A.X. The optional Cross-compiler word set

A.x.1 Introduction

This optional wordset is based upon standards jointly developed and used by two independent suppliers of commercial Forth cross-compilers, Microprocessor Engineering Ltd. (MPE) and FORTH, Inc. This technology has been in field use since 1996.

As noted in **x.1 Introduction**, use of this wordset can rarely ensure perfect portability between different targets, but it can make it easier by standardizing notation for common functions.

It is not intended (nor considered desirable) for a Forth system to employ or provide this wordset unless it is designed specifically as a cross-compiler.

Similarly, it is not intended (nor considered desirable) that a program employ this wordset unless it is specifically designed to be run in a cross-compiled environment. It is our hope that modifying a program intended for a standard Forth system to be cross-compiled will be a straightforward process using this wordset.

Specific issues needing to be addressed by standard programs in order to be cross-compiled include:

- memory sections must be defined as described in **x.3.2.2.1 Defining Sections**.
- compiler directives (**IMMEDIATE** words) must be defined in **COMPILER** scope. It's advisable to group any such words for convenience.
- application-specific data types (if any) must be defined in **INTERPRETER** scope.
- the program must never attempt to store values into a **VARIABLE** or **2VARIABLE** or otherwise attempt to write to UData at compile time (e.g., in source code).
- the program must never attempt to execute a target colon definition at compile time (e.g., in source code).
- the program must not attempt to execute at compile time (e.g., in source code) target-specific behavior of application-specific data types (other than simple return of the data field address) unless such behavior is also defined for **INTERPRETER** scope.

We note that many cross compilers now available, including those from MPE and FORTH, Inc., provide features considerably beyond the minimum requirements specified herein. It is the intention of this standard to establish a basic set of entitlements that developers of cross-compiled applications may expect, and which implementors can readily support.

A.x.2 Additional terms and notation

A.x.2.1 Definitions of terms

compiling words: The intent is to include words such as **,** (comma), **ALLOT**, and compiler directives such as **IF** that modify the content of definitions. Note that its logical companion, "defining words," is already defined in **2.1 Definitions of terms**.

searching words: This category includes such words as **FIND**, **'**, and **[']**.

The reason for defining these categories of words is to specify that they are required on the host but not the target.

A.x.3 Additional usage requirements

A.x.3.1 Scopes

A scope is not a search order, although search orders are commonly used to implement scopes. When implemented using the Search Order wordset, scopes normally specify a search order, as well as the compilation wordlist. Special words within some scopes, such as **INTERPRETER**'s **:** (colon) and **COMPILER**'s **DOES>** and **;** (semicolon) may set different search orders. Possible implementation examples are given in Table 1 and with each scope below.

Table 1: Example of search orders in scopes

Scope	Compilation wordlist	Interpreting search order	Compiling search order
HOST	**FORTH**	**FORTH**	**FORTH**
INTERPRETER	***INTERPRETER**	**FORTH** ***INTERPRETER**	***INTERPRETER** **FORTH**
COMPILER	***COMPILER**	**FORTH**	**FORTH**
TARGET	***TARGET**	***INTERPRETER** ***TARGET**	***COMPILER** ***TARGET**

A.x.3.1.1 HOST Scope

When implemented using the Search Order wordset, a typical search order might consist of **FORTH** (only) with a compilation wordlist of **FORTH**. **HOST** scope is used extremely rarely, for custom additions to a cross-compiler.

A.x.3.1.2 INTERPRETER Scope

When implemented using the Search Order wordset, a typical search order might consist of ***INTERPRETER**, **FORTH** (where ***INTERPRETER** is the wordlist containing the words peculiar to this scope) with a compilation wordlist of ***INTERPRETER**.

Standard data objects defined in **TARGET** scope are also available in the ***INTERPRETER** wordlist. This enables objects defined using standard data defining words to be referenced interpretively as well as inside **TARGET** definitions, as required. This feature is necessary in order that, for example, variables may be referenced in assembler code. Note, however, that the entitlement of invoking a target data object in **INTERPRETER** scope is limited to the standard words that return the address of the object's target data space; if you have defined a special target behavior for a class of data objects the special behavior is not available interpretively on the host. A possible workaround for this would be to define a synonym in **INTERPRETER** scope that emulates the special behavior.

