 Table of Contents

E. ANS Forth portability guide (informative annex)

E.1 Introduction

The most popular architectures used to implement Forth have had byte-addressed memory, 16-bit operations, and two's-complement number representation. The Forth-83 Standard dictates that these particular features must be present in a Forth-83 Standard system and that Forth-83 programs may exploit these features freely.

However, there are many beasts in the architectural jungle that are bit addressed or cell addressed, or prefer 32-bit operations, or represent numbers in one's complement. Since one of Forth's strengths is its usefulness in **strange** environments on **unusual** hardware with **peculiar** features, it is important that a Standard Forth run on these machines too.

A primary goal of the ANS Forth Standard is to increase the types of machines that can support a Standard Forth. This is accomplished by allowing some key Forth terms to be implementation-defined (e.g., how big is a cell?) and by providing Forth operators (words) that conceal the implementation. This frees the implementor to produce the Forth system that most effectively utilizes the native hardware. The machine independent operators, together with some programmer discipline, enable a programmer to write Forth programs that work on a wide variety of machines.

The remainder of this Annex provides guidelines for writing portable ANS Forth programs. The first section describes ways to make a program hardware independent. It is difficult for someone familiar with only one machine architecture to imagine the problems caused by transporting programs between dissimilar machines. Consequently, examples of specific architectures with their respective problems are given. The second section describes assumptions about Forth implementations that many programmers make, but can't be relied upon in a portable program.

E.2 Hardware peculiarities

E.2.1 Data/memory abstraction

Data and memory are the stones and mortar of program construction. Unfortunately, each computer treats data and memory differently. The ANS Forth Systems Standard gives definitions of data and memory that apply to a wide variety of computers. These definitions give us a way to talk about the common elements of data and memory while ignoring the details of specific hardware. Similarly, ANS Forth programs that use data and memory in ways that conform to these definitions can also ignore hardware details. The following sections discuss the definitions and describe how to write programs that are independent of the data/memory peculiarities of different computers.

E.2.2 Definitions

Three terms defined by ANS Forth are address unit, cell, and character. The address space of an ANS Forth system is divided into an array of address units; an address unit is the smallest collection of bits that can be addressed. In other words, an address unit is the number of bits spanned by the addresses addr and addr+1. The most prevalent machines use 8-bit address units. Such **byte addressed** machines include the Intel 8086 and Motorola 68000 families. However, other address unit sizes exist. There are machines that are bit addressed and machines that are 4-bit nibble addressed. There are also machines with address units larger than 8-bits. For example, several Forth-in-hardware computers are cell addressed.

The cell is the fundamental data type of a Forth system. A cell can be a single-cell integer or a memory address. Forth's parameter and return stacks are stacks of cells. Forth-83 specifies that a cell is 16-bits. In ANS Forth the size of a cell is an implementation-defined number of address units. Thus, an ANS Forth implemented on a 16-bit microprocessor could use a 16-bit cell and an implementation on a 32-bit machine could use a 32-bit cell. Also 18-bit machines, 36-bit machines, etc., could support ANS Forth systems with 18 or 36-bit cells respectively. In all of these systems, DUP does the same thing: it duplicates the top of the data stack. ! (store) behaves consistently too: given two cells on the data

stack it stores the second cell in the memory location designated by the top cell.

Similarly, the definition of a character has been generalized to be an implementation-defined number of address units (but at least eight bits). This removes the need for a Forth implementor to provide 8-bit characters on processors where it is inappropriate. For example, on an 18-bit machine with a 9-bit address unit, a 9-bit character would be most convenient. Since, by definition, you can't address anything smaller than an address unit, a character must be at least as big as an address unit. This will result in big characters on machines with large address units. An example is a 16-bit cell addressed machine where a 16-bit character makes the most sense.

E.2.3 Addressing memory

ANS Forth eliminates many portability problems by using the above definitions. One of the most common portability problems is addressing successive cells in memory. Given the memory address of a cell, how do you find the address of the next cell? In Forth-83 this is easy: 2 + . This code assumes that memory is addressed in 8-bit units (bytes) and a cell is 16-bits wide. On a byte-addressed machine with 32-bit cells the code to find the next cell would be 4 + . The code would be 1+ on a cell-addressed processor and 16+ on a bit-addressed processor with 16-bit cells. ANS Forth provides a next-cell operator named CELL+ that can be used in all of these cases. Given an address, CELL+ adjusts the address by the size of a cell (measured in address units). A related problem is that of addressing an array of cells in an arbitrary order. A defining word to create an array of cells using Forth-83 would be:

```
: ARRAY   CREATE  2* ALLOT  DOES> SWAP 2* + ;
```

