Table of Contents

Notices

IBM and z/Architecture are registered trademarks of International Business Machines Corporation.

Introduction

After many years exploring various approaches to Forth language implementations and dialects, this document discusses some approaches and their various attributes. The objective is to find an approach providing feasible implementation in one or more mainframe architectures as stand-alone or bare-metal programs, an area of special interest to the author. Charles H. Moore started development of the concepts embodied by Forth in 1958. While mainframes did not enter the computing picture until the mid 1960's, Forth did reach them in the early 1970's when Mr. Moore moved Forth to an IBM® 360/50. This effort essentially re-targets mainframes as a Forth option.

Why target mainframes in the 21st century? Forth, as all high-level languages, has always been a productivity tool. The constraints of the early days of computing drove its inception. As its history illustrates, Forth has always found a desirable home in resource or functionality constrained environments. Look at its history: early computers, minicomputers, microcomputers, and now micro-controllers. But, are not mainframe systems massive? Yes.

But mainframes can now be emulated on single user systems such as PC's or even smaller such as a RaspberryPi. And while even very modern systems, such as Linux can be supported by emulation, for stand-alone or bare-metal programs, the programmer is back to the limited set of resources offered by the hardware, or its emulation. Re-enter Forth.

For information about Forth and its historical development refer to

- http://en.wikipedia.org/wiki/Forth_%28programming_language%29
- http://www.colorforth.com/HOPL.html
- This link, provided by Brad Rodriguez, has some details and helpful diagrams of the operation of classical Forth systems and their design considerations:

  http://www.bradrodriguez.com/papers/moving1.htm
- More generally Brad Rodriquez master list of papers has useful information:

  http://www.bradrodriguez.com/papers/mindex.html

Four approaches have been examined, with the most functionality being first and least robust being last:

- IEEE 1275, Open Firmware, a set of standards,
- *American National Standards for Information Systems – Programming Languages – Forth*, X3.215-1994, a standard,
- Tiny Open Firmware, developed by Brad Eckerd, available at `ftp://ftp.taygeta.com/pub/Forth/TinyBoot`,
- F- (F Minus), also by Brad Eckerd, available at `fminus.sourceforge.net`.

The first two approaches are standards so, if implemented, would define a level of delivered functionality. Implementations of ANS compatible Forth systems exist that could be used to guide an implementation. The second two are actual implementations and are instructive because they both target bare-metal environments as does the implementation proposed in this document.

Standards vs. Implementations

Forth language Implementations predated Forth language standards by decades. Naturally, once formal language standards emerged, implementations compatible with the standards started to occur and have become the norm.

ANS Forth explicitly states that its purpose is to promote Forth program portability by specifying an interface between a Forth system and a Forth program by defining the words provided by a Standard System. The interface includes:

- the forms that a program written in the Forth language may take;
- the rules for interpreting the meaning of a program and its data.

The ANS Forth standard then lists seven things not specified by it. All of them amounting to implementation aspects of a Forth System.

IEEE 1275 is a Forth language based firmware specification. It similarly states that it is “bus, vendor, operating system (OS), and instruction-set-architecture (ISA)-independent”. Just another way of saying it does not specify anything about implementation of the firmware than does the ANS Forth standard does for a Forth System.

While both of these standards provide a set of terms and concepts related to their respective purpose and scope, they do nothing for any discussion about implementation. The over three decades of Forth language implementations preceding such standards, however, has developed a set of concepts and terminology related to Forth language implementations. These are introduced in the “Terminology” section and expanded in the “Forth Virtual Machine Architecture” section.

Cross-Compiling

ANS Forth identifies two types of compliance levels:

- a system, or
- a program.

The standard differentiates a “system” from a “program”. A program is defined in the 1994 standard as:

> A complete specification of execution to achieve a specific function (application task) expressed in Forth source code form.

A system on the other hand makes available, as a minimum, a set of core words for program usage, accessed from a dictionary, and a text interpreter. Presumably, the source program is interpreted by the text interpreter making it available for use in the same environment.

ANS Forth defines a cross-compiler as

> A system that compiles a program for later execution in an environment that may be physically and logically different from the compiling environment. In a cross-compiler, the term “host” applies to the compiling environment, and the term “target” applies to the run-time environment.

No further information is provided on the topic of cross-compiling in the standard. Elizabeth Rather of Forth, Inc., authored a proposed set of word extensions for cross-compilers

> http://www.mpeforth.com/arena/XCtext5.PDF

In particular, the host compiling environment is expected to provide a Forth system while the program produced by such a system purely performs its specific function in the target environment.

Ultimately this document is foundational for a cross-compiler targeting bare-metal program operation on the IBM mainframe family or compatible systems.

Threaded Interpretive Languages

This document assumes a degree of familiarity with threaded interpretive language (TIL) concepts and to some degree the Forth language in general. For more general introduction to such languages (of which FORTH is really an example), refer to the above links and additional links found at those sites. This section introduces terms used within this document within this context.

Threaded interpretive languages execute a series of predefined and extensible functionality. Each distinct function is identified by a set of characters referred to as a “word.” These words become the human representation of the functional nugget. Such words are used by humans that create a Forth program or interact with a running Forth system. Each word's “meaning” is defined by how it interacts with a set of LIFO stacks and any accompanying side effects. By allowing new words to be defined in terms of existing words, the language is extended.

Ultimately instructions in the underlying platform must be executed. Various terms have been used to describe such Forth words, for example, a primitive word or a code word. The latter is derived from the practice of using a Forth word with the letters “C”, “O”, “D” and “E” in it when defining a word composed of machine instructions. This document will use the term “code word” for Forth words defined by machine instructions.

A word composed of other words, not just machine instructions, is usually referred to as a “colon” word (because the “:” character usually initiates a definition of these words). A thread, in this context, is a series of sequentially executed words. Various Forth words provide flow control within the word itself. The manner in which the **implementation** strings a series of words together to execute the thread is referred to as its threading model.

Words may be interpreted in multiple ways. At the hardware machine level resides the “inner interpreter.” Frequently it is referenced as the Forth virtual machine. In this regard, Java is similar to Forth. Java too has an internal virtual machine. The Forth virtual machine implements the threading model that executes Forth colon word sequences.

At the level of a human user, Forth words may be interpreted based upon their character based names, each word separated by white space, typically a blank character. This level of interpretation can be referred to as the “text interpreter”. Usually the text interpreter is itself a set of Forth words.

The text interpreter usually operates in one of two modes. In both cases a user is supplying input words but how the input is interpreted is different in the two states. In interpretation state, each word is directly executed as it is recognized in the input. In the second state, compilation state, the words are not executed but used to define new words. Forth systems always have a mechanism for storing programs to allow them to be added to the system by a user without having to completely retype them. Over time these facilities have varied. Ultimately they always depend upon words designed for this purpose.

Major Features

This section describes a number of features of each approach. It first describes each feature and will then provide a comparison.

Standards

Forth is based upon the concept of words as described above. Each of the various standards that has emerged over time, attempts to standardize the words and their meanings. As the standards evolved, related words have been grouped into word sets. The ANSI 1994 standard has 254 core words. Because of the historical variability and diverse uses of Forth, this standard allows for a subset of core words to be provided by a system, but requires it to be labeled as “ANS Forth System (Subset)”. An ANS Forth system must provide an outer-interpreter, referred to as the “text interpreter” by the standard.

The IEEE 1275 Open Firmware standard has a number of optional word sets, however, it requires three specific interfaces to be supplied for compliance, each of which capitalizes on a set of required and optional words. These interfaces are the device interface, the user interface (or outer-interpreter) and a client program interface providing an interface to the firmware from a booted program.

The device interface of IEEE 1275 introduces an interesting intermediary form of Forth program, called FCode. FCode is an intermediate form of compiled Forth words. In the standard, specific Forth words are represented by a numerical value. The values are fixed and the FCode representation is created external to the Open Firmware implementation by means of a “tokenizer”. FCode is a specific case of an execution token known as byte code. The sequence of compiled tokens is then analyzed by a token analyzer to create and define words within the Open Firmware implementation available to all three of the interfaces. Additionally the device interface provides a form of object-oriented programming. Each “package” is essentially a class definition and when the package is “opened” it creates in essence a separate class instance on which methods may operate.

Implementations

Both of the implementations above, Tiny Open Firmware (TOF) and F Minus (F-), are influenced by the IEEE 1275 standard. This is particularly true in their use of a byte code representation of Forth programs. Both implementations have a few things in common. They both require a Forth system for use. They both create token byte code. They both can support multitasking. Unlike the IEEE 1275 standard, the token assignments are dynamic based upon the words created and used during the target system build.

They differ in how the byte code is handled. TOF will convert the byte code into executable machine instructions for use on the target hardware. TOF requires a token evaluator to be resident on the target to perform the conversion.

The F Minus system eliminates the conversion process entirely and all that resides on the target system is the token evaluator. The evaluator executes the token byte code directly.

The evaluator itself is generated by the F Minus system on a separate host as either a C or assembler source file. This source file is then turned into executable code using the corresponding target tool.

In many respects F Minus is a slimmed down TOF. This is consistent with the different objectives of the two systems. TOF creates a firmware package for small embedded systems. F Minus targets programming of micro controllers. The latter is optimized for size. TOF is typically around 20K in size. F Minus can usually fit in a ROM of between 4 and 8K in size.

Standard Structural Model

The 1994 American National Standards Institute standard X3.215, *Programming Languages – Forth* offers a number of useful definitions. First it defines the standard's scope in section 1.2 as:

> This standard specifies an interface between a Forth system and a Forth Program by defining the words provided by a Standard System.

The standard does not address how a Forth system might be implemented for the execution of the defined words. Nevertheless, in the course of defining the standard certain terms are used whose definitions do expose certain relationships between the terms.

word: Depending upon context, either 1) the name of a Forth definition; or 2) a parsed sequence non-space characters, which could be the name of a Forth definition.

definition: A Forth execution procedure compiled into the dictionary.

compile: To transform into dictionary definitions.

dictionary: An extensible structure that contains definitions and associated data space.

data space: The logical area of the dictionary that can be accessed.

data field: The data space that is associated with a word that is defined via **`CREATE`**.

code space: The logical area of the dictionary in which word semantics are implemented.

execution token: A value that identifies the execution semantics of a definition.

name space: The logical area of the dictionary in which definition names are stored.

execution semantics: The behavior of a Forth definition when it is executed.

interpretation semantics: The behavior of a Forth definition when it is encountered by the text interpreter in interpretation state.

compilation semantics: The behavior of a Forth definition when it is encountered by the text interpreter in compilation state.

