$10 Volume XX, Number 4 November 1998 December

FORTH DIMENSIONS

Forth and Functional MRI

How and Why to Use Multitasking

DynOOF-style Objects for the i21 microprocessor

Toward a Standard for Cross-compilers and Embedded Systems

Reed-Solomon Error Correction

OFFICE NEWS

As usual, FIG's administrative and sales office is keeping busy with processing your membership renewals, service requests, new memberships, and product sales. Is your membership due for renewal? It helps us if you renew your membership early, and it guarantees that you do not miss any issues of *Forth Dimensions*. When you renew your membership, please be sure *to give us your current e-mail address*. There are some exciting things on the horizon, and if we have your current e-mail address it will make it easy for you to participate! In fact, just e-mail it to us at office@forth.org.

Speaking of participating, please take a look at the back cover of this issue. If you or someone you know is thinking about submitting an article to *Forth Dimensions,* the time is now! *Forth Dimensions* is your group's magazine—to get the most out of it, everyone needs to put something into it! The most often heard comments I get here at the business office, when asking someone to write for *Forth Dimensions,* reveal that most people seem intimidated about needing to have an article in perfect form. Believe me, our editor, Marlin Ouverson, can work magic on prose! If you have an idea for an article and can get that to him, he can help you turn it into a masterpiece.

The 20th Annual FORML Conference was a great success! Over 35 FIG members from the around the world, some coming from England, Germany, and Australia, got together to discuss and share their ideas about Forth and its future. When good people with good minds are working together, really terrific things can happen! We have already started planning for the 21st FORML—if you have ideas or want to get involved, let us know and keep reading Office News for more details.

As we warned in the last issue of *Forth Dimensions*, prices for books and software on the FIG mail-order form will be going up. Because of adjustments to the magazine's publishing schedule, it went to press before the price changes could be made; new prices will appear when the next issue is published.

Looking forward to 1999, we can see it not only as the end of this millennium, but as the threshold to the next one! The future of Forth is in your hands!

Remember together we can make a difference!

Cheers,

Trace Carter
Administrative Manager
Forth Interest Group
100 Dolores Street, Suite 183
Carmel, CA 93923 USA
voice: 831.373.6784 • fax: 831.373.2845
e-mail: office@forth.org

CONTENTS

DEPARTMENTS

EDITORIAL

Preparing for the future

The future of Forth and of the Forth Interest Group is practically a perennial topic of speculation. In the relatively early years of FIG—I was not here at the beginning, so I can't speak for those times—it seemed to me there was some jealousy of the popular language du jour (Pascal, at one point, or even Ada). Those were times when the predictable response to almost any description of another language's strengths elicited a response that was wistful ("But Forth could have done that!") or even belligerent ("Forth has always had that!"). As in so many young organizations, strategic planning took second place to living in the moment while dreaming of hitting it big.

The best way to prepare for the future is to start realistically—from where we are, and not from where we wish we were. That means working with the resources that are available to us, in cooperation with people who follow words with action, to achieve a goal we can agree upon. The *manner* in which we conduct these affairs will be influenced—if we care at all about public perception of the Forth language—by the skill with which we use, or by the ignorance or arrogance that causes us to ignore, contemporary methods of communicating a message and delivering a service or product.

Forth has always thrived by providing improved functionality with constrained resources. In a way, our organization has been doing that. Cleverly done, FIG could make a transition to full on-line presence so thoroughly that much of its day-to-day business could be streamlined. Set-up will be more complex than most volunteer projects, and members might called upon to view change as a positive factor. Some administration would be required, but we can design our services and the methods of delivering them such that their price/performance ratio *makes* them attractive alternatives.

What does the future hold? Forth certainly has a history of finding a niche in which it excels; and while the much-sought killer app has not appeared, there have been many impressive Forth projects over the years, and it is a staple in the back room of many shops. (Maybe we should start thinking of Forth as the *killer tool*.) We can minimally expect that the occasional projects that are not well-served by more-popular approaches will continue to benefit spectacularly from the direct approach and various traits Forth demonstrates.

Some people wait to see what the future holds, others invent it. Whichever approach is taken by this generation of leaders in the Forth community, we should consolidate our current resources and deploy them skillfully to serve Forth users and developers and to demonstrate that we are able to change with the times, and to come out stronger, more organizationally agile and adept with the tools, and perhaps even the styles, of the day.

Measures we can and should be taking now include on-line archiving of all our publications, in facsimile reproductions of presentation quality, in a searchable, indexed, cross-platform format that allows either direct copying or some other simple access to the published code; that might eliminate physical warehousing and much shipping. *Forth Dimensions* should convert to an all-electronic format to capitalize on the conveniences offered by hyperlinks, code sharing, on-line subscription *and* distribution, archiving, and cost savings. The FIG mail-order form could, after the aforementioned electronic archive is complete, be greatly simplified and implemented as a secure-transaction feature of the web site. On-line conferences via IRC or other technology, including thread-linked archives of comp.lang.forth, could replace much of the function lost with the general (though not universal) demise of local FIG chapters. With a commitment to establishing this kind of presence, both the electronic form of this magazine and the overall web site could be redesigned to serve as showcase, reference library, historical archive, code bank, collaborative center, classroom, and technical support.

Of course, the collective resolve it will require to achieve a fair measure of these functions will be greater than it would be if the organization were better endowed. But I think great enthusiasm can overcome many other deficiencies, and our organization has never been short of that. We simply need to consolidate our resources, re-focus on the achievable, and coordinate our efforts.

What will be required to move FIG into a new incarnation that serves better and more economically is resolve, cooperation, patience, and willingness to change. And, of course, labor.

Marlin Ouverson
editor@forth.org

Forth Dimensions
Volume XX, Number 4
November 1998 December

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Trace Carter

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $45 per year ($60 overseas air). For membership, change of address, and to submit items for publication, the address is:

Forth Interest Group
100 Dolores Street, suite 183
Carmel, California 93923
Administrative offices:
831.37.FORTH Fax: 831.373.2845

The Forth Interest Group

The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

FORTH DIMENSIONS (ISSN 0884-0822) is published bimonthly for $45/60 per year by Forth Interest Group at 1340 Munras Avenue, Suite 314, Monterey CA 93940. Periodicals postage rates paid at Monterey CA and at additional mailing offices.

POSTMASTER: Send address changes to FORTH DIMENSIONS, 100 Dolores Street, Suite 183, Carmel CA 93923-8665.

DynOOF-style Objects for the i21 microprocessor

András Zsótér • zsa@itvc.com
Redwood City, California

Since graduating with a Ph.D. from the University of Hong Kong in 1996, the author has been working for the iTV Corporation, on the low-cost internet access product mentioned in this paper.