Use of 2* to scale the array index assumes byte addressing and 16-bit cells again. As in the example above, different versions of the code would be needed for different machines. ANS Forth provides a portable scaling operator named CELLS. Given a number n, CELLS returns the number of address units needed to hold n cells. A portable definition of array is:

```
: ARRAY   CREATE  CELLS ALLOT
   DOES> SWAP CELLS + ;
```

There are also portability problems with addressing arrays of characters. In Forth-83 (and in the most common ANS Forth implementations), the size of a character will equal the size of an address unit. Consequently addresses of successive characters in memory can be found using 1+ and scaling indices into a character array is a no-op (i.e., 1 *). However, there are cases where a character is larger than an address unit. Examples include (1) systems with small address units (e.g., bit- and nibble-addressed systems), and (2) systems with large character sets (e.g., 16-bit characters on a byte-addressed machine). CHAR+ and CHARS operators, analogous to CELL+ and CELLS are available to allow maximum portability.

ANS Forth generalizes the definition of some Forth words that operate on chunks of memory to use address units. One example is ALLOT. By prefixing ALLOT with the appropriate scaling operator (CELLS, CHARS, etc.), space for any desired data structure can be allocated (see definition of array above). For example:

```
CREATE ABUFFER 5 CHARS ALLOT ( allot 5 character buffer)
```

The memory-block-move word also uses address units:

```
source destination 8 CELLS MOVE  (  move 8 cells)
```

E.2.4 Alignment problems

Not all addresses are created equal. Many processors have restrictions on the addresses that can be used by memory access instructions. This Standard does not require an implementor of an ANS Forth to make alignment transparent; on the contrary, it requires (in Section 3.3.3.1 Address alignment) that an ANS Forth program assume that character and cell alignment may be required.

One of the most common problems caused by alignment restrictions is in creating tables containing both characters and cells. When , (comma) or C, is used to initialize a table, data is stored at the data-space pointer. Consequently, it must be suitably aligned. For example, a non-portable table definition would be:

```
CREATE ATABLE   1 C,  X ,  2 C,  Y ,
```

On a machine that restricts 16-bit fetches to even addresses, CREATE would leave the data space pointer at an even address, the 1 C, would make the data space pointer odd, and , (comma) would violate the address restriction by storing X at an odd address. A portable way to create the table is:

```
CREATE ATABLE   1 C,  ALIGN X ,  2 C,  ALIGN Y ,
```

ALIGN adjusts the data space pointer to the first aligned address greater than or equal to its current address. An aligned address is suitable for storing or fetching characters, cells, cell pairs, or double-cell numbers.

After initializing the table, we would also like to read values from the table. For example, assume we want to fetch the first cell, X, from the table. ATABLE CHAR+ gives the address of the first thing after the character. However this may not be the address of X since we aligned the dictionary pointer between the C, and the ,. The portable way to get the address of X is:

```
ATABLE CHAR+ ALIGNED
```

ALIGNED adjusts the address on top of the stack to the first aligned address greater than or equal to its current value.

E.3 Number representation

Different computers represent numbers in different ways. An awareness of these differences can help a programmer avoid writing a program that depends on a particular representation.

E.3.1 Big endian vs. little endian

The constituent bits of a number in memory are kept in different orders on different machines. Some machines place the most-significant part of a number at an address in memory with less-significant parts following it at higher addresses. Other machines do the opposite the least-significant part is stored at the lowest address. For example, the following code for a 16-bit 8086 **little endian** Forth would produce the answer 34 (hex):

```
VARIABLE FOO   HEX 1234 FOO !   FOO C@
```

The same code on a 16-bit 68000 **big endian** Forth would produce the answer 12 (hex). A portable program cannot exploit the representation of a number in memory.

A related issue is the representation of cell pairs and double-cell numbers in memory. When a cell pair is moved from the stack to memory with 2!, the cell that was on top of the stack is placed at the lower memory address. It is useful and reasonable to manipulate the individual cells when they are in memory.

E.3.2 ALU organization

Different computers use different bit patterns to represent integers. Possibilities include binary representations (two's complement, one's complement, sign magnitude, etc.) and decimal representations (BCD, etc.). Each of these formats creates advantages and disadvantages in the design of a computer's arithmetic logic unit (ALU). The most commonly used representation, two's complement, is popular because of the simplicity of its addition and subtraction algorithms.

Programmers who have grown up on two's complement machines tend to become intimate with their representation of numbers and take some properties of that representation for granted. For example, a trick to find the remainder of a number divided by a power of two is to mask off some bits with AND. A common application of this trick is to test a number for oddness using 1 AND. However, this will not work on a one's complement machine if the number is negative (a portable technique is 2 MOD).

The remainder of this section is a (non-exhaustive) list of things to watch for when portability between machines with binary representations other than two's complement is desired.