The generic term “execution semantics” is broken down into three categories in the standard's discussion of Rationale for its descriptions.

1. Initiation semantics – the code that is executed upon entering a definition associated with the **`:`** word that initiates the definition.
2. Termination semantics – the code that is executed upon exiting a definition associated with the word **`EXIT`** or **`;`** used to conclude the definition.
3. Run-time semantics – code fragments, such as literals or branches that are compiled into colon definitions by words with explicit compilation semantics.

These three categories are in addition to the semantics specified by the use of other words compiled as execution tokens into a word's definition.

No mention is made of the Forth virtual machine. The Forth virtual machine and the actual structure of the dictionary are implementation defined and not part of the ANS Forth standard's scope.

Cells

ANS Forth utilizes the concept of a cell. The cell is “the primary unit of information in the architecture of a Forth system.” Section 3.1.3 has this to say about cells:

> Cells shall be at least one address unit wide and contain at least sixteen bits. The size of a cell shall be an integral multiple of the size of a character. Data stack elements, return stack elements, addresses, execution tokens, flags, and integers are one cell wide.

Stacks

The Forth stacks, data and return, are described in terms of cells. The ANS Forth standard, per Brad Rodriguez, requires the data stack to be a minimum of 32 cells and the return stack to be a minimum of 24 cells. Mr. Rodriguez recommends 64 cells for each.

Terminology

The previous section introduced a set of terms based upon the ANS Forth standard. Classical Forth systems used a different set of terms. No direct “translation” appears to exist between the two sets of terms. This section attempts to provide some clarity between the sets, at least from the perspective of terms meaningful in this document. It is hoped that this section will aid anyone referencing links encountered here or other documentation from the era predating the ANSI standard of 1994.

Outside of differences in word semantics, terms used by classical Forth system discussions are derived from a set of implementation practices commonly found in these systems. The operative word here is **implementation**. The ANS Forth standard specifically avoids implementation discussions. The ANS Forth standard developed a new set of terms that are intended to be implementation neutral.

In both frameworks, the dictionary is the foundation of the system. However, in classical Forth systems the dictionary is usually discussed in terms of its common implementation components. Each entry in the dictionary in classical Forth systems is composed of four areas:

- **Header** of internal data, typically, but, not exclusively, the header links entries; a
- **Word** character string, sometimes considered part of the header; a
- **Code Field Address (CFA)**, the address of machine instructions associated with the word, used as the word's execution token; and a
- **Parameter Field (PF)** defining the word's semantics, may contain compiled references to colon words or machine instructions in the case of code words.

The classical Forth dictionary entry can be visualized this way for a colon definition. Each “XTn” is an execution token. The CFA is highlighted in orange. The Parameter Field is highlighted in blue. The Header and Word string are highlighted in gray. For a program lacking a dictionary, these two fields are not present.

Header	Word (variable)	CFA	PARAMETER FIELD						
			XT1	XT2	XT3	XT4	XT5	XT6	XT7

The dictionary entry for a code word is very similar. The execution tokens are replaced by machine instructions. The only change is to highlight the machine instructions in green.

Header	Word (variable)	CFA	PARAMETER FIELD						
			MI1	MI2	MI3	MI4	MI5	MI6	MI7

The blue and green highlighting will continue to be used when describing the layout of a definition for different threading models.

ANS Forth does not discuss the Header at all. It is implementation specific and outside its scope. The word string in ANS Forth is placed in the name space, an undefined logical part of the dictionary. In both cases the CFA and parameter field are placed within the ANS Forth code space, where execution semantics are defined.

In a cross-compiled environment, the host system maintains the dictionary information for the cross-compiled target program. Only those portions of the dictionary upon which the cross-compiled target program depends are required within the program. The threading model discussed below dictates what those elements are.

How ANS Forth treats the Code Field Address requires a little explanation around the use of the Code Field Address in classical Forth systems. It was mostly an address of some machine instructions. For code words the machine instructions would execute the word's execution semantics. In colon words the machine instructions would start execution of the word's compiled word sequence, or as ANS Forth calls it, the initiation semantics. In classical Forth systems this is usually referred to by the name **ENTER**. A word's Code Field Address usually played a role when the word was compiled into another colon word's definition. In some cases the colon word using the word would have the address of that word's Code Field Address compiled into its definition. In some cases, the contents of the word being compiled was coded into the definition. In classical Forth systems then, the Code Field Address within a dictionary entry served two roles:

1. located the starting location of the execution semantics of the word and
2. was used directly or indirectly in a colon word's definition when the word is part of another word's definition.

ANSI Forth separated these concepts in its terms. Whatever gets compiled into a colon word's definition that identifies a word's execution semantics is termed its execution token. Under this definition, this term for the second usage includes the classical function of the CFA without constraining it to an actual memory address. ANS Forth places only one constraint on an execution token:

> The association between an execution token and a definition is static. Once made, it does not change with changes in search order or anything else. However it may not be unique ... . - see A.3.1.3.5

The most common usage of the CFA is to point to a colon word's entry logic. While not an official term within the standard, this first usage of CFA is described in ANS Forth as the word's initiation semantics and is considered part of the word's definition.

The Parameter Field within a classical Forth's dictionary entry is where the word's semantics were defined, as separate from the CFA. This is true of both colon and code words. In ANSI Forth a word's semantics is placed within the logical portion of the dictionary called either the data space or code space and an individual word's semantics when compiled is part of the using word's data field. For ANSI Forth, whatever went into the word's CFA is part of the word's semantics and hence would be considered in ANS Forth as being in the word's data field.

Anything implemented in machine instructions is outside of the scope of the ANS Forth

standard. It nevertheless recognizes the need for machine instructions in the implementation and lumps all of it in what it terms the code space, another logical part of the dictionary. The standard states nothing about what is contained in the code space. To emphasize that, the Rationale section of the standard for code space, A.3.3.2, is completely empty. It does not even contain a statement such as “out of scope”. Nothing at all is stated about the code space.

Terms used in classical Forth systems relating to machine instructions, for example,

- NEXT, or the inner interpreter, per Brad Rodriguez,
- ENTER,
- EXIT,
- Forth Virtual Machine, or
- threading model

are all part of the undefined code space in ANS Forth. The remainder of this document discusses the code space. Next, some general architectural considerations are discussed. Following that is a discussion of implementation considerations for a mainframe system and details for such an implementation.

The details are still being developed and will be elucidated in this document when available.

Forth Virtual Machine Architecture

This document takes the view that the Forth virtual machine encompasses all of what the ANS Forth standard leaves to the code space. As with any hardware machine, the Forth Virtual Machine consists of two components: registers and instructions. Both are abstractions implemented in hardware specific software. The registers perform specific roles in the machine. In the context of the virtual machine, the instructions are the system's execution tokens in whatever form they take in the implementation. The base instruction set are execution tokens of code words. Colon word execution tokens can be thought of as local extensions to the Forth virtual machine base instruction set, its collection of code words. The threading model provides the interface between the virtual machine's architectural registers and its instruction set of execution tokens.

Required Code Words

The minimum number of code words required for a Forth implementation is a topic of frequent discussion. The answer is really, it depends. The virtual machine must be written in machine instructions. The boundary between code words and the virtual machine tends to blur. They cooperate together. As few as nine have been identified in one artificial case. An early Forth implementation, eForth, was explicitly developed with 35 code words, excluding platform specific words (DOS, in this case). eForth was specifically designed for portability, so its choice could be viewed as a practical lower limit. This is why its code words are included in the following table.

F Minus uses a similar set of primitives, although larger (57), tailored in one case for the PIC18F452 micro-controller. F Minus introduces two non-standard virtual machine architectural elements, the A register and the user task pointer (for multitasking). A number of words in F Minus utilize these new VM components. F Minus reserves five opcode values or byte code execution tokens (0x26, 0x29, 0x39 0x3E and 0x3F) and has two for expansion (0x0A and 0x0B). A total of 64 byte code values are available for standard usage in F Minus.

eForth	F- Word	F- Opcode	ANSI Forth	Open Firmware FCode
EXIT	EXIT	0x01	6.1.1380	0x33
EXECUTE	TEXEC	0x2F	6.1.1370	0x1D
_LIT	~LIT	0x38		0x10
_ELSE	~BRAN	0x1A		0x13
_IF	See ~GBRAN and ~MBRAN			0x14
C!			6.1.0850	0x75
C@			6.1.0870	0x71

eForth	F- Word	F- Opcode	ANSI Forth	Open Firmware FCode
!			6.1.0010	0x72
@			6.1.0650	0x6D
RP@	RP@	0x3B		
RP!	RP!	0x0D		
>R	>R	0x05	6.1.0580	0x30
R@	R@	0x02	6.1.2070	0x32
R>	R>	0x03	6.1.2060	0x31
SP@	SP@	0x3A		
SP!	SP!	0x0C		
DROP	DROP	0x0F	6.1.1260	0x46
SWAP			6.1.2260	0x49
DUP	DUP	0x3C	6.1.1290	0x47
OVER	OVER	0x07	6.1.1990	0x48
CHAR-				
CHAR+			6.1.0897	0x62
CHARS			6.1.0898	0x66
CELL-	CELL-	0x13		
CELL+	CELL+	0x12	6.1.0880	0x65
CELLS			6.1.0890	0x69
0<	0<	0x17	6.1.0250	0x36
AND	AND	0x2C	6.1.0720	0x23
OR	OR	0x2E	6.1.1980	0x24
XOR	XOR	0x2D	6.1.2490	0x25
UM+				
BYE			15.6.2.0830	
NOOP	NOOP	0x00		0x7B
	~CALL	0x04		
	A@	0x06		
	BREAK	0x08		
	~OPCODE	0x09		

eForth	F- Word	F- Opcode	ANSI Forth	Open Firmware FCode
	UP!	0x0E		
	1+	0x10		
	1-	0x11		
	REL>ABS	0x14		
	ABS>REL	0x15		
0=	0=	0x16	6.1.0270	0x34
See _IF	~GBRAN	0x18		
See _IF	~MBRAN	0x19		
INVERT	INVERT	0x1B	6.1.1720	0x26
	2*C	0x1C		
	2*	0x1D		
	U2/	0x1E		
	2/	0x1F		
	C!A	0x20		
	C!A+	0x21		
	W!A	0x22		
	W!A+	0x23		
	!A	0x24		
	!A+	0x25		
	A!	0x27		
	~0BRAN	0x28		
+	+	0x2A	6.1.0120	0x1E
	+C	0x2B		
	C@A	0x30		
	C@A+	0x31		
	W@A	0x32		
	W@A+	0x33		
	@A	0x34		
	@A+	0x35		
	C@AC	0x36		

eForth	F- Word	F- Opcode	ANSI Forth	Open Firmware FCode
	`@AC`	0x37		
	`STATUS`	0x3D		

Registers

The classical Forth VM implements a number of registers. Ideally these VM registers use a hardware register for its implementation. Where that is not possible, the register must be implemented in random access memory (RAM).