Introduction

Some say that Forth always had the essentials of object-oriented programming via CREATE and DOES>. However, modern OOP techniques use multiple methods associated with a data structure (as opposed to only one provided via DOES>), late binding, and named data fields inside an object's data area.

Several attempts have been made to add object-oriented features to Forth. Probably one reason that such attempts were so numerous is the relative ease of the implementation of OOP in Forth.

One model developed by the author of this paper places performance above all else [2, 3]. This model will be referred to as the *DynOOF model*, named after one of its implementations called the Dynamic Object-Oriented Forth.

The syntax and semantics of the model is explained in details elsewhere [1]. DynOOF objects are accessed via an *object pointer* (OP), which is usually implemented as a register of the CPU. If resources permit, another register is used as a pointer to the object's *virtual method table* (VMT), which is used at run time for method calls.

Objects must be explicitly selected either by overwriting the OP with O! or by saving the previous OP and then overwriting it with {. In the latter case, the previous object pointer can be restored later by }[1].

After an object has been selected, the programmer can treat fields as ordinary global variables. The method-invocation syntax is also as simple as calling any other Forth word.

The definition of virtual methods, however, requires some extra syntax, but this paper can be understood without it, so it is not presented here [1].

Accessing data inside an object, or calling virtual methods through a late-binding mechanism, can be nearly or equally efficient as accessing ordinary global variables and calling normal Forth colon definitions [3]—providing that the CPU has free registers to hold the Object Pointer and the address of the VMT.

The main goal of this paper is to discuss issues related to implementing the model for the i21 microprocessor in iTvc's Forth target-compiled environment. Because the i21 is a stack machine which, following the Forth tradition, has been designed to meet only the bare minimum needs of the programmer, at first it does not seem to be the best candidate for implementing something like OOP built right into the Forth virtual machine.

The OOP implementation described in this paper demonstrates that it can be done, and that such an implementation is suitable for commercial use.

1. The words { and } have been replaced by <{ and }> in more recent systems, to avoid name collision with the array syntax used by the Forth Scientific Library.

The i21 microprocessor

The i21 microprocessor is a stack machine which is being developed at the iTv Corporation (Redwood City, CA). Its general architecture and instruction set has evolved from the MuP21 microprocessor, which was discussed in detail in the pages of *Forth Dimensions* a few years ago [4].

The i21 is used by iTvc in the design of low-cost, Internet-access devices. The information presented in this paper is in use at iTvc. The object model was the foundation of the implementation of several components of iTvc's system software, including a file system for flash devices, an HTTP client program, and a small HTTP server for embedded systems.

Registers and stacks

Although the i21 is a stack processor, its minimal instruction set and limited stack depth is not enough to support the execution of a wide variety of standard ANS Forth [5] programs.

In order to support all the features in the standard, iTvc's i21 Forth uses "soft" stacks, i.e., the high-level Forth machine does not use the i21's internal stacks as Forth's stacks. Its stacks reside in RAM. The addresses of the memory locations used as stacks in the high-level Forth are kept on the i21's internal stacks.

The following register model was used before implementing OOP:

- The A register: Forth's IP (instruction pointer).
- The top of the data stack: Forth's SP (data stack pointer).
- The top of the return stack: Forth's RP (return stack pointer).
- The second item on the return stack: Forth's UP (pointer to the user variable area).

The above model uses nearly all easily accessible registers of the i21. Items deep in either stack cannot be used efficiently, because even getting to them requires a number of i21 instructions.

The original DynOOF model uses two special registers: One for holding the address of the active object and one for holding the address of the VMT of the class of the active object. [2, 3]. In the i21 Forth, the resources only allowed one such pointer.

The second item on the i21's internal data stack is easily accessible, so it has been chosen to be the OP[2]. This also means that virtual method calls require an additional de-referenc-

2. The register, apart from being easily accessible, also has to be spare, i.e., not clobbered by other pieces of code. Because the i21 registers are arranged as a stack, and routines are not supposed to dig into the stacks and scramble their contents, this condition has been met. Patching a DynOOF-style OOP support into an already existing system on a CPU with a more traditional register file would probably have been a lot more difficult, if not impossible, without rewriting a significant portion of the system.

ing, thus they are somewhat more expensive than in the original DynOOF model. However, field accesses are still fast and, because they are significantly more frequent than virtual method calls, the model still provides sufficient overall performance for real-world applications.

The representation of an object instance

In the original DynOOF model, an object instances is represented on the stack by its address [1, 3]. This is true for objects in dictionary space as well as for objects allocated in dynamic memory.

Because iTvc's Forth system is target compiled, it is impractical to initialize objects at compile time, because the object's `Init` method (its constructor) would have to run on the target compiler's host system whenever an object is initialized at compile time, and the same constructor would have to run on the target system for dynamically created objects.

Therefore, all objects have to be created dynamically. iTvc's operating system (called 4OS) provides two different set of words for memory allocation. One set is handle based—a block of memory is represented on the stack by a handle which is the address of a pointer to the data area in memory. This is to allow moving a memory block without the application code having to keep track of it. The other set of memory-allocation words use addresses, just like the ANS Forth memory-allocation wordset [5].

In iTvc's implementation of the DynOOF model, the former, handle-based, memory allocation scheme was chosen for implementing objects. Objects are represented on the stack as handles. However, this difference is completely transparent as long as the application programmer follows the API described by the appropriate documentation.

Words which create or destroy an object (`New` and `Delete`) operate on handles. Words like `O@`, `O!`, `<(`, and `)>` use and update an *object handle* (OH). The handle of the active object is stored in RAM. Every time OH is updated by one of the above words, the object's address is copied from the handle to the OP inside the CPU.

The only restriction is that the data area of an object cannot be moved in RAM while the object is active (which would render the OP invalid). However, 4OS never actually moves data areas without an explicit request from the application program.

Another difference from the original DynOOF model is in what the address of the object means. In the original model, the address of an object was the address of the first data field, and the object's VMT was stored in the cell immediately before the object's address. In the i21 version, the first cell at the object's address contains the address of its VMT and the first data field starts immediately above it. The old version was designed to protect the programmer, while doing interactive development, from accidentally overwriting the VMT address inside the object instance (the address on the stack is always the address of a data area). On the i21, the address of the VMT is stored in the first cell of the object's body because accessing a negative offset would require additional instructions, which would make method calls slower.

Threaded code for field access and method calls

Both previous implementations of the DynOOF model were native code environments. The Forth system lay down machine code and OOP words usually compiled into one or two machine code instructions [3]. No special effort was necessary to optimize how OOP words are compiled.