The **INTERPRETER** version of **'** is restricted to searching **TARGET** definitions only. This is important so that, for example, you can assemble a branch to a target word. If you need a search-order-dependent version, you must use the **HOST '** .

INTERPRETER scope is used commonly for defining **SECTION**s and **EQU**s that will control the compilation of the target program, for defining custom **TARGET** defining words, and for other target configuration actions.

A.x.3.1.3 COMPILER Scope

When implemented using the Search Order wordset, a typical search order might consist of **FORTH** (only) with a compilation wordlist of ***COMPILER** (where ***COMPILER** is the wordlist containing the words peculiar to this scope). This has the effect of making the host words available to construct the cross-compiler's compiler directives.

The action of the **INTERPRETER** version of **:** (which constructs a **TARGET** definition) might be to set the search order to ***TARGET**, ***COMPILER**. This has the effect of making the **COMPILER** words (as well as references to other target words) visible inside the target definition without having to be **IMMEDIATE**. The **COMPILER** version of **;** would reset the search order to its normal state in **TARGET** scope.

It is also common for the **COMPILER** version of **DOES>** to set a search order of ***TARGET**, ***COMPILER**, because the words following **DOES>** must be target words.

The word **IMMEDIATE** (6.1.1710) is inappropriate in this context. The reason is that it is used after a word is defined, and the number of things that may need to be changed is too great for an after-the-fact fixup to be practical.

COMPILER scope is used rarely in applications, for defining custom target compiler directives.

A.x.3.1.4 TARGET Scope

TARGET scope is where virtually all application programming takes place.

When implemented using the Search Order wordset, a typical search order might consist of ***INTERPRETER** (only) with a compilation wordlist of ***TARGET** (where ***TARGET** is the wordlist containing the words peculiar to this scope).

It may seem surprising that **TARGET** words aren't available in **TARGET** scope. The reason is that words are being executed, and **TARGET** words can't be executed on the host! Many cross compilers support an interactive testing environment in which **TARGET** words may be typed on the host and executed by a simulator or target device connected to the host, but this standard does not require or assume such a capability.

A.x.3.2 Data space management

A cross compiler is concerned with managing target data space. This standard provides no access to host data space except through the **HOST** memory reference words. During the compiling process, target address space used for CData and IData sections is mapped to host physical resources (e.g., memory or disk) by implementation-dependent mechanisms, however this is intended to be transparent to the application programmer.

CData was originally conceived as "code space" used by the cross-compiler to build its program image, and that is still its primary purpose. It is also a useful place to put read-only data (e.g., tables of coefficients for a function). Since embedded targets commonly run from PROM or flash (or use PROM or flash to store a program image which is transparently run from RAM) it is convenient for the compiler to segregate code and data space.

For many reasons, it is not possible for applications to portably manipulate code space, so ANS Forth has always confined program access to data space (cf. **3.3.3 Data Space**). Therefore, the only practical program use for CData is as "constant data" storage. Nonetheless, it is well to remember that the cross compiler will very likely be placing program information there. Note especially the fact that **x.3.3.3.2 Contiguous regions** terminates a contiguous region by "defining the next object in that section type" — that "next object" isn't necessarily a *data* object, but may be an executable definition if the section type is CData.

The fact that this standard makes provision for defining code and data spaces separately doesn't necessarily imply that they have to be separate. Note that **x.3.2.2 Memory Sections** requires that sections of the same type must not overlap, but there is no such prohibition about sections of different types. In theory, if you wish all your memory integrated, you could define all three sections with the same address parameters. However, this raises the question about the compiling and allocation words **ALLOT**, **,** etc. On an integrated-memory system, they must *not* behave separately, and initialization of IData becomes complicated.