Some Forth VM's extend the set of normal registers with new ones. The underlying hardware may make this desirable or even necessary. An example of an extension is the use of a *A* register in the F Minus system. This register is a generic register allowing data to be passed between separate words without direct interaction with a stack. A number of F Minus words provide access to and utilize this register in various contexts. Most of the words that utilize the *A* register in F Minus do so as if it were an address register.

Another common extension is the User Pointer (*UP*) register used for multitasking. See the related section below for more information concerning the *UP* register.

Each register's role is discussed in more detail in its respective section.

Brad Rodriguez in the above link provides an excellent discussion of Forth VM registers. That information is summarized in the following table. Recommendations are made for use of hardware registers. For performance reasons, the availability of hardware registers for a Forth VM register is more critical for some than others. The number in the “Priority” column indicates the priority for hardware register assignment when registers are limited, 1 being the highest priority. The three columns “Memory Addressing”, “Arithmetic Calculations”, and “Hardware Branch” identify the hardware capabilities required of the register when assigning it for the role of the respective Forth VM register.

Register	Required	Memory Addressing	Arithmetic Calculations	Hardware Branch	Priority	Description
W	yes	yes	yes	Note 1	1	Working register.
IP	yes	yes	yes	no	2	Interpreter Pointer. Note 2
SP	yes	yes	yes	no	2	Stack Pointer for data stack
RP	yes	yes	yes	no	2	Return Pointer for colon word return stack.
X	no	yes	yes	Note 3	3	Working register.
TOS	no	yes	yes	no	4	Top of data stack cell
UP	no	yes	no	no	5	User Pointer for multitasking. Note 4

Note 1: Used by direct thread model.

Note 2: Not used by the subroutine thread model.

Note 3: Used by indirect thread model.

Note 4: Not a classical Forth register.

IP – Instruction Pointer

The Forth VM has an instruction pointer (*IP*) register that keeps track of where within a colon word the machine is executing and is used to identify the next word of the definition to be executed. In ANS Forth terminology, the *IP* register points to the memory location of the next execution token to be executed.

The actual memory contents and how the contents relate to colon word execution is discussed in the “Threading Model” section.

Code words of course use the hardware equivalent that points to the next hardware machine instruction.

SP and RP – Data and Return Stack Pointers

The Forth VM uses two stacks resident in RAM, a data stack and a return stack. Both stacks are last-in/first-out stacks (LIFO). The data stack contains arguments on which a word operates, adding and removing values as the word definition dictates. The return stack is primarily used for managing the flow through colon words, maintaining the location to which a word must return.

Each of the two stacks have a Forth VM register pointing to the stack's logical top cell: the

- *SP* (stack pointer for the data stack), and the
- *RP* (return pointer for the return stack).

Whether a stack grows to higher or lower memory addresses is an implementation detail.

The SP and RP are universal. However, when words are being compiled, an additional stack is used, the control flow stack. It too is a LIFO stack. It is used to maintain flow control through sets of words. This stack is used when compiling various conditional execution words, usually involving multiple such words. Ultimately these result in colon words that effect the execution stream by acting on the *IP* register. In the discussion of required code words, these usually result in one of the branch related words being compiled into a colon word's definition. The ANS Forth standard informally refers to these words as containing run-time semantics.

TOS – Top of Data Stack

When available, an optional register can be the *TOS*, or Top of Stack, register. The *TOS* register maintains the content of the top data stack cell. Having this in a register saves a lot of memory accesses because frequently words will manipulate this cell but not need others.

Because the *TOS* register is a logical extension of the data stack, having the *TOS* available influences how data stack cells are popped from and pushed to the data stack. Popping a

cell from a totally memory resident stack means simply adjusting the *SP* register to point to the next stack cell. But with the *TOS* register, the contents of the cell pointed to by the *SP* register must be placed in the *TOS* before the *SP* register is adjusted. Similarly, pushing a cell onto the stack means that the *SP* must be adjusted to the next cell, the *TOS* register contents is then placed in this cell, and finally the value being "pushed" onto the stack is placed into the *TOS* register.

W – Work Register

A *W* register is usually part of the virtual machine. This is an extra register used for a number of functions, a "work" register.

UP – User Pointer

The User Pointer (*UP*) register is an extension of the classical Forth VM.

The ANSI Forth Standard does not discuss the concept of multitasking. Multitasking usually takes the form of cooperative multitasking, but can support preemptive multitasking as in the case of CAMEL Forth. Multitasking as typically implemented involves two VM concepts:

1. a task specific area (TSA) and
2. the *UP* register itself.

The task specific area

- maintains the state of a task while it has released control of the processor or Forth VM to other tasks, and
- provides the information for initializing the Forth VM when starting the task.

When preemptive multitasking is available, the task specific area will maintain hardware registers in addition to the Forth VM register state while the task has been preemptively interrupted. The *UP* register simply points to the active task specific area. Words are supplied to support creation of a task and relinquishing control. It may make sense to keep this register in a memory location.

See the "Multitasking" section for a broader discussion of how multitasking influences the Forth system as a whole.

Execution Semantics

The ANSI Forth standard says this in section 3.4.3:

> Execution may occur implicitly when the definition into which [an execution token] has been compiled is executed or explicitly ... .

The implicit execution is controlled by the *IP* register within the Forth VM. It points to the next execution token to be executed.

Three stages are used in the execution of an execution token: NEXT, ENTER and EXIT.

Code words only use the NEXT stage. Colon words use all three. Each stage is implemented in the Forth VM by machine instructions. So, each resides in the ANS Forth code space.

NEXT Execution Stage

The start of the execution semantics of each word is the NEXT function of the Forth VM having the responsibility to:

1. access the next execution token to be executed (as identified by the *IP* register),
2. update the *IP* register to point to the next sequential execution token in the current word,
3. locate the word's definition in either the data space (for colon words) or the code space (for code words), and
4. pass control of the hardware to the word's execution semantics.

The NEXT execution stage is the Forth “inner interpreter”. It operates outside of any word definition.

For code words, control is passed to the word's definition contained in code space, its machine instructions. Code words simply return control to the beginning of the NEXT execution stage by passing hardware control back to it, that is, the code word branches back to the machine instructions that implement the NEXT execution stage, causing interpretation of the next colon or code word.

For colon words, the Forth VM passes control to the ENTER execution stage.

ENTER Execution Stage

The ENTER execution stage provides entry into a colon word's definition. It performs the colon word's initialization semantics. This involves saving the location of the next execution token of the word previously being executed on the return stack and updating the *IP* with the location of the first execution token in the new word's data field.

The initiation semantics completes when it returns to the Forth VM NEXT execution stage starting the implicit execution of the new word's execution tokens. When a word's execution is complete, it must pass control of the Forth VM back to the colon word that called it. This is done by entering the EXIT stage.

EXIT Execution Stage

Because the Forth VM is implicitly executing compiled execution tokens when a colon word is complete, the EXIT execution stage is entered by the Forth VM executing the execution token of the EXIT word definition. As an execution token of a colon word, the EXIT execution stage is associated with the word that signals the end of a colon definition. This word is a single character, the semi-colon, or `;`, This word is a code word. It acts directly on the Forth VM registers.

The word's role is to reverse the actions taken by the ENTER execution stage, thereby allowing the Forth VM to return to the original colon word that caused the current word to be executed. It removes the top item on the return stack (the previously saved *IP* register contents) and restores it to the *IP* register.

As with each code word, it completes by returning control to the “inner interpreter”, the NEXT stage, by branching back to it. Implicit execution of execution tokens then resumes in the original word.

Run-time Semantics

Some words will modify the *IP* register during their implied execution within a colon's data field. This usually involves some form of flow-control that alters the implied sequential execution of the word's definition or because some data field content needs to be ignored by the Forth VM. Data that is ignored are typically some character or numeric value compiled within the definition. Think of these data being constants. Because the Forth VM *IP* register is only accessible by machine instructions, these words always involve code words.

There is nothing precluding a colon word's execution token from using data placed within its definition as is done with the some code words. The manipulation of the *IP* must be done in this case via manipulation of the saved *IP* register on the top of the return stack.

In either case, some information is placed within the colon word's definition that is not an execution token. This requires that the word compilation semantics is specific to the word that places this information into the colon definition. If it is a colon word that is placing the information into the definition it must support its own specific compilation semantics.

Colon words can do the equivalent by means of `IMMEDIATE` words that build the execution token followed by the related in-line content.

The `CREATE` *build parameter field* `DOES>` sequence allows arbitrary data to be built into a word's parameter field with an arbitrary set of execution semantics. The sequence is usually placed inside a word that uses this sequence.

Both of these cases are implemented by interaction with the text interpreter and are outside the scope of this document.

Threading Models

How the Forth VM processes colon words is referred to as its threading model. The threading

model really defines how each series of words defining a colon word is encoded into the word's definition as seen by the Forth VM itself. Various approaches have evolved. The selection of the threading model is influenced by both size and speed considerations. The size of the encoding drives the amount of storage required by the system. The encoding method drives performance. Some encoding schemes are more efficient to execute. Ultimately the nature of the hardware may dictate which encoding method is more efficient in its environment. Regardless of the encoding, it will always result in a processor memory address that contains a hardware instruction to which control of the hardware processor is passed when the address becomes the next hardware instruction address, in simple terms, a hardware branch address.

Four threading models have emerged.

- Indirect threading,
- Direct threading,
- Subroutine threading, and
- Token threading

Processor and other environment considerations usually drive the selection.

The information provided by Brad Rodriguez on porting Forth includes a good discussion of the various thread models encountered in classical Forth systems. Some of the models discussed at the above link are related to specific hardware and are not mentioned below.

Execution tokens are compiled into ANS Forth colon words. This description of a cell requires, for ANS Forth compatibility, that the execution token be the size of a cell. In practice this is usually either two-, four- or eight-bytes. The cell size of a given system is one of the factors in determining which model might be most attractive relative to size.

Each threading model influences:

- the design and placement of the Forth VM three execution stages,
- code word definition content,
- colon word definition content, and
- a cross-compiled program's content.

Each of these aspects are identified for each threading model. The descriptions are all based upon the classical Forth dictionary entry content and how that content changes for each model. This makes the evolution very clear. Values are placed on and removed from the Return Stack. For purpose of the following descriptions, assume these are assembler macros.