In a threaded code model, there are different possible ways to implement field accesses and virtual method calls:

- Each field name and method name can appear in the dictionary as a separate definition. The individual definitions can hold the necessary offsets to access the proper data field or proper VMT entry, respectively. Fields in different classes which have the same offset generate several identical definitions in the dictionary. Also, methods of different classes which have the same offset in the VMT require identical definitions in different classes. In a live Forth system, this solution would be natural. In iTvc's target-compiled environment, the size of the executable is considered to be important; thus, adding identical definitions to the dictionary is not acceptable.

- A preset range of field access and method call definitions can be added to the dictionary, and the compiler chooses the correct token when a field or method name appears in the input.

  Unlike in the previous case, only one token is required for fields starting at a certain offset in different classes. The compiler has to keep track which fields map to which token, and generate new tokens when they are needed.

- The third possibility was the one chosen for iTvc's implementation of OOP because it only requires two definitions: one for field accesses and one for virtual method calls. The same trick is used as in the traditional Forth word `LIT` (which is compiled into definitions by the Forth system when a number inside a definition is encountered).

  Traditionally, the token `LIT` is laid down, followed by the number itself. When this `LIT` token is executed, it fetches the number following it from the dictionary and pushes that number to Forth's stack. The implementation of fields is analogous to `LIT`. When the appropriate token—called `(Field)`—is executed, it fetches the number following it from the dictionary (the field offset), adds that number to the second item on the i21 internal data stack (OP), and pushes the result to Forth's data stack in memory.

Figure One

```
CODE (Field) ( -- Addr )
 OVER     @A+    A@       PUSH    \ Get Offset. Save register A.
 +        OVER   $01 #    nop     \ Calculate Field's Address.
 +        A!     !A       DROP    \ SP+1->A, OP+Offset -> (A).
 A@       POP    A!               \ A->SP, Restore A.
 next
END-CODE
```

For the i21, the implementation of `(Field)` looks like Figure One. For comparison, the `DOVAR` routine which is used in accessing global variables looks like Figure Two.

And in the body of each variable, there is a `call DoVar` instruction followed by the data area of the variable. The same number of instruction words[3] is needed for both a field access and a global variable access.

Thus, a field access inside an object (once the object has been selected) has about the same cost as a global variable access. On the other hand, the same class of objects can have several object instances, while the system has only one set of global variables. This makes objects particularly useful in a multitasking environment (such as iTvc's 4OS), especially because the `USER` variables (which are unique for each task) are more expensive in the current implementation.

The token for methods—called `(Method)`—behaves similarly but, instead of calculating a data address, it fetches the address of a routine from the object's VMT and then executes that routine. (See Figure Three.)

Choosing the third technique has reduced the number of definitions needed for field access and virtual method calls to only two. Also, the compiler has been kept simple. On the other hand, each and every field access and virtual method call requires two items in the dictionary instead of one.

Some optimizations

Using two cells in the dictionary space instead of one is wasteful.

A virtual method is usually called from only a few definitions[4], so the two-item-long calls do not increase the code size significantly.

On the other hand, field accesses are frequent, and a design goal was to keep OOP code small. As shown in the previous section, selecting a way of implementing fields which never, ever wastes dictionary space could prove to be difficult. In order to make field accesses more efficient, the compiler has to recognize word sequences.

Let us consider the following code fragment:

```
Class: Bar Field FBar ;Class

: FOO1 ( -- ) FBar @ Foo FBar ! ;
```

The definition of `FOO1` contains eight tokens: `(Field)`, the offset of `FBar`, `@`, `Foo`, `(Field)`, the offset of `FBar`, `!`, and `EXIT`.

In both cases, when `FBar` is accessed, we are not really interested in its address but in reading or changing its value. Fetching and storing the content of a field is a very frequent operation. Having three tokens for field accesses instead of one can speed up field fetches and stores. We keep `(Field)` for pushing a field's address onto the stack when we need it. Two new tokens are needed—`(Field@)` and `(Field!)`—which work like (Field) followed by a fetch or a store, respectively.

After implementing those tokens in i21 assembly, a modified `@` and `!` has to be implemented which recognizes that a field has just been compiled, and replaces the token `(Field)` with `(Field@)` or `(Field!)`, as appropriate. This modification can be added easily to the target compiler in a couple of lines of code.

After the above modifications, the definition of `FOO1` will contain only six tokens: `(Field@)`, the offset of `FBar`, `Foo`, `(Field!)`, the offset of `FBar`, and `EXIT`.

The latter sequence of tokens is not only shorter than the original one, it also executes significantly faster. Memory accesses are expensive on the i21 and, in the latter case, we avoid storing the address of the field on Forth's stack in memory and then reading it back. Thus, we not only have reduced the number of tokens compiled into definitions, we have also eliminated two memory accesses (which took place behind the scene in the original version) for each field fetch or field store.

(Continues on page 13.)

3. On the i21 microprocessor, just like on the P21, instruction codes are five bits wide. A 20-bit word contains up to four instructions.

4. Because methods can be implemented as ordinary colon definitions [1], only those which have to be overridden in a derived class need to be declared as virtual methods. Thus, the number of virtual methods can always be kept low, and even those can be wrapped into colon definitions, if such a minor difference in code size matters.

Figure Two

```
CODE DoVar ( -- addr )
 01 #     nop     nop     nop     \ + must be in the next word.
 +        A@      OVER    A!      \ SP+1->SP, Save A, SP->A
 POP      !A      A!              \ Return address-?(A), Restore A
 next
END-CODE
```

Figure Three

```
CODE (Method) ( -- )
 A@       PUSH     OVER     A!      \ Save A, OP->A
 @A       @R+      nop      nop     \ Fetch Address of VMT, Fetch Index
 +        A!       @A       POP     \ Fetch (VMT+Index), Restore A
 A!       PUSH     ;                \ Jump to (VMT+Index)
END-CODE
```

Toward a Standard for

Cross-compilers and Embedded Systems

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 that 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` 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.