In practice, it is far more convenient to segregate these memory spaces. However, it is not uncommon to see a test environment for a ROM-based target to define a CData space in a chunk of UData to provide for test definitions in RAM.

Special issues arise in the case of "Harvard architecture" processors (which enforce run-time segregation of code and data spaces). They may, for example, allow parallel address spaces (e.g., 0-64K code space, 0-64K data space). Some even have different cell sizes for code and data space. Since a standard application is not permitted to read or write actual code, differing cell sizes can be reasonably well concealed within the cross compiler (or visible only in platform-dependent code). However, the potential for parallel address spaces has required the addition of the words **@C**, **C@C**, and **CMOVEC** for explicitly accessing code space. These words are useful for managing CData at compile time, and are necessary for reading code space in a Harvard target at run time in the target program.

On Harvard architecture processors that rigorously enforce the code/data segregation, it may be necessary to define separate CData sections for code and constant data. In most compilers the user controls the disposition of multiple CData sections, so they may be installed appropriately in the target.

A.x.3.2.1 Types of memory

Note that the table listing the locations of standard data objects omits **CONSTANT**s and **2CONSTANT**s. The reason for this is that since their data space may not be referenced nor their values changed, there is no need to specify where they reside. They may reside in CData or IData, or they may cause literal values to be compiled (in which case they don't exist as entities in the target at all).

The host's image of IData and CData may be read and written by the host system, but UData may not. We are aware that many cross-compilers provide interactive access to a target under test, and under this circumstance UData may be read and written. However, although this is a convenient and desirable feature, this standard does not require it, and standard programs may not assume such a capability.

The availability of the words that select memory types provides total flexibility for user-defined defining words to specify where their data space(s) may go. For example, here is an example of an array that keeps its parameters in IData and its payload in UData:

```
: BIGARRAY ( n -- )        \ Defines an array of n bytes in UData
      CREATE DUP ,         \ Save size
      UDATA HERE  IDATA ,  \ Save payload address
      UDATA ALLOT          \ Reserve n bytes in UData
   DOES> ( n – addr )      \ Takes an index and returns nth addr
      2@  ROT MIN  + ;     \ Return addr, clipped to size
```

This type of strategy is preferred when the payload is large, because placing it in IData would enlarge the initialization table unnecessarily.

Note that **HERE** and **ALLOT** are working in the current section of the current section type. It is neither necessary nor appropriate to specify section type in the **DOES>** portion, because when this is executed the target program is dealing with actual addresses, mapped to the right places. Section manipulation is a compile-time issue.

Note that the **DOES>** is followed by target code, which may not be executed on the host. If you wish **INTERPRETER** access to the IData space of **BIGARRAY** on the host, you may get it with the phrase:

```
      ' BIGARRAY >BODY
```

It is possible to place data structures in CData, but only if they are regarded as constant (read-only) data by the target program at run-time. That is, you may say:

```
CDATA  CREATE FOO  10000 ,   IDATA
```

…and later say **FOO @** in the target program, but **FOO**'s data space is read-only to the target.

A.x.3.2.2 Memory sections

Since it is highly likely that two different platforms will have different physical memory configurations, it is reasonable to ask whether this is something worth standardizing. We believe that, even though it may be necessary to re-map the specific addresses in one or more memory sections when porting a program to a new platform, it is a valuable contribution to portability to standardize the specification mechanism and syntax. In most cases it should be possible to accommodate a different memory map by simply adjusting the specific address arguments in the section definitions, with no other changes being necessary.

In many, perhaps most, instances there will be only three defined sections, one of each type. Multiple sections of the same type may be used to, for example, place frequently-accessed data in on-chip RAM or to manage a discontinuous target memory map.

A.x.3.2.2.1 Defining Sections

As an example, consider the following configuration of a program that runs from PROM on a platform with RAM at addresses 800-BFF_H and PROM from 8000-$FFFF_H$. It's configured with the following sections:

```
INTERPRETER   HEX
0800 08FF IDATA SECTION IRAM        \ Initialized data
0900 0BFF UDATA SECTION URAM        \ Uninitialized data
8000 FFFF CDATA SECTION PROGRAM     \ Program in external ROM
```

Note that you must be in **INTERPRETER** scope in order to define your sections.