Indirect Threading

In this model a colon word contains an address that points to the definition of the compiled word. The contents at this address is the address of the machine instructions that execute

the word. So two address fetches and a branch are required for each word executed by the inner interpreter within a colon word:

1. Colon usage: an address to the word's definition, for example, its dictionary entry's CFA.
2. Word definition: an address to the word's machine instructions, the dictionary entry's CFA itself.
3. Branch to the word's machine instructions.

For code words, the CFA dictionary entry will point to the same dictionary entry's PF, where the word's machine instructions reside.

For colon words, the CFA will point to the NEXT execution stage machine instructions.

The Forth VM in the indirect threading model incorporates the NEXT and ENTER. EXIT is entered via its own code word definition when invoked by the a colon word definition. EXIT may use a portion of the Forth VM to perform the function or perform it entirely within its definition. In this later case, it will pass control to NEXT when it has completed the exit stage processing.

The compiling of the execution token as a dictionary entry's CFA address was a very simple approach for a text interpreter also written in Forth. When a word was encountered it was looked up in its dictionary and the dictionary and the address of the CFA could be readily determined and compiled into the colon definition. It allowed all words to have the same construction and use the same execution stages.

In a cross-complied program, the dictionary header and word string are not required by the program. Only the CFA and PF fields (or logical equivalents) would exist in the program. While this threading model makes sense for a Forth system (which actually has a dictionary) it does not make as much sense for a cross-compiled program lacking a dictionary.

In the following descriptions a name preceded by an @ indicates the address of the associated name. A branch is indicated by -> preceding the name.

In this threading model the Forth VM contains two of the three word execution stages. Mainframe machine instructions are use to illustrate the operation of the virtual machine. The pseudo code in the comments use a parenthesis to indicate fetching memory content. The arrows indicate where the result of the action is placed.

```
NEXT     DS    0H
* NEXT execution stage machine instructions
         L     W,0(,IP)     (IP) -> W   W contains A(CFA)
         LA    IP,4(,IP)    IP + CELLSIZE -> IP for return
         L     X,0(,W)      (W) -> X    X points to CFA insn
         BR    X            JP (X)      enter CFA insn

ENTER    DS    0H
* Colon word entry execution stage machine instructions
         PUSHR IP           Save the IP on return stack
```

```
LA      IP,4(,W)      W + CELLSIZE -> IP (points to PF)
B       NEXT          Fetch
```

A colon word definition in this threading model looks like this. The execution tokens in this model are the addresses of each word's CFA.

Header	Word (variable)	CFA	PARAMETER FIELD						
		@ENTER	@CFA1	@CFA2	@CFA3	@CFA4	@CFA5	@CFA6	@; CFA

A code word definition in this threading model looks like this. Code words in this model may use the X register content as its base register.

Header	Word (variable)	CFA	PARAMETER FIELD						
		@MY_PF	MI1	MI2	MI3	MI4	MI5	MI6	->NEXT

The last execution token in a colon word definition, the code field address of the word `;` is itself a code word. This word contains the machine instructions that perform a colon word's exit stage processing. Like any code word, the last thing it does is pass control to the Forth VM's next stage processing.

Header	Word	CFA	PARAMETER FIELD	
	;	@MY_PF	Colon word exit stage machine instructions	->NEXT

The word `;` may alternatively be implemented with its CFA containing @EXIT, where the word's machine instructions are part of the Forth VM. In this case the word `;` has no parameter field.

The following illustrates the ; word definition and the exit stage processing for the indirect thread model.

```
EXIT      DC    A(*+4)      The ; word's code field address
          RPOP  IP          Pop the return IP from the stack
          B     NEXT        Continue previous colon word
```

Direct Threading

Direct threading removes one of the fetches by replacing it with a machine instruction. The word's definition is defined by a machine instruction that branches to its machine instructions. This model uses one fetch and a branch:

1. Colon usage: an address to the word's definition, for example, its dictionary entry's PF.
2. Word definition: branch to the word's machine instructions.

Earlier discussion of classical Forth systems and their terminologies were predicated on the assumption that the direct threading model was being used.

For code words, the PF area contains its machine instructions.

For colon words, the PF area must prime the ENTER execution stage with some instruction snippet to ensure the Forth VM knows where its colon definition resides.

The Forth VM contains the NEXT and ENTER execution stages outside of any colon or code word definitions. All or a part of the ENTER execution stage, is physically part of its respective colon word definition. If split, some of ENTER will be part of the Forth VM. ENTER must pass control to NEXT to start retrieving the execution tokens of the newly entered colon word definition.

Why is there any machine instructions at all in the colon word definition if ENTER just passes control to NEXT? The Forth VM must know when a word is a colon word with execution tokens and when the word is a code word of machine instructions. In the indirect threading model, the CFA provides that role by where it points the VM for execution of the word. With the CFA eliminated, all that is left is machine instructions. So at least one (that passes control to ENTER) is required.

EXIT is entered via its own code word definition when invoked by the a colon word definition. EXIT may use a portion of the Forth VM to perform the function or perform it entirely within its definition. In this later case, it will pass control to NEXT.

In a cross-compiled program, only the PF field is required in the program. The dictionary header, word string and CFA are eliminated.

In this threading model the Forth VM still contains between two of the three word execution stages. NEXT remains. ENTER may be partially present or completely moved to the colon word definition.

```
NEXT     DS    0H
* Colon word next execution stage machine instructions
         L     W,0(,IP)     (IP) -> W   W contains A(PF)
* W points to the word's Parameter Field (an instruction)
         LA    IP,4(,IP)    IP + CELLSIZE -> IP
         BR    W            JP (W)      enter CFA insn

ENTER    DS    0H
* Colon word entry execution stage machine instructions
         RPUSH IP           Save the IP on return stack
         LA    IP,4(,W)     Point IP to first execution token
         B     NEXT
```

The colon word definition looks like this with the direct threading model. The execution token is the address of a word's parameter field. As this diagram of a colon word definitions illustrates, the parameter field now contains a mixture of machine instructions and execution tokens.

<table>
<tr><td rowspan="2">Header</td><td rowspan="2">Word (variable)</td><td rowspan="2">CFA</td><td colspan="6">PARAMETER FIELD</td></tr>
<tr><td>->ENTER</td><td>@PF1</td><td>@PF2</td><td>@PF3</td><td>@PF4</td><td>@; PF</td></tr>
</table>

The code word definitions now becomes this. If it needs a base register it can use the *W* register.

Header	Word (variable)	CFA	PARAMETER FIELD						
			MI1	MI2	MI3	MI4	MI5	MI6	->NEXT

And the definition of the word `;` now is this.

Header	Word ;	CFA	PARAMETER FIELD	
			Colon word exit stage machine instructions	->NEXT

The `;` word's definition, EXIT, would be similar to this. In essence the same as in the indirect threading model without a code field address.

```
EXIT    DS    0H
        RPOP  IP
        B     NEXT
```

Subroutine Threading

With subroutine threading, addresses as pointers are eliminated entirely and a colon word's definition is compiled as a series of machine subroutine calls to the machine instructions of each of the words it uses. If the machine associates a subroutine call with a hardware implemented return stack, a word's initiation semantics can be completely eliminated. This model is essentially direct threading but using a machine instruction in the colon definition instead of an execution token.

This effectively moves the NEXT execution stage entirely into the machine hardware, turning the hardware subroutine return stack into the Forth VM's return stack, and turning the Forth VM *IP* register into the hardware instruction address register.

For both colon and code words, subroutine threading operates the same from the viewpoint of a dictionary entry's field usage.

The Forth VM may now totally reside within word definitions and does not exist with any code external to the words. ENTER and EXIT now reside completely within the words definitions. NEXT has been replaced by the hardware.

In a cross-compiled program, only machine instructions and in-line data exist. All parameter fields are now code words.

The design of the hardware processor and its instruction set architecture will determine if this threading model is possible or desirable. This model is best suited for platforms that embed the machine instruction address within a call or branch type instruction. This model also allows for common and simple words to have their machine instructions compiled in-line, avoiding the subroutine call entirely.

The colon word definition within this threading model looks like this. The symbols -> following a name indicate a machine instruction subroutine call. The symbols <- indicate a machine subroutine return sequence. Depending upon the machine instruction set capabilities there may be some instructions at the start of the colon word to facilitate the subroutine enter processing. Corresponding instructions may be necessary to return.

Header	Word (variable)	CFA	PARAMETER FIELD					
			ENTER	PF2->	PF3->	PF4->	PF5->	EXIT

The colon word parameter field might be similar to this.

```
COLONWRD DS    0H
* Enter execution stage processing
         RPUSH IP      Save calling colon word return address
* Execution “tokens” follow
         BRAS  IP,PF2  Call word at PF2
         BRAS  IP,PF3  Call word at PF3
         BRAS  IP,PF4  Call word at PF4
         BRAS  IP,PF5  Call word at PF5
* Exit execution stage processing
         RPOP  IP      Retrieve the return address
         BR    IP      Return to calling colon word
```

The code word definition changes slightly.

Header	Word (variable)	CFA	PARAMETER FIELD						
			MI1	MI2	MI3	MI4	MI5	MI6	<-

A code word would be constructed like this. The example shows the option of accessing data inline within the calling colon word. For a colon word there is no need to push the return address onto the return stack.

```
CODEWRD  DS    0H
[        LR    X,IP      Save caller's inline if needed  ]
[ Set up a local base register if needed.  ]
* Do my work
[        LA    IP,x(,IP) Point past my inline data in caller ]
         BR    IP        Return to calling word
```

The word ; has been eliminated, being replaced by machine instruction(s) returning control to the calling colon word at the end of each colon word definition.

Token Threading

The two previous examples improved on the design of the indirect threading model by moving

closer to the hardware offering execution speed enhancements. The token threading model goes in the opposite direction by replacing a word's definition address where it is used in a colon word definition with a smaller byte code value. In classical Forth terms, a token. The byte code is translated to the word's definition address and then proceeds as in the indirect threading model. This threading model is really indirect threading using a reduced size addressing model.

1. Colon usage: a byte code for the compiled word
2. Byte code translation: Convert the byte code to the word's code field address.
3. Branch to the word's machine instructions

It may also be possible to eliminate the third step by directly translating the byte code into the address of the word's machine instructions.

All words, code and colon, get assigned a byte code as part of the compilation process. The dictionary gets replaced with a byte-code translation table with the word's CFA field address being its only entry. Why is the address of the CFA used instead of the address of the PFA? The primary purpose of the CFA field is to differentiate between colon words and code words. The distinction remains a requirement for token threading.