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
```

Elizabeth D. Rather • erather@forth.com
Manhattan Beach, California

Elizabeth Rather is the President of FORTH, Inc. and the chair of the American National Standards Institute (ANSI) technical committee responsible for the Forth standard.

Table One

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

Table Two

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.

Table One summarizes the availability of words defined in various scopes.

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 Table Two. 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.

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 addr> <high addr> <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 configuration (shown in Listing One) of a program that runs from PROM. It's configured with the sections shown in Listing One.

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 to 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, (char —)
Compile *char* 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 VARIABLEs to be in UDATA, and will recommend that as the default. We don't specify where CONSTANTs go, because some compilers compile references to CONSTANTs as

Listing One

```
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
```

Listing Two

```
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
```

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 some systems, when connected to a target via an interactive link). 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 @, !, MOVE, ERASE, etc., with them at compile time.

There is no way to initialize UDATA objects at compile time. Large buffers and arrays should be placed in UDATA, because IDATA objects enlarge the size of the ROM image by the size of their initialization array.

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 the example in Listing Two.

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 by 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 to ANS Forth can enable implementors to create standard cross-compilers, and application programmers to write standard programs that can be modified trivially to run in either domain, or provide initial stubs to enable their programs to run even on systems not providing the cross-compiler words.

FORML Conference #20, 1998

Although moved to the weekend before rather than after Thanksgiving for the second year in a row, the annual FORML Forth feast remains sumptuous, especially in this, officially its twentieth, incarnation. There were differences other than time: at dinner the first night we had black napkins and three forks; food on the boardwalk between the conference center and the beach has been replaced by some pressed, and formed, and durable look-alike, doubtless good for the environment, and splinter free. But there was also the familiar: a talk by Glen Haydon on Forth Philosophy, another by John Hart on Forth as a hardware design language, several papers and a new lias from Wil Baden, several papers and a lecture by Dr. Ting Chuck's latest versions of his latest things, wine and che e parties, impromptu talks, working groups, wine bottle awa ls, Bob Reiling taking care of us and telling us what to do.

There were a few more of us than last year, and many more pa rs, though too many of those arrived late, hot from expe ence but perhaps short on review. The milestone anniversary brought a few of those who come but not every year: M ch Bradley, Tom Zimmer, Andrew McKewan, and Peter K ggs, but not, for example, Mike Perry, Bill Ragsdale, or R n Braithewaite. There were new faces from Germany and A stralia, as well as some from closer to home. The new faces, in general, sent papers in advance.

The official theme was Forth Interfaces to the World; an unofficial theme was object-oriented Forth. Other topics involved meta- and target-compilation and embedded systems, chip designs, and even some actual applications.

Peter Knaggs, from Bournemouth University in England, presented papers on program verification, software localization, and the future of the ANS Forth standard. His paper on "Typing Forth" is a theoretical discussion of stack analysis that goes beyond counting (stack depth checking) to checking data types. Traditional Forth pretty much lets us do what we want, allowing us to fail with terrifying swiftness, to learn, and move on; more strongly typed languages (like Pascal and, to a lesser extent, C) prevent some of this but at some cost in flexibility and freedom. We know the tradeoffs, and whether Knaggs' direction leads down a slippery slope is a matter of opinion.

Localization involves customizing language and date, numeric, and monetary formats in the user interface of a program to correspond to a user's expectations. This is a serious problem for programs intended to be distributed widely, and one that Americans probably undervalue. One issue for Forth is the use of the phrase `COUNT TYPE` to display strings, since strings of the same meaning in different languages will not always be the same length, nor is there even any guarantee that characters and bytes will always be the same size.

Dr. Knaggs also discussed the ongoing Standards process and the problem of keeping up. This is a two-pronged problem: getting good ideas out there, and consolidating similar good ideas to cut down on the proliferation of dialects. For example, `ONLY` and `ALSO` were presented in an appendix to the 83-Standard, and `CATCH` and `THROW` were presented in a talk at the last session of a long-ago FORML. Both were validated by adoption and eventually became Standard. Knaggs proposes to automate this process through his website (http://dec.bournemouth.ac.uk/forth/ans/).

Useful as it is to all of us, as a point of reference and argument and as a way to share code, the ANS Standard is most useful to those, like FORTH, Inc., that are trying to sell Forth beyond the choir. Thus it is no surprise that Elizabeth Rather presented a report on the required new round of ANS Standard meetings. The issues this round are eliminating obsolescent words from the standard, clarifications (changes in wording but not technical content), and the new issues of internationalization/localization (Knaggs' theme) and embedded systems and cross-compilers. Rather, in another session, presented a proposal for a standard for embedded systems and cross-compilers. The proposal includes a small number of new words and purports to conform to or at least to clarify existing practice by defining four semantic "scopes" (Host, Interpreter, Compiler, and Target) and three kinds of target data space (code, initialized data, and uninitialized data).

Standards or not, target- and meta-compilers have long been a favored FORML theme. This year Andy McKewan discussed optimizing compiler issues with cmForth and ping-pong meta-compilation with riForth; John Rible discussed late-binding in an optimizing ANS Forth target compiler; and Dr. Ting, no friend of the whole idea of meta-compilation, presented some improvements to eForth. Dr. Ting's other presentations involved robotic control (with a discussion of fast square roots), a kind of Forth multi-processing that involved Forth systems sending commands to each other (responding "OK" for success and something else for failure), a discussion of P16 chip architecture, and an unaccountably well-received personality test.

Dr. Ting's robotics paper (there's always a robot at these things lately, even if only in spirit) was one of a relatively small group that focused on actual applications. Others included "Open Network Forth: Control System for the Munich Accelerator Facility," presented by Heinz Schnitter, which included a model of communicating Forth systems running on different processors similar to the one described by Dr. Ting, and a paper on a MIDI controller for a guitar-like instrument, presented by Brad Rodriguez.

The FORML schedule is always pretty much the same: paper presentations Friday afternoon and evening, Saturday morning, and the first session Sunday morning; working groups Saturday afternoon; impromptu talks Saturday

Richard Astle • Del Dios, California
rastle@bigfoot.com

Richard Astle has a bachelors degree in math from Stanford University, a master's in creative writing from San Francisco State, and a Ph.D. in English literature from the University of California (San Diego).

evening; wine and cheese parties both nights; and a final wrap-up in the second Sunday morning session. If the prepared papers (last-minute or not) are an indication of what we're working on, working groups and impromptu talks lean towards what we're interested in.

Like last year, and unlike no previous time in my recollection (but I've been to only fourteen of the twenty FORMLs, and my memory may no longer be perfect), much of the working groups session was taken by a working group of the whole. The first of the whole-group topics this year was the proposed embedded standard. Discussion seemed to focus on two things: whether we need a standard in this area (that old cowboy argument on this new front) and on whether the generated target system would or would not be standard when compiled from a system with or without these proposed standard words, an argument that seemed to miss the point. I have some small experience writing target- and meta-compilers, and the proposal seemed to me merely to be a way to clarify some difficult thinking, and I was surprised at the level of discussion it provoked, the most heated of the weekend—indeed, I believe, of the past several years. Perhaps the dissension had as much to do with history and positions in the Forth universe as anything else: No one wants new requirements for something they already do—that's a strong source of objection to the ANS standard itself—and one can imagine FORTH, Inc. as counterpoint to the freewheeling Forth spirit, where anyone can do anything because they can. Still it was nice to hear *some* argument, if only to remind us.

