Toward a Standard for Cross-compilers and Embedded Systems

Elizabeth D. Rather
FORTH, Inc.
111 N. Sepulveda Blvd., Suite 300
Manhattan Beach, CA 90266

ABSTRACT

One of the agenda items for ANS Forth involves addressing the issues raised by embedded systems and cross-compilers. This is worthwhile, as such systems represent a large body of Forth usage.

In 1996, FORTH, Inc. and MPE developed a joint set of standards for such systems. These have been used in the Europay project to program eight smart card terminals with three different kinds of CPUs ranging from 8051s to 68Ks, and are incorporated in our SwiftX cross-compilers which are now used by over fifty different customers on six different processor families. This represents a good body of experience to form the basis for a proposal.

1. Issues

The following issues need to be addressed:

- **What needs to be in the target?** On many embedded systems it's inappropriate to have a full dictionary, heads, compiler, interpreter, etc., resident in the target. Is it an "ANS Forth System" if the *combination* of host and target provide all CORE words?
- **What about managing memory spaces?** Presently, ANS Forth's "dictionary" only contains data, and the rules for pre-initializing data spaces are unclear. Embedded systems have to worry about ROM and RAM, on-board and external memory, etc.
- **How about managing scope/vocabulary issues?** If the cross-compiler itself is written in Forth, as many are, how do you distinguish the underlying system's Forth words from the versions that construct the target, or are only executable in the target?

2. Host and Target Roles and Functions

ANS Forth contains two recognizable sets of functionality:

1. Words that build and manage definitions and data structures, and

2. All other executable words.

In a cross-development environment, the first set may be confined to the host, so I will call these *Host* functions. The second set, normally built by the first set, I shall call *Target* functions.

Host functions include all defining words, "syntactic elements" such as **IF** and **DO**, words that put things in data structures such as **,** (comma), and **DOES>**. Target functions include normally executable words like **+** and **DUP**.

A conventional Forth integrates these two. A cross-development system segregates them, and manages them quite distinctly. There may be versions of target words that are executable on the host as well as the target.

I propose to introduce a new optional wordset for cross-compiling. It will begin by identifying which of the present ANS Forth words (in all wordsets) fall into which category. It will then establish the principle that an "ANS Forth system" exists if, during development, the full set is available even though the host functions may be on a separate computer from the target. The target is not *required* to provide host functionality, although it may do so.

3. Managing Scopes

A "scope" may be defined as the logical space in which a word is visible or can operate. In this context, the host and target systems require separate scopes, to distinguish (for example) the **DUP** that is used in the host computer's underlying Forth from the one that is executable only on the target, and the **:** used to build cross-compiler functions from the one that builds target definitions.

I propose to define the following scopes:

- **HOST**: This provides access to the underlying system's Forth, and is used to construct the cross-compiler. It's rarely used explicitly in programs built for the target, but is available in case the programmer needs to do something special.
- **INTERPRETER**: These words are executed on the host to construct and manage target definitions and data structures, and include all defining words plus words such as **,** (comma). New, application-specific, defining words are defined in **INTERPRETER** scope.
- **COMPILER**: This is used to make words executed inside **TARGET** definitions to construct structures, etc.
- **TARGET**: This is the default scope, which contains all words executable in the target (and are *not* guaranteed to be executable on the host).

By default, new commands belong to the **TARGET** scope; i.e., they are compiled onto the target. But after the **INTERPRETER** command, new words are added to the host that will be found when the host is interpreting on behalf of the target.

If you use any of these *scope selectors* to change the default scope, we recommend that you later use **TARGET** before commands can again be compiled to the target.

The compiler directive in force *at the time you create* a new colon definition is the scope in which the new word will be found. As a trivial example:

```
TARGET  ok
```

```
: Test1 1 . ;  ok
Test1 1  ok

INTERPRETER  ok
Test1
Error 0  TEST1 is undefined
Ok
```

The table below summarizes the availability of words defined in various scopes.

If defined in:	Available in these scopes while interpreting:	compiling:
COMPILER	Not allowed	**TARGET**
HOST	**HOST**, **INTERPRETER**, **COMPILER**	**HOST**, **INTERPRETER**, **COMPILER**
INTERPRETER	**TARGET**	**INTERPRETER**
TARGET	Not allowed	**TARGET**

Scopes may be defined using wordlists and search orders, although they may also be defined using non-ANS Forth techniques providing the correct functionality is supported.

4. Data Space Management

Target memory space can be divided into multiple *sections* of three types, shown in the table below. Managing these spaces separately provides an extra measure of flexibility and control, even when the target processor does not distinguish code space from data space.

Type	Description
CDATA	Code space; includes all code plus initialization tables. May be in PROM. **CDATA** may not be accessed directly by standard programs.
IDATA	Initialized data space; contains preset values specified at compile time and instantiated in the target automatically as part of power-up initialization. It is writable at run-time, though, so it must be in RAM.
UDATA	Uninitialized RAM data space, allocated at compile time. Its contents cannot be specified at compile time.