There is no change for code words from the code word definition used by the indirect threading model.

For colon words, the execution token becomes the token associated with the words entry in the Token Translation Table.

As with the indirect threading model, the Forth VM will contain NEXT and ENTER outside of any word definition. EXIT is entered via its own code word definition when invoked by a colon word definition. EXIT may use a portion of the Forth VM to perform the function or perform it entirely within its definition. In this later case, it will pass control to NEXT.

This model almost always is less efficient than the other threading models. However, the use of byte codes will typically reduce the overall size of the Forth implementation. By using a byte code encoding scheme that allows for one or two byte sized byte codes, the number of supported words can be increased while continuing to reduce size. Further reductions can be achieved by only placing in the Forth system byte codes actually referenced, reducing the number of required byte codes, their corresponding translation table entries, and code word machine instructions.

The ANS Forth requirement for execution tokens to be the size of a CELL can be met by ensuring that execution tokens when placed on a stack are the size of a CELL. The actual size of the information compiled into a colon word's data field is not really defined by ANS Forth.

The Forth VM resembles that of the indirect threading model. It adds to the VM the Token Translation Table (TTT). The TTT converts the token to the address of the words CFA. In almost all cases the token becomes an index into the TTT. The exact algorithm may vary. For this description, the token is a displacement from the start of the Token Translation Table.

```
NEXT     DS    0H
* NEXT execution stage machine instructions
         ICM   X,B'0011',0(IP) (IP) -> X   X contains token
         LA    IP,2(,IP)   Point to next token
* Locate the word's CFA address from the Token Translation Table
         LA    W,TTT(X)    W contains A(A(A(CODE)))
* Point to the actual CFA
         L     W,0(,W)     W contains the CFA address A(A(CODE))
         L     X,0(,W)     (W) -> X    X contains A(CODE)
*                          For code words it is the PF address
* Enter the CFA for the word
         BR    X           JP (X)      enter CFA insn

ENTER    DS    0H
* Colon word entry execution stage machine instructions
         RPUSH IP          Save the IP on return stack
         LA    IP,4(,W)    W + CELLSIZE -> IP
         B     NEXT        Fetch tokens from new word

* Token Translation Table
TTT      DS    0F
CEXIT    DC    A(EXIT)
TEXIT    EQU   CEXIT-TTT
C1       DC    A(P1CFA)
T1       EQU   C1-TTT
C2       DC    A(P2CFA)
T2       EQU   C2-TTT
etc.
```

The Token Translation Table takes an additional four bytes (in this description) for each word. The number of bytes to reference the word in a colon definition is reduced from 4 to 2 (in this description). For any colon word referenced more than two times, space is saved.

The colon word takes on this content.

Header	Word (variable)	CFA	PARAMETER FIELD						
		@ENTER	T1	T2	T3	T4	T5	T6	TEXIT

The code word has the same content as in the indirect threading model. Code words in this model may use the *X* register as a local base register.

Header	Word (variable)	CFA	PARAMETER FIELD						
		@MY_PF	MI1	MI2	MI3	MI4	MI5	MI6	->NEXT

And the **;** word becomes, and the label `EXIT` is associated with this word's CFA.

Header	Word	CFA	PARAMETER FIELD	
	;	@MY_PF	Colon word exit stage machine instructions	->NEXT

The following illustrates the **;** word definition and the exit stage processing for the indirect thread model.

```
EXIT     DC    A(*+4)     The ; word's code field address
         RPOP  IP         Pop the return IP from the stack
         B     NEXT       Continue previous colon word
```

Speed vs. Size Trade Offs

Each of the threading models has inherent speed vs. size trade offs. The following table ranks each in relative terms.

Model	Size	Speed
Token Threading	Small	Slowest
Indirect Threading	Medium	Slow
Direct Threading	Medium	Medium
Subroutine Threading	Largest	Fastest

The degree of difference in performance and size will vary on different processors. Processors that embed within an instruction a memory address are suited for subroutine threading. Processors that do not have that option are not well suited for subroutine threading. They will require an address field in memory some place and hence are more natural candidates for the indirect or direct threading models. Where constrained resources are present, token threading should be seriously considered. The table suggests that direct threading is the best option to achieve reasonable speed in a reasonably sized system.

A Forth system's cell size influences the decisions around size and speed. Execution tokens are compiled into ANS Forth words. For ANS Forth compatibility, the execution token must be the size of a cell. In practice this is usually either two-, four- or eight-bytes. The cell size then becomes a factor in the trade off. Can a cell be eliminated? How much space or instructions are required by each cell or set of machine instructions?

Speed is expected to be improved by reducing the code path of machine instructions involved in the three execution stages of a Forth word described in the “Execution Semantics” section. Reverting to code words may be an option for some words to improve speed. Size reductions are mostly around reducing elements in the system.

Size is not always driven by processor constraints. Forth system storage factors may drive the need for size considerations. In this context, token threading may be ideal for storage and a different threading model might be perfect for execution. A hybrid approach using a token

analyzer converting tokens into one of the other models might be the best of both worlds. The analyzer itself might be best coded using tokens and the rest of the system utilizes a different threading model.

Starting and Stopping the Forth VM

Starting the Forth VM is simple. Some machine instructions will prime the *IP* register with the location of an execution token. Then the instructions will cause the Forth VM to enter for the first time the NEXT execution stage. The machine instructions can do this the same way any code word would by passing control of the machine to the first instruction of the NEXT execution stage.

Now that the Forth VM is executing semantics via the three execution stages, stopping the Forth VM becomes a matter of executing a word that exits the execution stages. The ANS Forth standard provides the **`BYE`** word for this purpose. This or a similar word could be compiled into the first colon word that controls the entry and exit of the virtual machine. If a text interpreter is available, the termination could be entirely left up to the provider of text input to the text interpreter, usually a human.

ROM Memory Model

The ANSI Forth Standard addresses a memory model in only one context. This is found in Appendix E.5, “ROMed application disciplines and conventions”.

> When a Standard System provides a data space that is universally readable and writable we may term this environment “RAM-only”.

The definition of data space and the need to compile new words into it makes the assumption that a RAM-only environment is required for an ANS Forth Standard System. Appendix E.5 acknowledges some environments may utilize ROM. The following describes the key elements of such an application:

> Programs designed for ROMed application must divide data space into at least two parts: a writable and readable uninitialized part, called “RAM”, and a read-only initialized part, called “ROM”. ... A Standard Program must explicitly initialize the RAM data space as needed.
>
> The separation of data space into RAM and ROM is meaningful only during the generation of the ROMed program. If the ROMed program is itself a standard development system, it has the same taxonomy as an ordinary RAM-only system.

Note that a ROMed application is “generated”. This implies a process external from the application performing this task. This process must implement the equivalent of a text interpreter to transform an application source text into the ROMed application. So another Forth system is required to perform this task. The F Minus and TOF systems are examples of this model.

A ROMed application provides a degree of security for the applications functionality. The

portion of the data space in ROM can not be corrupted. It is for this reason that this topic is included here.

In the ROMed application model more than the data space is effected. The following table illustrates what is placed where. Items with a green background are within ANS Forth standard's scope. Other items are not within the scope of the ANS Forth standard. Code space, although as a concept is covered by the ANS Forth standard, its contents and its relationship to the data space is not. For that reason, it is shown with a yellow background.

ROM	RAM
Colon word definitions	Data stack
Code word definitions	Return stack
Constant values	Variable values
Code space	Transient data space areas
Forth VM	Transient code space areas
Other re-entrant machine instructions	

In this model, locations used within RAM are uninitialized data areas from the perspective of the portions of the system in ROM. RAM in this model acts like a `DSECT` in mainframe terms or a `.bss` section in Unix terms.

The next section of this document discusses:

- Why the ROMed application model should be the foundation for mainframe Forth systems and
- How this model is adapted to mainframe systems.

Multitasking

This section examines the implications related to Forth multitasking. First classical multitasking schemes are examined and then the implications related to expanding the concepts for the potential objectives of this document: preemptive protected tasks.

Classical Forth Multitasking

A classical Forth system operates as a single “task” with access to all system memory. The “task” executes the Forth VM. The single “task” that executes in a classical Forth system is the text interpreter, itself made up of various words defined in the global dictionary.

Moving from a single task to multiple requires separation of task specific context, in particular, data areas that have a single representation in a single task system, to multiple instances. Separate instances of the Forth VM context exposed to the Forth program must exist:

- VM registers: *IP*, *SP*, and *RP*; and
- program stacks to which *RP* and *SP* point, including in the case of the data stack, *TOS*.

Separate Forth language variables, for example, **BASE**, and storage areas, for example, **PAD** and related words), and more generally the transient data areas must exist. Other storage areas provided by the system, for example buffers used for input/output, must also be specific to the task. The ANS Forth Block Word Set would be an example of this class of storage usage.

Adding the concept of multitasking to this single task system leverages what is in place in the single “task” system. The only place where a task can be created is through the dictionary. Words are defined that allow the text interpreter to create a task, or rather, its representation through its TSA, start the task, end the task, etc. Separating the data of one task from another can be accomplished by the text interpreter by creating a separate vocabulary for each task. Each vocabulary can define its own set of **PAD** related words, **BASE** and others.

Beyond separating **VARIABLE** definitions, each task must have separate data and return stacks. Minimally, they must be located through the TSA. Alternatively, the task's stacks may reside in the TSA. Either approach allows the task's *SP* and *RP* registers to be initialized when the task is started. If the TSA contains a sequence of execution tokens similar to those required when the Forth system starts execution, the TSA can also be used to initialize the *IP* register. If such a sequence is not present, an execution token is required that identifies the word that does execute them.

The simplest multitasking design uses cooperative multitasking. When cooperative multitasking is in use, each task must voluntarily give up control of the Forth VM. In this design only a few Forth VM registers require preservation. These are typically stored within the TSA addressed by the Forth VM *UP* register. Each TSA is usually linked to the others in a circular list. The *UP* register contents itself does not require preservation because the linked list provides the address of the next task area, the contents of the *UP* register, when it is given access to the Forth VM. The registers requiring preservation are: *IP*, *TOS*, *SP*, and *RP*.

Another common practice in multitasking classical Forth systems is to implement the words that require separate storage (for example **PAD** and **BASE** and friends) within the TSA. This allows the single implementation of these words in the dictionary to be shared with each task. The words access the *UP* register to locate their respective content. This technique obviously changes how such words are implemented.

When preemptive multitasking is used rather than cooperative multitasking, the TSA must be enhanced to preserve the contents of the hardware registers. This of course results in preserving the Forth VM registers that reside in hardware registers. Any Forth VM registers requiring preservation but implemented in memory, must also be preserved by moving their contents to the TSA.