The second whole-group topic, unfortunately necessary, was the future of FIG, Forth, and FORML, led by Trace Carter, FIG's business manager. FIG membership has dwindled to about 670, but interest in Forth is indicated by the fact that the FIG website gets about 500 hits per day. Proposals for dealing with these facts included charging for downloads and making *Forth Dimensions* an online magazine in an attempt to bring FIG back into the black. Someone suggested a help line: 1-900-FORTHU2. The future of FORML is another issue, and suggestions included moving the site and time. These discussions are ongoing, and will no doubt be reported elsewhere in these pages.

Part of the Working Groups session was reserved for actual working groups: three topics took people to different corners of the room. The discussions this year were on Forth chips, external language interfaces, and (the one I attended) object-oriented programming.

Now that we have an ANS standard, Wil Baden's "if you've seen one Forth you've seen one Forth" has perhaps shifted to "if you've seen one OOP Forth you've seen one OOP Forth." Wil himself presented the first OOP paper of the conference, "Forth SOOP" (the "S" is for "simple"), followed closely by Brad Rodriguez's "Object Oriented Forth and Building Automation Control." Wil's paper attempts to demystify OOP by implementing a version of it simply. We have various models from other languages, ranging (among those I'm familiar with) from C++ (half-hearted OO), through Delphi (Object Pascal) and Java to Smalltalk (where even integers are objects), but for most of us discussions of inheritance, encapsulation, early- and late-binding, polymorphism, message passing, etc., are as in an alien language. As Forth programmers, we tend to try to understand something by implementing it, leading into various dark woods (implementation precedes comprehension). Wil's paper goes a way to dispel the mystery, as does John Carpenter's "OOP in Forth... Using What We Have." Still, we could use a real, thorough, model or two. Zimmer and McKewan's Win32Forth includes an object model (perhaps to be revised on the basis of the discussion in the working group), as does John Sadler's Ficl (rhymes with Tcl), but whether either of these goes far enough to be useful is something to be proven in practice.

That objects are useful has been proven for other languages in the construction and management of relatively large software projects. Sadler's paper, "Ficl — Object Forth Wraps C Structures," demonstrates a technique of using objects in Forth as an "interface to the world," linking the official and unofficial themes of this year's FORML. (Ficl is available from the FIG website, and is also the subject of an article in the January, 1999, *Dr. Dobbs Journal*, our esteemed editor's previous publication.) But Chuck Moore, in an impromptu talk, raised the question of overkill, asking whether objects do anything that Forth doesn't already do, and challenging us to provide useful arguments and examples to justify the overhead. He also challenged the idea of a standard for embedded systems, as he tends to question all standards, saying "FORML would not be FORML without a touch of paranoia," and asking us to be careful not to rule things out but instead to gather them in. At least this is what I have in my notes. When Chuck speaks it is always a bit oracular, coming in at an angle from what the rest of us are thinking, cutting through, clarifying, laying bare.

Other impromptu talks involved Forth chips (Chuck and Dr. Ting), Forth in boot on the new Macs (John Hall), the future of JFAR and EuroFORML (Peter Knaggs), and other things too disparate and numerous to mention.

I've missed a few moments in this thematic traverse, and perhaps what I've mentioned thus far reflects my own interests more than it should. Charles Essen, who came the furthest and talked about "gum trees" on the Stanford campus, gave a paper on implementing temporary dictionaries, implemented by manipulating DP. Temporary dictionaries allow us to use, for example, symbolic names for constants without cluttering the name space. Brad Rodriguez gave a paper on implementing a software UART on the PSC-1000, aka shBoom, a descendant of Chuck's chip of the same name. Though the chip is a Forth chip, he worked this project without Forth. FIG President Skip Carter presented a paper discussing Forth running on an operating system (Linux for example) achieving real-time performance by diving to the micro kernel.

Chuck gave a couple of papers, one on Color Forth, which he's been using for a few years, and another on a new command line, which scrolls right to left on a single screen line, interpreting words as they're typed as in LaForth. Chuck's Forths are always small and idiosyncratic, at right angles to any attempt at standardization or any need for it. About his new command line he said, "There's one other reason for doing this, and that's that it's different, and neat."

Mitch Bradley told a story of Open Boot, Apple, and the seemingly whimsical destruction of the Macintosh clone market; Randy Leberknight echoed this with his experience as a Motorola employee, where he went to write Open Boot software for their Macintosh clone and now finds himself coding in C.

Almost finally, another large topic: Wil Baden's oeuvre: besides his paper on SOOP, which won the prize (a bottle of

(DynOOF, continued from page 7.)

wine) as the "most elegant" presentation, he presented a paper on his technique of mixing source-code and text in his papers and printing them various ways, something we've heard before but not in such detail, and another containing a glossary of all his "tool-belt" words. We all have favorite little extensions to Forth, words not quite in the standard but hopefully definable with it, that we add to whatever system we use and use as our own private primitives: Wil has more of these than most of us, and now we (at least those of us who were there) have a list. Whether any of these will make their way to Knagg's website remains to be seen.

That's about it. All that remains is a summation and the end of the conference. The last session is traditionally a time for a panel discussion, awards, and closing remarks. There was no panel this time, only Dr. Ting and a psychology test that he used to show that FIG and FORML are in trouble because we members and attendees tend to be introverted, intuitive, logical, and relaxed. The awards (bottles of wine) went to Charles Essen, from furthest away (Australia), Wil Baden, for the most elegant presentation (SOOP), Heinz Schnitter, for the largest application, and Brad Rodriguez, for the smallest, and to John Sadler, for yet another freeware Forth (Ficl).

Finally, the sad part of the weekend, which this year was not just the end of the conference but also the end of an era, Bob Reiling stepping down from the directorship after twenty years. There were tributes of course, and the presentation of an engraved cigar box—"finally a place to put all those late papers"—and no doubt he will remain with us, but the conference will run less smoothly without his leadership.

Still, in any end is another beginning, and next year we will be twenty-one.

Conclusion

In this paper, the details and rationale of the implementation of DynOOF-style OOP support for iTvc's i21 Forth have been presented.

It has been demonstrated that, even on machines which do not have many directly addressable registers, such as the i21, a spare register can be found for an object pointer and efficient field accesses can be implemented.

Techniques of implementing field accesses and virtual method calls on a threaded Forth system have been compared. One which generates the smallest number of definitions has been chosen for the implementation.

This paper and the corresponding implementation demonstrates that OOP Forth—even on processors with limited resources, as in a target-compiled, embedded environment—can be very small and fast. In fact, storing data in objects instead of global variables can even be beneficial in a multitasking environment. Thus, programmers who write applications for such systems do not have to avoid using OOP when that is the right solution.

Credits

The software described in this paper is part of the commercial system and application code of the iTv Corporation. Individual employees to be credited include Chuck Moore (who led the i21 microprocessor development), and Jeff Fox and Michael B. Montvelishsky (who wrote the target compiler, high-level Forth environment, and the 4OS operating system).

Further information on 4OS and iTvc software and hardware is available at http://www.itvc.com/ or by e-mail to info@itvc.com.

References

1. Zsótér, A. "An Assembly Programmer's Approach to Object-Oriented Forth," *Forth Dimensions* XVI.6, pp. 11–17 (1995).
2. Zsótér, A. "Implementation of Object-Oriented Programming via Register-based Pointer," *Journal of Microcomputer Applications* 18, pp. 279–285 (1995).
3. Zsótér, A. "Does Late Binding Have to Be Slow?" *Forth Dimensions* XVIII.1, pp. 11–17 (1996).
4. C.H. Ting, C. Moore, "MuP21 — a MISC Processor," *Forth Dimensions* XVI.6, pp. 41–43,10. (1995).
5. "Draft Proposed American National Standard for Information Systems — Programming Languages — Forth," American National Standards Institute, Inc.

Simple Object-Oriented Programming

Programming with objects is like working with trained animals, instead of pushing around boxes with a broom.

—INGALLS per HORCH

SOOP was inspired by a comp.lang.forth article from HELGE HORCH. He used a wordlist for each class and subclass.

As wordlists may be a scarce resource, for FORML 1998 I used trees of linked lists to mimic wordlists. **:NONAME** was used to build the fields of a class: *variables, buffers, constants,* and *operations.*

My goal was to keep it simple: easy to write, easy to read, and fast.

The programmer's words were:

