Size	3

Usage

```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpNN 5642  ; jump to address 5642 if the negative flag is cleared
```

Jump if positive

Jump to the address specified by the constant if both the zero and negative flags are cleared. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpP []
Arg. count	0 / 0
Flags	N
Size	3

Usage

```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpP 232    ; jump to address 232 if the negative flag is set
```

Jump if not positive

Jump to the address specified by the constant if either the zero or negative flag is cleared. The address is relative to the start of the function. Jumps are only possible within the function they occur in.

Variants	JmpNP []
Arg. count	0 / 0
Flags	N
Size	3

Usage

```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpNP 642  ; jump to address 642 if the negative flag is cleared
```

Call

Calls the function at the location specified. The address of the instruction immediately after the call is pushed to the stack.

Variants	Call []
Arg. count	0 / 0
Flags	-
Size	5

Usage

```
PushCD 100  ; argument for the function to call
Call 345    ; call function at address 345
...
PushGQ 10   ; origin 100
Dec
PopGQ 10
Ret         ; and return
```

Indirect call

Calls the function at the address pushed to the stack. The address of the instruction immediately after the call is pushed to the stack.

Variants	ICall []
Arg. count	1 / 0
Flags	-
Size	1

Usage

```
PushLQ 10   ; address of the function to call
ICall       ; call function
```

```
...
PushGQ 10   ; origin 100
Dec
PopGQ 10
Ret         ; and return
```

Return

Pops the return address from the stack and program execution continues from that address.

Variants	Ret []
Arg. count	0 / 0
Flags	-
Size	1

Usage

```
PushCD 100  ; push return value
Ret         ; return from function
```

IO

ByO

Transfer a value to an output port.

Variants	ByO []
Arg. count	2 / 0
Flags	-
Size	1

Usage

```
PushLS 100  ; the value to put on the port
PushS 110   ; the port number
ByO         ; output it
```

ByI

Read the value of an input port.

Variants	ByI []
Arg. count	1 / 0
Flags	-
Size	1

Usage

```
PushS 110  ; the port number
ByI        ; read it
```

BiO
Transfer the value of the carry flag to a single port bit.

Variants	BiO []
Arg. count	1 / 0
Flags	-
Size	1

Usage

```
PushS 10    ; the bit number
BiO         ; output carry to bit
```

BiI
Transfer the value of a single port bit to the carry flag.

Variants	BiI []
Arg. count	1 /
Flags	-
Size	1

Usage

```
PushS 10    ; the bit number
BiI         ; read bit value to the carry flag
```

Misc

Wait
Wait the specified number of milliseconds (a double) before proceeding to the next instruction.

Variants	Wait []
Arg. count	1 / 0
Flags	-
Size	1

Usage

```
PushCD 10   ; the number of milliseconds
Wait        ; wait
```

Threads

THREADING IS NOT SUPPORTED YET!

Spawn
Spawns a new thread and starts it.

Kill
Kills a thread.

Suspend
Suspends an existing thread.

Resume
Resumes a suspended or sleeping thread.

Sleep
Puts a task to sleep. If the task is already sleeping the longer of the two periods are used.

Building the VX core

Compile time settings
A number of settings can be adjusted before the VX VM firmware is compiled and programmed into the target controller. Using compile time configuration of the core firmware any program storage media can be used (internal EEPROM or flash, external flash card, USB port using a PC as virtual media to name a few).

The operating frequency of the processor must be configured allowing the firmware to adapt to any system speed. This affects the internal timers, the wait instruction, communication baud rates etc.

Threading model

THREADING IS NOT SUPPORTED YET!

Loading a program creates a main process. This process can create threads. These must be given an appropriate amount of stack space (taken from the processes stack space).

Threads have the same priority as the process they exist in. Single process environments inherently have no priorities.

To start a new thread the following arguments are required

Start address

Stack space

The VX tool chain

The VX platform is supported by a dedicated tool chain. All programs are made specifically for the VX VM.

The tools contained in the tool chain is

- **VXA.** The VX assembler. Generates .vxx executable files from .asm source files.

- **FSIC.** File Store Image Creator. Converts directories on a PC to image files (.vfi) for the file system used on the VX platform.

The VX assembler

The VX assembler (VXA) is a command line tool capable of generating VX executables.

Only one instance of section, label or mnemonic can appear in a line.

Sections

The assembler accepts four sections

- Code
- Const
- Stack
- Data

Labels

Labels bind a human readable textual name to a memory address.

Labels can be used in all sections.

Comments

Comments in the assembler source file is allowed by the assembler by removing any instances of ; (semicolon) and the rest of the line on which they were found.

Command line options

Options are passed to the assembler when executing it.