Hardware interruptions implemented in Forth, must have their own equivalent of a TSA that allows initialization of the Forth VM for the use by the ISR. Typically, an ISR will operate with interruptions disabled, so preservation of the ISR's Forth VM context is not required. Preserving the context of the interrupted task though is of course necessary in its TSA.

For a Forth application, as opposed to a Forth system, some things are unnecessary, such as the ability to allow each task to create its own word definitions.

The *UP* register is the anchor for access to the user specific data. Usually the *UP* register does not need to be saved, because it is really derivable from the chain of TSA's.

Most multitasking systems assume that tasks and related areas are allocated from memory at run time using various words to accomplish it.

There are no standard words for multitasking, but certain ones occur frequently:

- **MULTI** – Enable multitasking
- **SINGLE** – Disable multitasking
- **THREAD / TASK** – Defines a task's TSA.
- **PAUSE / PREEMPT** – Switch to another ready task (by a task)
- **RESTART** – Dispatch a task ready to run
- **STOP** – End a task
- **SEND** – Send a message to a task
- **RECEIVE** – Receive a message from a task
- **ISR** – Define and end a Forth interrupt service routine (ISR)
- **INT-ON** – Enable an interrupt
- **INT-OFF** – Disable an interrupt
- **INT-SET** – Install an interrupt service routine (ISR)
- **INT-REMOVE** – Remove an ISR
- **IPREEMPT** – Switch to another ready task by an ISR

- **`SIGNAL`** – Signal a semaphore is available
- **`WAIT`** – Wait for a semaphore to become available
- **`ME`** – Identify the running task

Mainframe Systems

This section presumes an understanding of mainframe system architecture.

Forth VM Considerations

While the discussion of threading models was nearly the last topic discussed above, it is evident that it has a major impact on the design of the Forth VM as implemented by the hardware machine instructions.

Some observations about mainframes:

- Mainframe systems usually have relatively large amounts of storage available compared to many hardware platforms on which Forth VM's have been implemented. The exception being some of the System/360 models.
- Hardware does not support directly a stack usable for the purposes of a Forth VM. The stack available with the PROGRAM CALL instruction is not suitable for either Forth VM stack.
- Compared to many hardware platforms, the mainframe has a large number of hardware registers, sixteen.
- Incorporation of an address within an instruction is not universally available. These systems have this option:
    1. System/360 Mode 20 direct addressing mode,
    2. ESA/390 and z/Architecture® models with relative addressing instructions: BRANCH RELATIVE, BRANCH AND SAVE RELATIVE, and
    3. z/Architecture models in either ESA/390 or z/Architecture modes with BRANCH RELATIVE LONG, and BRANCH AND SAVE RELATIVE LONG instructions only on z/Architecture.

  Otherwise, systems lacked this ability.

These observations have these implications for a Forth VM implementation on mainframe systems:

1. Mainframes are best suited for direct or indirect threading models.
2. Subroutine threading is only reasonable for systems supporting embedded addressing.
3. Token threading should be considered for small memory mainframe systems.
4. All Forth VM registers could reside in hardware registers on most systems.

In all cases, the stacks must be implemented in software. Fundamentally this means adjusting the *SP* and *RP* Forth VM registers for every pop and push from or to the stacks respectively. To minimize these stack operations in software, the TOS is strongly recommended to be in a hardware register. Because the pop and push operations adjust the

SP and *RP* registers by the size of a CELL, the CELL size is also strongly recommended to be in a hardware register.

Multitasking

When the Forth VM supports multitasking, it starts to take on attributes of an operating system kernel. Such kernels separate the activities within a hardware system into those of the kernel and those of the user. The Forth VM User Task Area starts to create that separation.

Mainframe systems from the beginning were designed with multitasking, specifically preemptive multitasking, as a standard capability. This is implemented using an interrupt mechanism that requires preservation by the interrupt service routine of the interrupted program's state (PSW and register contents), in Forth terms, a User Task Area, and a dispatching algorithm for passing control to a previously interrupted user task.

While the previous section suggests that the Forth VM *UP* register could reside in a register and Brad Rodriguez encourages that, on mainframe systems designed for multitasking from the beginning, a feature not normally available on other platforms lacking virtual storage, exists. Mainframes can provide storage protection between tasks. Normally this is today provided by many systems only through utilization of virtual storage implemented by a processor memory management unit. Mainframe systems do not require virtual storage to implement storage protections.

To allow user programs maximum use of the hardware registers, and disconnect “kernel” functions from the user tasks, mainframes historically used the logical equivalent of a *UP* register implemented in protected storage. A mainframe Forth VM supporting multitasking should continue the standard practices of mainframes, placing the *UP* in storage.

Memory Model Considerations

Mainframes systems, with the exception of a some System/360 models, have storage protection facilities, protecting user tasks from corrupting other tasks, control program, or kernel operations. This mechanism utilizes storage keys associated with a storage page. Storage page sizes vary between systems. They are either 2048 or 4096 bytes in size.

Mainframe Addressing

Mainframes have four memory addressing mechanisms for instruction operands (as opposed to branch or subroutine calls discussed above):

- Addressing storage directly by a machine instruction's embedded address. Only register 0 universally participates in this mode. The instruction displacement constitutes the embedded address. In this mode, embedded addresses allow an address of 0-4095 within an instruction.

  The System/360 Model 20, extends this by using the base register encoding within the instruction to augment the displacement to increase embedded addressing to 32K. In

base register and displacement terms, base registers 0-7 can be thought of as containing fixed values. Registers 0 and 4 contain 0 (0K). Registers 1 and 5 contain 4096 (4K). Registers 2 and 6 contain 8,192 (8K). Registers 3 and 7 contain 12,288 (12K). This provides direct addressing for locations 0-16,383 (0x0-0x3FFF).

- Addressing storage using a register's content as an address. Registers participating in direct mode addressing are excluded from this option.
- Effective addressing using a base register plus displacement, and
- Indexed addressing using an index register in addition to the base register and displacement. This method is not available on System/360 Model 20.

Most platforms have these options, but the mainframe implementation is particularly robust because fifteen of the 16 registers can be used directly or take on the role of a base register or an index register. Registers participating in direct addressing are precluded from this usage. That is register 0, universally, and registers 1-7 on System/360 Model 20.

For a summary of this information see the table at the beginning of the “Implementing on Mainframes” section.

A Forth VM memory model that separates read/write memory areas from the read-only ones (for example F Minus) can be readily implemented using an index register as the starting location of the area and a base register for an offset into the area.

The Forth VM NEXT execution stage must be universally accessible. Returning to this set of machine instructions marks the end of nearly every word. Colon words have a common set of requirements for the ENTER execution stage and EXIT execution stages. How can this be achieved on mainframe systems? For some models direct addressing could be used to access these routines. For other systems, particularly those without embedded addressing, these must be accessible via a register in conjunction with or without a displacement value.

Securing Tasks

Mainframe systems from the earliest days provided a means for securing separate processes from corrupting each other or the control program required for mainframe systems. Only the vary earliest and smallest systems excluded this capability. In the context of Forth, the bare metal Forth system has the role of being its own control program. ANS Forth offers only the ROMed application conventions that address securing portions of an ANS Forth system.

In this model, locations used within RAM are uninitialized data areas from the perspective of the portions of the system in ROM. RAM in this model acts like a `DSECT` in mainframe terms or a `.bss` section in Unix terms.

RAM resident portions still have potential for corruption. Any direct use of storage addresses in RAM have the potential to corrupt. The ANS Forth VM *SP* and *RP* registers are not directly alterable. The stack contents of both are alterable, but not the pointers themselves. Consequently, stack overflow or under flow could result in data corruption.

Mainframe systems use a mechanism that partitions storage into equal sized pages. Each

page is allowed access by a specific process or group of processes sharing a “key”. Pages may be contiguous or not. Adjacent pages assigned different keys create access boundaries that protect the page from access by an application accessing the other page. Even a system that supports one application could benefit from these protections.

The mainframe defines 16 such storage access keys. What this means for a mainframe Forth system is that 16 separate processes can be protected from each other. If each of the RAM areas of a task (data stack, return stack, and transient) are separated and assigned different keys than the other applications, the tasks are protected from each other. This requires a minimum of 3 pages per task. Such a model certainly increases the run-time storage requirements of the system, but do not increase the ROM area requirements by much. Only systems with sufficient storage can support storage protection.

Mainframes do not actually have ROM storage. But, by assigning a unique storage key to a ROM area, it can be protected. Optimum protections are provided when the ROM area is assigned a key of 0, but all tasks operate in a different key. They can execute out of the key 0 but can not alter it in any way.

Accessing the Data Space

Ultimately accessing a data space location involves accessing a memory location. Two mechanisms are useful in the context of the Forth VM:

- direct address binding and
- offset binding.

The first case uses the actual memory location. In the second case, the memory location is stored in the data space as an offset to which a starting memory location is applied to derive the actual memory address. The term **location** will be used here when a portion of the data space or code space is being accessed independent of the binding method for the actual address.

On some processors, offset binding would entail undesirable overhead. On a mainframe system, a memory location can be established from the sum of a base register, a displacement and an index register. This is a natural for use of the offset binding design. The index register could contain the starting location of the portion of the data space, the base register the offset and a base, typically zero, would directly result in the correct memory address.

For an environment where data space and code space may reside in readable and writable locations, two offset binding addresses are required, one for the read-only area and one for the read/write area. Only a single read-only area is needed. The read-write area can be be a single one or multiple. See the discussion on “Multitasking” for a reason why multiple read-write areas might be useful.

The offset binding design really creates two new Forth VM registers:

- a ROM base pointer (*ROMBP*) and

- a RAM base pointer (*RAMBP*).

For a Forth system that uses anything other than a token thread mode, relocation is extremely difficult. A token threaded system simply needs to relocate the token binding table to relocate itself or generate the thread model doing the binding at that point in time.

Input/Output Operations

ANS Forth offers a few words related to input and output operations, but otherwise is really silent. Mainframe systems, particularly at the bare metal level, have unique requirements. Generally input/output (I/O) utilizes CPU interruptions. In addition to interruptions, specific storage resident structures participate in I/O. These structures vary in size and content depending upon the basis of I/O operations, but all of them must be located in storage. Access to storage is also required by the I/O interrupt service routine for the purpose of preserving and presenting results to the I/O requester.

Cooperation between the I/O interrupt service routine and the I/O requester implies that some structure is “shared” between them. Here, this structure is called the **Device Block** (DB). Different I/O architectures based upon different I/O machine instructions require different sized DB's and have, obviously, differing content or structure. In all cases it will preserve I/O interruption information.