```
CLASS classname
VARIABLE varname
n  BUFFER: buffername
n  CONSTANT constantname
: operationname ... ;
END-CLASS

class  SUBCLASS subclassname

class  BUILDS objectname

Within an operation:
    SUPER field
    COMMON forthword

object method
```

In object-oriented programming, the operations of a class are called *methods.* In the approach here, the data area specifications are also methods—they give addresses of data areas in the object. Classically, the data areas would be private and would require methods to manipulate them.

The words defined in the definition of a class are sometimes called *fields.*

(**END-CLASS** is a new name for **END**, and **BUILDS** is a new name for **BUILD**.)

My implementation has been greatly improved by RICK VANNORMAN, and I no longer distribute it.

Rick has added within a class:

```
PRIVATE
PUBLIC
class  BUILDS objectname
DEFER: operationname  ... ;
CREATE name
class  BUILDS objectname
n  class  BUILDS[ ]  objectname
```

It is still easy to write, easy to read, and fast, but is much more powerful. He will be presenting it in a forthcoming issue.

Simple Object-Oriented Programming

Here are the concepts of simple object-oriented programming.

A *class* is a defining word for a collection of definitions to be made later. Like Forth **CREATE DOES>** defining words, it needs to build an instance of the definitions before any part can be used in a program. **CREATE DOES>** is limited to one cell or data area and one operation. A class may have many named cells, data areas, and operations.

```
CLASS GREETING
VARIABLE COUNT
: HI  ." Good morning "     1 COUNT +! ;
: BYE ." See you later "    COUNT @ . ;
END-CLASS
```

In this silly example, the class GREETING is a pattern for code with one variable and two operations. It is not yet usable code. To create usable code, you build an *object.* The object is an *instance* of the class, and building it is *instantiation.*

```
GREETING BUILDS JOHN
```

Now we have as usable one variable and two operations. To use them outside of the class you must precede them with the name of the object.

```
JOHN HI
0 JOHN COUNT !
JOHN BYE
```

We can build other objects of the same class. Here's JACK.

```
GREETING BUILDS JACK
```

Another variable and two more operations are now available. The variables have different data space; the operations have the same definitions and code space, but work on different data areas. JOHN HI will increment one variable, and JACK HI will increment the other. One variable is JOHN COUNT, and the other is JACK COUNT.

You can see that a class is an expansion of the **CREATE DOES>** idea. Any initialization of variables and buffers has to be done after the instantiation. It may be useful to provide an operation, typically named **INIT**, to do extensive initialization.

In a class, methods can be defined with **VARIABLE**,

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

All code was checked in Power MacForth and SwiftForth. WIL BADEN, after many years of profane language, has retired to Standard Forth. For the source code, send e-mail requesting Stretching Forth #23:SOOP.

BUFFER:, **CONSTANT**, and **:**. Variables and buffers will be initialized to all 0 when the object is built. You can make use of this in your programming. **VARIABLE** and **BUFFER:** definitions cannot be initialized by interpretation in a class definition, and their addresses cannot be used yet—they haven't been built. This is the same as variables and buffers in a cross-development system.

We can make a new class that is an extension of a class and then add members to it with a word that seems to be misnamed: **SUBCLASS**. Think of **SUB** for down and **SUPER** for up in the hierarchy. A subclass may redefine operations and constants that were in its superclass as well as adding new operations, constants, variables, and buffers. A subclass is more specialized than its superclasses.

```
GREETING SUBCLASS SPANISH
: HI  ." Buenos dias "  1 COUNT +! ;
: BYE ." Hasta la vista " SUPER BYE ;

VARIABLE CURSES
: CURSE ." Caramba "  1 CURSES +!  0 COUNT
! ;

END-CLASS

SPANISH BUILDS JUAN
```

SPANISH is a class that is a subclass of GREETING and has redefined HI and BYE. It has new variable CURSES and new operation CURSE.

In a subclass, the operations in the superclass can be used with the names SUPER HI and SUPER BYE. If a name is not duplicated in the subclass, it does not need **SUPER**. The variable COUNT does not need **SUPER** because there is no name conflict in the subclass. **COMMON** before a word will get the Forth word with that name.

In this example, I have used COUNT and BYE as names. These names will be used in the class and its objects with the meaning I've defined. If I want the Standard meaning in the class members, I write **COMMON COUNT** and **COMMON BYE**. (In real life, the use of COUNT would be a poor choice, but I wanted to illustrate this feature.)

Subclasses are classes and, in turn, can be extended by **SUBCLASS**.

There are no rules for naming classes, methods, and objects. You may want to choose some convention of your own use. Suffixing names with : : and : is one way to distinguish related classes and objects. Other suffixes for classes may be used.

existing-object method can occur in a class definition.

Within a class, ***n*** **BUFFER:** ***name*** declares a data area.

Constants known only in a class or its objects can be defined.

Alas, redefining the value of a constant won't change values used before the change.

An important operation of object orientation is to prevent outside access to crucial parts of an object. For this, sections of the class definition can be bracketed between **PRIVATE** and **PUBLIC**. If we change the beginning of GREETING to

```
PRIVATE
VARIABLE COUNT
PUBLIC
```

the contents of COUNT cannot be legally examined or modified outside the object.

Because classes can solve the problem of name conflicts, classes can be written and studied more easily.

A class is (1) a list of its methods, separated into private and public, (2) the total size of data space to be assigned, and (3) a pointer to the superclass.

A member of the list is (1) a representation of the member's name, and (2) its execution token.

An object is an instance of a class. It is (1) a pointer to the class, and (2) the assignment of data space.

Thus, instantiations assign data space, but no new names or execution tokens, for the class members.

Experience

(1) My first use was to handle input files. The class is in the example below. Normally, I just use one input file. I have **FILE**, **REWIND**, and **CLOSE** in class **FILES**, and **MAXLINE**, **LINE**, **OPEN**, **READ-LINE**, **READ**, and **LIST** in subclass **INPUTFILES**.

I define my normal input file as an object.