To convert a single-cell number to a double-cell number, ANS Forth provides the operator S>D. To convert a double-cell number to single-cell, Forth programmers have traditionally used DROP. However, this trick doesn't work on sign-magnitude machines. For portability a D>S operator is available. Converting an unsigned single-cell number to a double-cell number can be done portably by pushing a zero on the stack.

E.4 Forth system implementation

During Forth's history, an amazing variety of implementation techniques have been developed. The ANS Forth Standard encourages this diversity and consequently restricts the assumptions a user can make about the underlying

implementation of an ANS Forth system. Users of a particular Forth implementation frequently become accustomed to aspects of the implementation and assume they are common to all Forths. This section points out many of these incorrect assumptions.

E.4.1 Definitions

Traditionally, Forth definitions have consisted of the name of the Forth word, a dictionary search link, data describing how to execute the definition, and parameters describing the definition itself. These components are called the name, link, code, and parameter fields. No method for accessing these fields has been found that works across all of the Forth implementations currently in use. Therefore, ANS Forth severely restricts how the fields may be used. Specifically, a portable ANS Forth program may not use the name, link, or code field in any way. Use of the parameter field (renamed to data field for clarity) is limited to the operations described below.

Only words defined with CREATE or with other defining words that call CREATE have data fields. The other defining words in the Standard (VARIABLE, CONSTANT, :, etc.) might not be implemented with CREATE. Consequently, a Standard Program must assume that words defined by VARIABLE, CONSTANT, : , etc., may have no data fields. There is no way for a Standard Program to modify the value of a constant or to change the meaning of a colon definition. The DOES> part of a defining word operates on a data field. Since only CREATEd words have data fields, DOES> can only be paired with CREATE or words that call CREATE.

In ANS Forth, FIND, ['] and ' (tick) return an unspecified entity called an **execution token**. There are only a few things that may be done with an execution token. The token may be passed to EXECUTE to execute the word ticked or compiled into the current definition with COMPILE,. The token can also be stored in a variable and used later. Finally, if the word ticked was defined via CREATE, >BODY converts the execution token into the word's data-field address.

One thing that definitely cannot be done with an execution token is use ! or , to store it into the object code of a Forth definition. This technique is sometimes used in implementations where the object code is a list of addresses (threaded code) and an execution token is also an address. However, ANS Forth permits native code implementations where this will not work.

E.4.2 Stacks

In some Forth implementations, it is possible to find the address of a stack in memory and manipulate the stack as an array of cells. This technique is not portable, however. On some systems, especially Forth-in-hardware systems, the stacks might be in a part of memory that can't be addressed by the program or might not be in memory at all. Forth's parameter and return stacks must be treated as stacks.

A Standard Program may use the return stack directly only for temporarily storing values. Every value examined or removed from the return stack using R@, R>, or 2R> must have been put on the stack explicitly using >R or 2>R. Even this must be done carefully since the system may use the return stack to hold return addresses and loop-control parameters. Section 3.2.3.3 Return stack of the Standard has a list of restrictions.

E.5 ROMed application disciplines and conventions

When a Standard System provides a data space which is uniformly readable and writeable we may term this environment **RAM-only**.

Programs designed for ROMed application must divide data space into at least two parts: a writeable and readable uninitialized part, called **RAM**, and a read-only initialized part, called **ROM**. A third possibility, a writeable and readable initialized part, normally called **initialized RAM**, is not addressed by this discipline. 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.

The words affected by conversion from a RAM-only to a mixed RAM and ROM environment are:

, (comma) ALIGN ALIGNED ALLOT C, CREATE HERE UNUSED

(VARIABLE always accesses the RAM data space.)

With the exception of , (comma) and C, these words are meaningful in both RAM and ROM data space.

To select the data space, these words could be preceded by selectors RAM and ROM. For example:

```
ROM  CREATE ONES  32 ALLOT  ONES 32 1 FILL  RAM
```

would create a table of ones in the ROM data space. The storage of data into RAM data space when generating a program for ROM would be an ambiguous condition.

A straightforward implementation of these selectors would maintain separate address counters for each space. A counter value would be returned by HERE and altered by , (comma), C,, ALIGN, and ALLOT, with RAM and ROM simply selecting the appropriate address counter. This technique could be extended to additional partitions of the data space.

E.6 Summary

The ANS Forth Standard cannot and should not force anyone to write a portable program. In situations where performance is paramount, the programmer is encouraged to use every trick in the book. On the other hand, if portability to a wide variety of systems is needed, ANS Forth provides the tools to accomplish this. There is probably no such thing as a completely portable program. A programmer, using this guide, should intelligently weigh the tradeoffs of providing portability to specific machines. For example, machines that use sign-magnitude numbers are rare and probably don't deserve much thought. But, systems with different cell sizes will certainly be encountered and should be provided for. In general, making a program portable clarifies both the programmer's thinking process and the final program.

 Table of Contents
 Next Section