A device is identified by a unique hardware identifier called here its **device address** and provided in the device's DB. Different I/O architectures use different forms of address of different sizes. For some architectures, the device address is not consistent over time and can change. These systems offer a consistent designation called its **device identifier**. For systems lacking a device identifier, the device address can serve that role.

Some architectures require the system to know before hand what addresses it will use and others do not. The ability to probe devices to identify them has not always been available. Some architectures did not support it, some partially and some for all devices. The probe operation provides information about the type of device that is associated with a specific device identifier. The manual mapping of used device identifiers to a specific device type will be provided by a stable structure, the **Input/Output Device Definition** (IODD).

In addition, the IODD must be associated with a specific DB. This is provided by means of a new single cell data type: **input/output identifier**, or *ioid*. This can be either an index into a table of DB's or the address of the DB itself. From the standpoint of Forth, an *ioid* is an abstract data type only to be used for the reference of a DB.

In the very earliest and smallest systems, a specific logical device function might be assigned to a specific device identifier with the same device always associated with it. Whether the device identifier had to be manually specified or could be learned by a probe process, connection of the logical use of the device still becomes a manual process. Only some of the most recent Linux kernels learned this lesson and address it. Mainframe systems learned this early on. The logical mapping of a function to a DB can be established in Forth by means of a **`CONSTANT`** or **`VARIABLE`** word definition whose name identifies the function and its value provides the *ioid* associated with it.

All mainframe systems can utilize an I/O interrupt service routine. I/O can be performed without it but it is not recommended. The primary responsibility of the ISR is to preserve the interrupt information within the device's DB. As the shared structure between the ISR and the application, any need for special interrupt handling for the device requires the ability to allow the application to provide its own ISR processing word.

Colon Words as Interrupt Service Routines

From the standpoint of the Forth VM, an Interrupt Service Routine (ISR) is nothing more than a different task. By establishing a task area for each ISR from which the Forth VM''s registers can be initialized (with the associated RAM areas), a colon word can be readily used to perform ISR processing. This is really the same as starting the Forth VM all over again with a new context.

The native code ISR interface must perform the following actions:

1. Respond to the interruption by saving the state of the executing task,
2. Process the interruption by passing control to the Forth VM after initializing it to execute the colon word for that purpose, and,
3. Return the last or some previously interrupted task by restoring its state

Pretty standard fare for an interrupt service routine

Because the ISR must cleanly start from uncorrupted data, it should be part of the ROM protected data.

Responding to the Interrupt

On mainframe systems any one of six interruption classes may cause an interrupt. The action of the machine interrupt is to store the current Program Status Word (PSW) along with some interruption related data in storage locations assigned for this purpose. The PSW is the only piece of program state related to the interrupted program that is saved by the hardware.

Preserving the rest of the interrupted program's state involves saving the contents of the program's CPU register contents so that they may be restored. The problem with that is, some register is actually required to store the content in the task user area. But none of the registers are available because they all contain program state. The contents of at least one register must be saved without using a register. Only direct addressing allows for this to occur. This restricts the location where the register's content is saved to the first 4096 bytes of storage. If the ISR native code itself is not located within these 4096 bytes and it requires a base register then at least two registers will need to be stored within the area accessible by direct addressing. When storage protection measures are considered, this requires at least this portion of the ISR be executed with the key matching the key established for the locations where the contents are stored. By having the new PSW used by the CPU to enter the ISR to have the same key as the page in storage, the ISR can store data (in this case the register contents) into the page.

Now that a register is available, the Forth VM *UP* register can be moved to a hardware register. Using this register as a base:

- the previously saved register contents,
- registers contents not yet saved and
- the interrupted task's PSW (found in the interrupt class' Old PSW assigned storage location)

can all be preserved in the interrupted task's User Task Area. The ISR can now prepare the Forth VM for handling the interrupt.

Executing a Colon Word ISR

Using the Forth VM in an ISR requires creating a new context for the Forth VM. This is the same as entering the Forth VM when the system is initialized or when a new task is started. All of the information is in a task area (TA). Entering the Forth VM to process an interrupt is essentially passing control to a newly initialized task. The Forth VM registers are initialized and control is passed to the NEXT Execution Stage of the word identified by the VM's *IP* register.

Once the Forth VM is running, what should it do? There are some standard requirements for each ISR. Each must process a set of information previously stored in its assigned storage location and then return allowing normal processing to resume. This could all be handled in a single code word if desired. Alternatively, each interrupt class could use its own sequence of words:

1. Accessing its interruption information, placing some or all of it on the data stack,
2. Executing its ISR processing word, leaving an exit action of the data stack and
3. Exiting the ISR as directed by the exit action found on the stack.

The tailored ISR is really only the second word in the sequence. It does not need to know anything about the ISR environment. The first and last words represent standard logic that does not need to be recreated. In fact, exiting the ISR could be the same for each interruption class.

Resuming Interrupted Processing

Resuming interrupted processing is nothing more than selecting a task for execution and dispatching it. The task might be the one that was interrupted by the interruption or some other. Refer to the following section on “Multitasking” for details.

Mainframe Multitasking

At the foundation of a mainframe Forth system is the concept of a context. Whether multitasking is supported or not, at least one context exists. Multitasking adds additional contexts. Interrupt handling includes additional contexts.

The Forth VM Context

In addition to a Forth VM's implied capabilities (executing Forth word definitions), it is characterized by

- two LIFO stacks in read/write storage,
- read/write variable and transient areas,
- its internal registers' contents, and, a
- program accessible data and code spaces.

This document uses the expression **Forth VM context** or **context** to encapsulate these logical constituents of a Forth VM. Each valid context is associated with a **context area** (CA) that is part of the code space transient area. It contains the values needed to initialize the context and a place to store context information, including machine state, during an interruption. A new data type, **context identifier** or *cid*, is established for the purpose of identifying a specific context instance. It may be an index into a table, an address, or some other value. Regardless of the internal code space meaning of the value, when used by words in the data space, the *cid* is an opaque value other than for the purpose of identifying a CA.

Knowing what contents to place in the Forth VM registers is clearly a prerequisite to all of these cases. The critical Forth VM registers are: *IP*, *RP*, *SP*, and, if used, *ROMBP* and *RAMBP*. Because the *IP* always points to an execution token, establishing the *IP* register implies it is pointing to at least one execution token, the context execution token.

While the *IP* at any time points explicitly to a single execution token, the implicit adjustment of the *IP* during a word's execution semantics means the *IP* register implicitly points to a contiguous sequence of execution tokens.

Normally the context execution token should never return. Failure to plan for this and allow for it will allow unpredictable results to occur. While this should not happen, all too often things that should not occur, do. It is here recommended that the context execution token be augmented to provide a predictable (although perhaps drastic) response to this possibility. The previous discussion of ISR in fact recommends multiple execution tokens within its context stream.

For the context to start running, control of the CPU must be passed to the Forth VM. This can be done by a simple instruction branch. But, for a mainframe system, additional hardware specific controls are required. A branch is only feasible if these hardware controls are the same. In general they may vary. So in addition to the address of the machine instruction to which control is passed, the machine controls must also be initialized and preserved during any interruption of the task. For the mainframe this is a machine specific structure called the Program Status Word (PSW).

The stacks, data space variables and transient areas are not initialized. Only their size are required for allocation within RAM storage, later initialized and used. Per the ROMed application conventions, the data initializing the RAM area must come from the ROM areas.

The location of the context stream is required to allow initialization of the *IP*. This information constitutes the Context Area Definition (CAD).

Initiating a new context then includes two parts:

- machine instructions that prepares the CA and Forth VM for use.
- starting the Forth VM with the new context (dispatching the context), thereby, causing the context execution stream to run.

This is very similar to the colon definition of a word.

Forth VM contexts are required by each of the following:

1. Initiating the Forth VM (the system context),
2. Each task in a multitasking Forth VM (a task context), and
3. Using the Forth VM in an ISR (an ISR context).

While the nature of a context's content is the same for each type of context, what information is placed in the context differs by type.

System Context

The first case, initiating the Forth VM, is really just an example of the second. The actions required to initialize the Forth VM where multitasking is not present, is identical to that for a single task. So, even a system that is not supporting multitasking, really supports it, albeit for a single task.

The system context has a special role. Within its RAM area will be placed, in addition to the usual standard Forth content, all of the context areas including that of the system itself and the I/O DB's.

The ROMed application usage of ROM and RAM described in general terms are updated here to identify the unique contents of the system context.

System Context ROM	System Context RAM
All Colon word definitions	Data stack
All Code word definitions	Return stack
All Constant values	System Variable values
All Code space	Other System Transient data space areas
Context Area Definitions (CAD's)	Context areas (CA's)
Input/Output Device Definitions (IODD's)	Device blocks (DB's)
Forth VM	
Native code ISR interface	
Native code initialization support	

ISR Context

Interrupt service routine contexts share the System ROM context with the system. An interrupt service routine is nothing more than a “task” within the system. They also share system specific areas such as context areas and device blocks with the system. What is unique are the VM architecture elements: data and return stacks and certain transient data areas, for example relating to string management. System context variables are shared with the system.

Task Context

A task context shares with system the word definitions and constants built into the application. However, all Forth architecture elements are unique to the task. So while a single set of variables may be defined for the application, tasks will have their own unique values established for them. And RAM must be allocated for all of them regardless of the usage. The universal usage of word definitions demands that the variables therein referenced must be available to all contexts in which the word is used, albeit not necessarily having the same value in separated contexts.

Transient Regions

Transient regions may change (or not). The key issue is whether the program depends upon them not changing. Transient areas are identified in association with words **PAD**, **WORD** and **#>.**

The address returned by **PAD** must be to an area at least 84 characters in length.

The transient area identified by **WORD** must be at least 33 characters in length. This word is used for text parsing. Parsing further uses an input area associated with words **>IN**, **SOURCE**, and **SOURCE-ID**.

The area supplied by **#>** must be at least (2*n)+2 characters in length, where n is the number of bits in a cell. This area is related to the words **<#**, **#**, **#S**, and **HOLD**.

Contiguous Regions

Contiguous regions are associated with words

- **ALLOT**,
- **,** (comma),
- **c,** (c-comma),
- **ALIGN,** and

- **HERE**.

As the ANS Forth standard states, these areas “shall be contiguous with the last region allocated with one of the above words. ... .” The management of these allocations only become a special consideration when secured multitasking is in use.