```
INPUTFILES BUILDS INPUT
```

When I need another file, I define it; for example:

```
INPUTFILES BUILDS MERGE
```

This will give me a complete set of methods for each file

The class **OUTPUTFILES** is another subclass of class **FILES**. It has **OPEN** and **WRITE**. I cannot **READ** or **LIST** an output file nor **WRITE** to an input file. The file knows what file access method to use to open, and whether to use **OPEN-FILE** or **CREATE-FILE**. I define my normal output file when I need it.

```
OUTPUTFILES BUILDS OUTPUT
```

(2) A linked list might have no particular order—that is, be last-in first-out, or first-in first-out. Or it might be in alphabetic sequence. It might be case sensitive or case insensitive. It may be split over several heads. It might be some kind of binary tree. Or, instead of a list, it might be hashed.

For all of these, I want operations **ITEM**, **ADD-ITEM**, **ADD**, **LIST**, and maybe **PRUNE**. The phrase ***str len object*** **ITEM** finds an item.

So I set up classes to define lists. For example, **ORDERED-LISTS**, **ALPHABETIC-LISTS**, **BINARY-TREES**, **HASH-LISTS**, and so on.

(3) To look at random-number generators, I make different classes for different generators, with **INIT**, **RANDOM**, and **CHOOSE** as operations.

Those were applications that happened in the first week after installing **CLASS**. I know there will be more.

Conclusion

SOOP and other approaches to object-oriented Forth unify and supersede application vocabularies, **CREATE DOES>** defining words, and struct definitions. It can also encompass multitasking. Local variables become *instance variables* of an object. The direct act of sorting is usually eliminated.

In constructing a class, the simple names of members make

it easier to code, and the clearer code is more maintainable. Intelligent objects take the place of navigating the structure.

Members of a class are defined with familiar operations: **VARIABLE**, **CONSTANT**, **BUFFER:**, and **:**. The meanings can be understood immediately.

Within similar or different kinds of classes, the names of members can be identical without confusion or conflict.

In a class, the data layout and operations are collected in one place. Operations that don't apply to an object can be kept away from it. Many run-time errors or crashes will be caught at compile time.

The format *object method* is like *structure operation* , and resembles what we normally use. Some other OOFs insist on *method: object* format. This has always seemed backwards to me, as well as requiring a magic character.

Glossary

CLASS
CLASS begins the definition of a defining word for a collection of future definitions.
Used: CLASS *classname*

SUBCLASS
SUBCLASS begins the definition of an extension or specialization of a class.
Used: *class* SUBCLASS *classname*

VARIABLE WITHIN A CLASS
VARIABLE defines a variable member of a class.
Used: VARIABLE *membername*

BUFFER: WITHIN A CLASS
BUFFER: defines a data area member of a class.
Used: *n* BUFFER: *membername*

CONSTANT WITHIN A CLASS
CONSTANT defines a constant member of a class.
Used: *n* CONSTANT *membername*

: WITHIN A CLASS
: starts the definition of an operation member of a class.
Used: : *membername* ... ;

; WITHIN A CLASS
; terminates the definition of an operation member of a class.
Used: : *membername* ... ;

END-CLASS WITHIN A CLASS
END-CLASS terminates the definition of a class.
Used: END-CLASS

BUILDS
BUILDS constructs an object as an instance of a class.
Used: *class* BUILDS *objectname*

COMMON WITHIN A CLASS
COMMON compiles a word from the standard dictionary.
Used: COMMON *forthword*

SUPER WITHIN A CLASS
SUPER compiles a field, beginning lookup for it in the superclass.
Used: SUPER *member*

Files Classes

REWIND rewinds a file.

```
1 : REWIND  ( fid -- )
2     0 0 ROT REPOSITION-FILE ABORT" Can't REWIND "
3     ;
```

FILES is the class for Files.

```
6 CLASS FILES

8     VARIABLE FILE
9     : REWIND  ( -- )  FILE @ COMMON REWIND ;

11    : CLOSE  ( -- )
12        FILE @ ?DUP IF  CLOSE-FILE ABORT" Can't CLOSE-FILE "
13                        0 FILE !
14        THEN
15        ;

17    255 CONSTANT MAXNAME
18    MAXNAME 1+ BUFFER: NAME
19    : .NAME  ( -- )  NAME COUNT TYPE SPACE ;

21 END-CLASS
```

INPUTFILES is the class for Text Input Files.
INPUT is the object for the principal input file.

```
24 FILES SUBCLASS INPUTFILES

26      : OPEN-FILE  ( str len -- )
27          R/O COMMON OPEN-FILE ABORT" Can't OPEN-FILE " FILE !
28      ;

30      : OPEN  ( str len -- )
31          2DUP OPEN-FILE
32          MAXNAME MIN NAME PLACE
33          ;

35      128 CONSTANT MAXLINE
36      MAXLINE 2 + BUFFER: LINE
37      : .LINE  ( str len -- ) ?TYPE CR ;

39      : READ-LINE  ( -- line length more )
40          LINE DUP MAXLINE FILE @ COMMON READ-LINE
41               ABORT" Can't READ-LINE "
42          ;

44      : READ  ( -- false | line length true )
45          READ-LINE ( line length more) DUP 0=
46               IF  NIP NIP ( false) REWIND  THEN
47          ;

49      : LIST  ( -- )
50          REWIND  BEGIN  READ WHILE  .LINE  REPEAT
51          ;

53 END-CLASS

55 INPUTFILES BUILDS INPUT
```

OUTPUTFILES-BIN is the class for Binary Output Files.
OUTPUT is the object for the principal output file.