The following lists the options available

- **File.** –f. The file to assemble. This option must be included.
- **Output name.** –o. Using the –o option an alternative name can be given to the generated .part file.
- **Generate list file.** –l. If this option is used a list file is generated displaying details on the generated object file.
- **Generate preprocessor file.** –p. Generate a file containing the output from the preprocessor. The output has the extension .pre.
- **Generate map file.** –m. Generate map file when processing the .asm file.

Assembling a program

To assemble a file using the default options simply call the assembler with the –f option as shown below:

```
vxa –f hello.asm
```

This will, if the source file is valid, produce a file named hello.part.

The File Store Image Creator

The internal microcontroller EEPROM is used for a tiny file system. Due to the extremely small size a read-only file system is used. Since files then cannot be downloaded or created individually a binary image must be build on a PC and then downloaded. The files can then be read and loaded by the system.

VX files

The VX system operates with a number of files. These are all described below. Some might be mostly self explanatory and so there description will be short.

Executables

Virtual eXecuter program files with the extension .vxx (Virtual eXecuter Executable) contain the assembled program code, constants and any auxiliary program information that may be needed to load the program.

The file is divided into two sections

- Load info
- Binary data

Load info

This section contains a number of fields that are used by the program loader to determine how to load the program and if it is at all possible to load and run it.

- **VX file type tag.** A valid VX executable file starts with a tag made up of the five ASCII letters "VXEXE". The file extension is only to make it more human-friendly.
  This is the only field in the load section that is human readable.
- **VX core version.** This is the core version that the file was generated for. This is a single byte.
  If the version matches the actual core version that program loading will proceed. If the version does not match it is up to the core to decide whether it is able to execute the program or not. Backward compatibility is i.e. NOT guaranteed!
- **Code size.** This is the size of the program code that will be executed. This is a two byte integer.
- **Constant size.** This is the size of the constant section. This is a two byte integer.
- **Data size.** This is the size of the data section. This is a two byte integer.
- **Stack size.** This is the size of the stack section. This is a two byte integer.

The load info section takes up exactly 14 bytes.

If the tag is not correctly identified or if the executable version is not supported or if the total memory requirement exceeds the amount of free RAM the program is not loaded and an error message will be produced on the default communication channel.

Binary data

The binary data section contains an image of what must be copied to memory i.e. the code and const sections. This section will be copied directly to the memory.

Assembler source files

These are the source code of programs.

Assembler source files have the extension .asm.

List files

A list file may be generated when assembling a source file to inspect the generated binary code.

List files have the extension .lst.

Map files

A map file may be generated when assembling a source file to inspect the mapping of labels in memory.

Map files have the extension .map.

Preprocessor files

The output of the preprocessor (first step in the assembling processor) may be output to a file.

Preprocessor files have the extension .pre.

Terminal interface

The main interface to a system running the VX VM is through the terminal interface. This terminal provides a simple interface like that of DOS or similar. It enables a user to list and execute files and to upload new disc images. As only the read-only version of the file system is implemented modifications or upload of individual files is not possible.

The file system

VX at first only support the VX File Store file system. This simple but ultra compact file system is read-only which in turn requires a disc image to be prepared on a PC using the VX File Store Image Creator (FSIC) application and then downloaded to the storage media. This download procedure can be done either trough a dedicated microcontroller programming tool or through the terminal interface.

Discs are mapped to single letters (again think DOS). The letter ‘A’ is reserved for internal microcontroller EEPROM (even if no EEPROM is present). Additional drives will be assigned letters in a yet unspecified manner but will depend on which disc interfaces is installed in the core.

What’s next

Compressed executables

In the future compressed executables may be implemented. These should have the VX file type tag “VXCEX” and will be decompressed when loaded. The three size values in the load info section will represent the in-memory size requirements.

Enable programs to be made up of more than one source file

Currently the assembler only accepts one source file. This is obviously a short coming and should be extended.

It would be desirable if the –s source file option could be included in the assembler files. This would enable assembler files to automatically include the source files they them self require.

FAT16 support

Support for the   FAT16 file system must be added ASAP to enable SD memory cards written by a PC to be used as discs in a VX system.

Terminal command summary

cd	Change directory. Changes to the specified path or emits an error if path is unavailable. As in DOS '..' denotes the parent directory.
<disc>:	Changes to the disc with the letter specified or emits an error if disc is not available.
list	Lists files and directories in current directory.
print <file>	Prints the contents of the file specified or an error message if the file was unavailable.
<file>	Loads and executes the specified if the fail is available and if the file is a valid VX executable file. Otherwise an error message is printed.
load <disc> <size>	Loads a disc image to the disc specified through the terminal interface. The terminal interface channel is assumed to be reliable and so the image must be sent "as is" with no encoding. The size field is used to determine when the entire image has been transferred and to verify that the image fits on the disc. To abort transfer after the command has been executed (but before the actual file transfer has commenced) press the ESC key.