Context Area (CA) Content

Each Context Area (CA) requires storage for the following information:

- Hardware status save area (if interrupts or multitasking is used),
- Context Area Definition address used to define this context
- Forth VM register state,
- Data stack of data stack cells,
- Return stack of return stack cells,
- Transient areas identified by **PAD**, **WORD** or **#>**, and
- **VARIABLE** and **VALUE** storage slots of data stack sized cells.

The address returned by **PAD** must be to an area at least 84 characters in length.

The transient area identified by **WORD** must be at least 33 characters in length. This word is used for text parsing. Parsing further uses an input area associated with words **>IN** and **SOURCE**.

The area supplied by **#>** must be at least (2*n)+2 characters in length, where n is the number of bits in a cell. This area is related to the words **<#**, **#**, **#S**, and **HOLD**.

Mainframe Multiprocessing

Forth systems are generally not expected to operate within a multiprocessing environment. Their typical domain is a single processor system. The PowerPC processor binding for IEEE 1275 discusses multiprocessors, but only as attributes of the device interface. The attributes allow a CPU in the device tree to have its state altered.

Multiprocessing systems share main storage. A fixed area, called the prefix area, is logically relocated in storage for each CPU for the purpose of providing unique assigned storage locations for each processor. A multiprocessing Forth system would need to allocate these areas. The primary purpose of this area is to provide unique interruption state areas for each processor when a task running on the CPU is interrupted. Potentially the code space associated with ISR contexts could be shared. The contexts themselves would have to be unique. This would require a context area for each ISR on each CPU.

Each prefix area would have to be created and initialized before its corresponding CPU can be caused to enter the running state. The various context areas would require creation

before any contexts are started. Other issues concerning shared state and atomic updates to storage are beyond the scope of this document. Exploring multiprocessing with a Forth application may be a future objective.

Implementing on Mainframes

The following table summarizes a number of key attributes of mainframe systems over the years of importance to a Forth implementation. In previous sections, the attributes were discussed where it made sense to do so. The details have been summarized here.

Platform / **Attributes**	**S/360-20** Note 2	**S/360** Note 3	**S/370** Note 4	**ESA/390** Note 6	**z/Architecture**	
					ESA/390	**z/Architecture**
Control Mode	unique	BC/EC	BC/EC	EC	EC	z/Architecture
Input/Output design basis	XIO	SIO	SIO	SSCH	SSCH	SSCH
PSW Size in bits	32	64	64	64	64	128
Register Size in bits	16	32	32	32	32	64
Direct Mode Storage/branches	yes	yes	yes	yes	yes	yes
Direct Mode Instruction Registers	0-7	0	0	0	0	0
Direct Mode Range	0-16,383	0-4,095	0-4,095	0-4,095	0-4,095	0-4,095
Base Registers	8 (8-15)	15 (1-15)	15 (1-15)	15 (1-15)	15 (1-15)	15 (1-15)
Index Registers	none	15 (1-15)	15 (1-15)	15 (1-15)	15 (1-15)	15 (1-15)
Binary Arithmetic Registers	8 (8-15)	16 (0-15)	16 (0-15)	16 (0-15)	16 (0-15)	16 (0-15)
Relative Branch Range	none	none	none	Note 7	+/- 2G	+/- 2G
Storage Protection	no	some	yes	yes	yes	yes
Storage Protection Page Size	none	2048	2048	4096	4096	4096
Reserved/assigned region	0x0-0x97	0x0-0x7F Note 1	0x0-0x31B	0x0-0x200	0x0-0x18FF	0x0-0x18FF
Available in 0x0-0x7F	none	**0x10-0x17**	**0x10-0x17** 0x4C-0x4F	**0x10-0x17** 0x40-0x4B 0x4C-0x4F 0x50-0x57	**0x10-0x17** 0x40-0x4B 0x4C-0x4F 0x50-0x57	all
Available in 0x80-0x1FF	0x98-0x1FF	all	0xA0-0xA3 0xA4-0xA7 0xB4-0xB7 0xB8 0xBC-0xCB **0xCC-0xD3** 0xD4-0xD7 0xF0-0xF3	0xA4-0xA7 0xA8-0xB3 0xB4-0xB7 **0xCC-0xD3** 0xF0-0xF3 0xFC-0xFF 0x110-0x117 0x118-0x11F	0xA4-0xA7 0xA8-0xB3 0xB4-0xB7 **0xCC-0xD3** 0xF0-0xF3 0xFC-0xFF 0x110-0x117 0x118-0x11F	**0xCC-0xD3** 0xD4-0xD7 0xD8-0xE7 0x118-0x11F 0x180-0x19F
Available in 0x200-0xFFF	all	all	**0x200-0x31A** **0x31C-0xFFF**	all	all	all

Platform / Attributes	S/360-20 Note 2	S/360 Note 3	S/370 Note 4	ESA/390 Note 6	z/Architecture	
					ESA/390	z/Architecture
			Note 5			
Available in 0x1000-0x18FF	all	all	all	all	all	0x1000-0x11FF 0x1400-0x17FF
Storage Size Range	4K-16K	16K-6M	64K-16M	2G max	2G max	5 GB-3T
Probed I/O devices	no	no	some	all	all	all

Note 1: Region does not include diagnostic scan out area.

Note 2: The S/360-20 has a control mode unique to itself, similar to S/360 BC mode.

Note 3: The 360-67 can operate in basic control mode (360-65) or extended control mode, native 360-67 operation. All other S/360 systems operate only in BC mode.

Note 4: The S/370 series can operate in basic control mode for backward compatibility with most of the S/360 series or extended control mode, native S/370 operation. S/3**7**0 EC mode is not compatible with S/3**6**0 EC mode.

Note 5: With the exception of the byte at 0x31B, this entire area is available.

Note 6: For the purposes of this table, 370-XA, ESA/370 and ESA/390 are considered together.

Note 7: Only ESA/390 supported relative branch instructions. 370-XA and ESA/370 did not. The relative branch range was +/- 16K in ESA/390 systems.

XFORTH Implementation

This section discusses the implementation of XFORTH, a Forth implementation in Python designed to cross-compile mainframe Forth applications. The previous sections provide generic descriptions. This section links the previous sections to the XFORTH implementation.

Background

XFORTH is based upon a modified version of a Python cross-compiler targeting the MSP430 processor and a set of related tools also developed in Python. The original package is available at:

https://pypi.python.org/pypi/python-msp430-tools/0.6

This package has been radically factored to support extensions, removing MSP430 specific functionality, while retaining foundational Forth text and execution interpretation under Python as a cross-compiler.

This modified version of the MSP430 package can be found in the Stand-alone Tool Kit `github` project at

https://github.com/s390guy/SATK

in the `xforth` directory. As with the original package, the contents of this directory are released under the Simplified BSD License.

XFORTH provides specific extensions to `xforth` targeting mainframe systems and the ASMA assembler available with the SATK. XFORTH is released under the GPLv3 License as is the rest of the SATK, `xforth` being the exception sited above.

Overview

XFORTH is a Forth application cross-compiler. Between command-line settings and an initial source file containing Forth statements, an assembler source file is created targeted for assembly by ASMA, the SATK mainframe assembler. The assembled application is ready for bare-metal execution in the targeted mainframe environment, be it real, virtual or emulated. The command-line settings define the targeted architecture and threading model used to create the application.

Three name spaces are supported by XFORTH:

- base name space of colon word definitions (shared between the Python interpreter and the target Forth application)
- built-in name space of XFORTH delivered words implemented in Python (only used by the XFORTH interpreter)
- target name space composed of targeted assembler code space words (corresponding

to the built-in words in Python) and other content (for example the Forth VM itself).

As described previously there are five factors defining the final result of a cross-compiled XFORTH application. They are identified in the following table:

Factor	Req. / Opt.	Specified By	Affects
Architecture	required	Command-line	Code space
Threading model	required	Command-line	Forth VM and word definitions
Interruptions	optional	**`INTERRUPT`** words	Contexts
Tasks	optional	**`TASK`** words	Contexts
Secure tasks	optional	Command-line	Contexts, contiguous areas

ASMA Considerations

ASMA assembles source statements into a host resident file that constitutes the bare-metal image to be loaded by any one of its supported formats. An image is composed of one or more “regions” that have absolute starting addresses and are independently loaded into memory. The actual sequence of different region's content is established by the sequence in which the regions are initiated in the assembly. Control sections are established and assigned to the active region and are also placed in sequence based upon the control sections initiation sequence within the region. At least one region that starts at absolute address 0 is required to deposit at that address an IPL PSW or restart new PSW allowing the image content to be executed. Whether additional regions are required is dependent upon how the low address directly addressable prefix area is utilized.

The cross-compiled forth application will require at least three areas:

- low-address area for PSW's and interrupt information
- the Forth application “ROM” area and
- the Forth application “RAM” area or areas when interrupts and/or multitasking is supported.

Forth application “RAM” areas have no presence in the loaded image as described above. They must be loaded from the Forth application “ROM” content. Establishing their locations and content in RAM is the responsibility of the application.

This means that at least two control sections are required:

- one for the initialized low-address area, and
- one for the “ROM” application content.

The question becomes, to which region at what location are they assigned? The architecture of the target system will drive these answers. The S/360-20 is a special case that will be ignored here.

Control sections are location independent. Their location is dictated by the region in which it resides. Because each forth system has certain common components, these will be assigned to specific named sections:

1. CONTROL – IPL, PSW's and interrupt handling data.
2. CPUINIT – Initializes the environment for use of the Forth VM (for example the system context area creation or entering 64-bit architecture mode).
3. FORTHVM – The Forth virtual machine including optional interrupt handling and multitasking support.
4. CADS – Context area definitions
5. IODS – Input/Output device definitions
6. CODE – The Forth application code space words
7. DATA – The Forth word definitions

XFORTH INCLUDE vs. ASMA COPY vs. ASMA MACLIB

Where is the split between assembler code generated by XFORTH and assembler statements accessed by ASMA? In most cases assembler code accessed by ASMA will be best suited for macros resident within a macro library. This is the case where the alternative is the use of the XFORTH **ASSEMBLE** code space content. There is no need to make XFORTH tightly coupled to the architectures supported. In this case, it is recommended that XFORTH generate macro calls within its **ASSEMBLE** code space content. By using macros, architectural differences can be pushed into the assembler content itself. Such cases require standardizing the macro names and prototypes.

On the other hand, where words are implemented in the code space, these must be defined by **CODE** definitions. The actual output consumed by ASMA could also include macro definitions, but the use of **CODE** definitions allows unused words to be excluded from the generated application. Standard words can capitalize on the mainframe commitment to backward compatibility by using the same XFORTH **INCLUDE** content for words that can leverage this backward compatibility. The majority of the Python supplied built in words fall into this category.