CData is commonly used for storing executable code, a program image, and the initialization information for IData. However, just as elsewhere ANS Forth allows access only to application-defined data space, users of cross-compilers are similarly allowed access only to data objects defined by the application in CData.

Since both CData and IData are initialized at compile-time, the decision where to place an initialized data object should be primarily based on whether the program needs to write to the object, since CData is read-only. Writable data objects that do not require initialization should be placed in UData, since doing so saves space that may otherwise be required for initialization.

Multiple sections of the same type are helpful when your memory map isn't contiguous, or when you want to be able to manage internal (on-board) RAM specially, for example, by putting your stacks and frequently-accessed system variables there. Except in situations where there are multiple sections of each type, the section names are rarely used. The section type specifiers (**IDATA**, etc.) are primarily used to control where the data space of new data objects being defined will be.

A.x.3.2.2.2 Memory Access

The memory access words that are executable while interpreting in **TARGET** scope must be defined in **INTERPRETING** scope.

Some targets allow access to CData at run-time, and cross-compilers supporting such targets may provide storing equivalents of **C@C** and **@C**, but such access is an environmental dependency.

a.x.3.3.3.2 Contiguous regions

The reason that contiguity of space allocated by **BUFFER:** and **RESERVE** is not guaranteed is to enable cross compilers to allocate UData in a top-down fashion if they so desire.

Note the restriction that a contiguous region is "terminated by defining the next object in that section type." In the case of CData, this may include executable definitions as well as data objects.

A.x.3.4 Effects of scopes on data object defining words

Some special issues arise when creating custom data objects in a cross-compiled environment: defining words are executed on the *host*, to create new definitions that can be executed on the *target*. Therefore, you must be in the **INTERPRETER** scope (see **x.3.1.2 INTERPRETER Scope**) when you create a custom defining word, and you must be aware of what data space you are accessing in the new data object.

Consider this example:

```
INTERPRETER
\ PRINTS defines words that display their values.

: PRINTS ( n -- )
   CREATE ,       \ New definition with value n.
   DOES> ( -- )   \ Execution behavior.
         @ . ;    \ Fetch value and display it.

TARGET

1 PRINTS ONE
2 PRINTS TWO
```

ONE and **TWO** are target definitions, *instances* constructed by the defining word **PRINTS**. Each instance has its own value, but all objects defined by **PRINTS** share the run-time behavior (**@** and **.**) associated with **PRINTS**.

You must specify **INTERPRETER** before you make the new defining word, and then return to **TARGET** to use this word to add definitions to the target. The **INTERPRETER** version of **DOES>** allows you to reference **TARGET** words in the execution behavior of the word, since that will be executed only on the target.

When **CREATE** (as well as the other memory allocation words listed in Section 3.3) is executed to create the new data object, it uses the *current section type*. We recommend adopting the policy that the default is IData. The defining words that explicitly use UData (**VARIABLE**, **BUFFER:**, etc.) do not affect the current section type. If you wish to force a different section type, you may do so by invoking one of the selector words (**CDATA**, **IDATA**, or **UDATA**) inside the defining portion or before the defining word is used. If you do this, however, you should assume responsibility for re-asserting the default section.

You can control where individual instances of **PRINTS** definitions go, like this:

```
CDATA
1 PRINTS ONE

IDATA
2 PRINTS TWO
```

In this case, the data space for **ONE** is in code space, but the data space for **TWO** is in initialized data space. (Not all processors support data objects in code spaces, so **TWO** is portable and **ONE** is not.)

Alternatively, assuming your processor permits it and you are willing to assume the appropriate dependencies, you could define **PRINTS** to explicitly assert CData:

```
: PRINTS ( n -- )
   CDATA          \ Select code section.
   CREATE ,       \ New definition with value n.
   IDATA          \ Restore default iData section.
   DOES> ( -- )   \ Target execution behavior.
         @ . ;    \ Fetch value and display it.
```

In this case, both the **CREATE** and the **,** (comma) will use CData.