At least one *instance* of each section must be defined, with upper and lower address boundaries, before it is used. Address ranges for instances of the same section type may not overlap. The syntax for defining a memory section is:

```
<low address> <high address> <type> SECTION <name>
```

An instance becomes the *current section* of its type when its name is invoked. The compiler will work with that section as long as it is current, maintaining a set of allocation pointers for each section of each type. Only one section of each type is current at any time.

As an example, consider the following configuration of a program that runs from PROM. 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
```

4.1. Vectored Words

The words used to allocate and access memory are vectored to operate on the current section of the current type. Use of one of the section type selectors **CDATA**, **IDATA**, or **UDATA**, sets the vectors for the vectored words. If you only have one section of each type, the section names are rarely used; however, if you have (for example) multiple **IDATA** sections, using the name specifies where the next data object to be defined will go. Multiple sections of a given type enable you to specify onboard and external RAM, for example, or handle non-contiguous memory maps.

The vectored words are:

ORG *(addr —)*
Set the address of the next available location in the current section of the current section type.

HERE *(— addr)*
Return the address of the next available location in the current section of the current section type.

ALLOT *(n —)*
Allocate *n* bytes at the next available location in the current section of the current section type.

ALIGN *(—)*
Force the space allocation pointer for the current section of the current section type to be cell-aligned.

C, *(b —)*
Compile *b* at next available location (**CDATA** and **IDATA** only).

, *(x —)*
Compile a cell at the next available location (**CDATA** and **IDATA** only).

4.2. Data Types

Target defining words may place their executable components in code space. Data-defining words such as **CREATE**—and custom defining words based on **CREATE**—make definitions that reference the section that is current when **CREATE** is executed.

Because **UDATA** is only *allocated* at compile time, there is no compiler access to it. **UDATA** is allocated by the defining words themselves; a summary of defining words is given below. At power-up, **UDATA** is uninitialized.

VALUEs must be in **CDATA**, because they are initialized. We define **VARIABLE**s to be in **UDATA**, and will recommend that that be the default. We don't specify where **CONSTANT**s go, because some compilers compile references to **CONSTANT**s as literals; for that reason, we would retain the restriction that they cannot be changed, and will not specify where they go.

The **@** and **!** words, as well as the string initialization words **FILL**, etc., may be used at compile time providing the destination address is in **IDATA**. It's an ambiguous condition (our compilers will abort) to attempt to access **UDATA** other than from the target at run time.

4.3. Effects of Scoping on Data Object Defining Words

Defining words other than **:** (colon) are used to build data structures with characteristic behaviors. Normally, an application programmer is primarily concerned with building data structures for the target system; therefore, the dominant use of defining words is in the **TARGET** scope while in interpreting state. You may also build data objects in **HOST** that may be used in all scopes *except* **TARGET**; such objects might, for example, be used to control the compiling process.

Data objects fall into three classes:

IDATA *objects* in initialized data memory—e.g., words defined by **CREATE**, **VALUE**, etc., including most user-defined words made with **CREATE … DOES>**.

UDATA *objects* in uninitialized data memory—e.g., words defined by the use of **VARIABLE**, **BUFFER:**, etc.

Constants—words defined by **CONSTANT** or **2CONSTANT**.

Unlike target colon definitions, target data objects may be invoked in interpreting state. However, they may not exhibit their defined target behavior, because that is available only in the target (or in interacting state). Constants will always return their value; other words will return the address of their target data space address. **IDATA** objects may be given compiled, initial values with **,** (comma) and **C,** (c-comma), and you may also use **@** and **!** with them at compile time. However, there is no way to initialize **UDATA** objects at compile time.

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 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
\ ARRAY is an array of specified size in UDATA.

: ARRAY ( n -- )
   IDATA CREATE                \ New definition with value n.
      UDATA HERE OVER ALLOT    \ Allocate space, get location
   IDATA ( Loc ) ,  ( Size) ,  \ Save size & location
   DOES> ( i – addr )          \ Take index, return addr of ith
      2@  ROT MIN  +           \ Compute indexed address
;

TARGET

100 ARRAY STUFF
```

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 above) is executed to create the new data object, it uses the *current section type*. The default in our practice is **IDATA**. The defining words that explicitly use **UDATA** (**VARIABLE**, 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 must assume responsibility for re-asserting the default section.

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

```
IDATA
CREATE BYTES   1 C,  2 C,

UDATA
CREATE STUFF   100 ALLOT
```

In this case, the data space for **BYTES** is in initialized data space, but the data space for **STUFF** is in *un*initialized data space.

5. Conclusions

The above is a brief description of technology that has been developed and used by two major vendors of cross-compilers, as well as many of their customers. We believe the rules and side-effects are well-understood. The number of actual new words is small.

The hope is that adding these words can enable implementors to create standard cross-compilers, and application programmers to write standard programs that can be trivially modified to run in either domain, or provide initial stubbs to enable their programs even to run on systems not providing the cross-compiler words.