A.x.3.5 Ambiguous conditions

A.x.4 Additional documentation requirements

A.x.4.1 System documentation

A.x.4.1.1 Implementation-defined options

Separate documentation requirements are given for host and target. In general, specification of features such as cell and character size, arithmetic and number representation, alignment requirements, etc., are of more interest with respect to the target than the host, because it is assumed that it is the target you are programming. Cell sizes of host and target may well differ; it is common, for example, to use a 32-bit Forth to support a cross-compiler for 16-bit target implementations. This will present a challenge for the cross-compiler implementor, but should be transparent to a program being written to run on the 16-bit target.

A.x.4.1.2 Ambiguous conditions

A.x.4.1.3 Other system documentation

A.x.4.2 Program documentation

A.x.5 Compliance and labeling

A.x.5.1 ANS Forth cross-compilers

The intent of this section is to make it clear that a “standard cross-compiler” is a programming environment comprised of the host and target together. It is expected that the primary responsibility for compiling falls on the host; therefore, the host is required to provide all the compiler support words. The target is not required to provide a compiler or searchable dictionary (though it may do so), but it is required to support all other CORE words.

A.x.5.2 ANS Forth programs

A.x.6 Glossary

A.x.6.1 Cross-compiler words

A.x.6.1.1250 DOES> "does" CROSS

This version of **DOES>** must be defined in **COMPILER** scope, for use in **INTERPRETER** definitions of target data objects. The **COMPILER** version of **DOES>** must make the necessary adjustments to search order or other implementation strategy to ensure that the run-time portion of the definition is executable on the target.

x.6.1.nnnn SECTION CROSS

Recommended usage is to specify the section type immediately before **SECTION**. For example:

```
0800 08FF IDATA SECTION IRAM        \ Initialized data
```

defines a region of IData named **IRAM** whose address range is 0800-08FF$_H$.

Note that use of a section *name* only changes the current section of *name*'s type, it doesn't change the current section type. That is only done by the type name (**IDATA**, etc.). If there is only one defined section of each section type, the section names are never actually required.

x.6.1.nnnn VARIABLES CROSS

Usage: <section-type> **VARIABLES**

Some cross-compilers have traditionally placed **VARIABLE**s in uninitialized RAM, whereas others have placed them in initialized data space, allowing for possible initialization. Addition of this configuration option allows programs to select an appropriate behavior.

Programs wishing to minimize the size of the target image may prefer to use UData for **VARIABLE**s, and use **VALUE** or a form such as:

```
CREATE <name>  <n> ,
```

for read/write data objects that need initial values.

A.x.6.2 Cross-compiler extension words

The words added here have been found to be extremely useful in cross-compiling environments.

A.x.6.2.nnnn @C "fetch-c" CROSS EXT

This word is intended to provide access to code space in Harvard architecture targets. In such targets, the addresses for CData and IData or UData may overlap, and code space is not readable without special action. In non-Harvard targets, **@C** is equivalent to **@** in target programs.

A.x.6.2.nnnn **BUFFER:** CROSS EXT

This word is taken from IEEE 1275, Open Firmware.

A.x.6.2.nnnn **C@C** “c-fetch-c” CROSS EXT

This word is intended to provide access to code space in Harvard architecture targets. In such targets, the addresses for CData and IData or UData may overlap, and code space is not readable without special action. In non-Harvard targets, **C@C** is equivalent to **C@** in target programs.

x.6.2.nnnn **CMOVEC** “c-move-c” CROSS EXT

This word is intended to provide access to code space in Harvard architecture targets. In such targets, the addresses for CData and IData or UData may overlap, and code space is not readable without special action. In non-Harvard targets, **CMOVEC** is equivalent to **CMOVE** in target programs.

x.6.2.nnnn **CVARIABLE** CROSS EXT

This word is especially useful in small embedded systems, as it allows the definition of single-character data objects, thus saving RAM.