```
 58 FILES SUBCLASS OUTPUTFILES-BIN

 60      : OPEN-FILE  ( str len -- )
 61          2DUP DELETE-FILE DROP
 62          W/O BIN CREATE-FILE ABORT" Can't CREATE-FILE "
 63          FILE !
 64          ;

 66      : OPEN  ( str len -- )
 67          2DUP OPEN-FILE
 68          MAXNAME MIN NAME PLACE
 69          ;

 71      : WRITE  ( str len -- )
 72          FILE @ WRITE-FILE ABORT" Can't WRITE-FILE "
 73          ;

 75 END-CLASS

 77 OUTPUTFILES-BIN BUILDS OUTPUT
```

How and Why to Use Multitasking

Most Forths provide multitasking, which allows independent threads of control to run cooperatively. I have been using multitasking for some time now, both in my freely available 16-bit Pygmy Forth and in its variants that I use for custom consulting work. I'll discuss some benefits of multitasking in general, and of cooperative multitasking in particular. The examples are written for Pygmy, but the principles apply to other Forths, too.

I seem to see most things these days from the viewpoints of (1) how to build reliable software and (2) how to do it rapidly. I hope to explain how multitasking can contribute to both goals. If you don't already use multitasking, perhaps this article will help break the ice and get you started.

Quick summary of how to use multitasking in Pygmy

- Define the word the task will execute
- Create the task with `TASK:`
- Initialize the task to point to the word it will execute with `TASK!`
- `WAKE` the task
- Enable the multitasker with `MULTI`
- Unlink a task from the active list by putting it to `SLEEP`
- Relink a task into the active list with `WAKE`
- Disable the multitasker with `SINGLE`

Some reasons to use multitasking

- Take advantage of delays by interleaving parts of the application so useful work can be done in time slots that would otherwise be wasted. A print spooler is an example. If four jobs each take five minutes, the total elapsed time is 20 minutes when done sequentially, but might be 12 minutes if interleaved. This leads to faster applications.
- Increase the responsiveness of an interactive application. Accept keypresses or mouse movements while the system is also doing something else. Don't make the user wait.
- Ease the work of building a complex application by decomposing it into simpler pieces which can be considered in isolation. This leads to more reliable applications and reduces development time.
- Ordinary code, especially high-level Forth code, is easier to write and easier to test than ISRs (interrupt service routines). Some parts of an application can be done as separate tasks rather than in ISRs. This leads to more reliable applications.

Examples of multitasking

- Robots or enemies in a 3D shoot 'em up game. Let each robot run as a separate task, pursuing its own goals in its own ways. Numerous robot tasks can run the very same code, with different behavior and locations depending on each task's local data.
- Classic concurrent programming examples such as Dining Philosophers.
- Polling a serial port or other device instead of using interrupts.
- Running a user interface loop in the foreground while real work is performed in one or more background tasks.

Cooperative vs. preemptive multitasking

In describing how multitasking works, two questions arise: (1) when does a task switch occur, and (2) which task is chosen to run next? There are two broad categories of multitasking: *preemptive* and *cooperative*. Each category has different answers to "when?" and might have different answers to "who's next?" While some Forth multitaskers are preemptive, most are cooperative. While some Forths might use complex scheduling algorithms, most use a simple round-robin system.

When?

In preemptive multitasking, a task can be forced to give up control when a timer goes off. Since the timer might go off at any arbitrary point in the task, more state must be saved and restored when switching tasks. For example, a preemptive task switch might occur in the middle of a `CODE` word (so all registers must be saved and restored, including scratch registers). Special measures (such as semaphores) must be taken to guarantee that sequential operations that *must not* be interrupted *will not* be interrupted. Otherwise, shared data could be accessed when it is invalid (such as when only one byte of a two-byte variable has been updated). The answer to when a task switch occurs is "when the timer goes off," and so is unpredictable. This is more appropriate for an operating system or multi-user system where the different tasks cannot trust one another.

A cooperative multitasker relies upon the tasks themselves to relinquish control voluntarily at appropriate points. This adds the important quality of predictability. A task switch cannot occur within a `CODE` word, thus scratch registers never need to be saved and restored. If two steps must be done in sequence without interruption, the task simply does not give up control between the two steps. The answer to when a task switch occurs is "when the programmer says so." However, if a task never gives up control, it will starve the other tasks and they will never run. In a cooperative multitasking system, the tasks must indeed *cooperate* with one another. This is not the system to use if you cannot trust the tasks. If you can trust the tasks, then this system is more efficient because of the reduced overhead consumed by the multitasker.

An ISR (interrupt service routine) can be thought of as a

Frank Sergeant • San Marcos, Texas
pygmy@pobox.com • www.eskimo.com/~pygmy

Frank maintains a medical accounts receivable system and does Forth consulting. Recent projects involved custom Pygmy Forths on 80x86 (32-bit) and 68HC11 (16-bit).

FORTH INTEREST GROUP
MAIL ORDER FORM

HOW TO ORDER: Complete form on back page and send with payment to the Forth Interest Group. All items have one price. Enter price on order form and calculate shipping & handling based on location and total.

FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April).

Volume 1 ***Forth Dimensions*** **(1979–80)** **101 – $35**

Introduction to FIG, threaded code, TO variables, fig-Forth.

Volume 6 ***Forth Dimensions*** **(1984–85)** **106 – $35**

Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 ***Forth Dimensions*** **(1985–86)** **107 – $35**

Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 ***Forth Dimensions*** **(1986–87)** **108 – $35**

Interrupt-driven serial input, database functions, TI 99/4A, XMODEM, on-line documentation, dual CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 9 ***Forth Dimensions*** **(1987–88)** **109 – $35**

Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 10 ***Forth Dimensions*** **(1988–89)** **110 – $35**

dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, stand-alone applications, 8250 drivers, serial data compression.

Volume 11 ***Forth Dimensions*** **(1989–90)** **111 – $35**

Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 ***Forth Dimensions*** **(1990–91)** **112 – $35**

Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompilation.

Volume 13 ***Forth Dimensions*** **(1991–92)** **113 – $35**

Volume 14 ***Forth Dimensions*** **(1992–93)** **114 – $35**

Volume 15 ***Forth Dimensions*** **(1993–94)** **115 – $35**

Volume 16 ***Forth Dimensions*** **(1994–95)** **116 – $35**

Volume 17 ***Forth Dimensions*** **(1995–96)** **117 – $35**

NEW **Volume 18** ***Forth Dimensions*** **(1996–97)** **118 – $35**

FORML CONFERENCE PROCEEDINGS

FORML (Forth Modification Laboratory) is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is for discussion of technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.

1981 FORML PROCEEDINGS **311 – $45**
CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS – a ROM-based multitasking operating system. *655 pp.*

1982 FORML PROCEEDINGS **312 – $30**
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. *295 pp.*

1983 FORML PROCEEDINGS **313 – $30**
Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems. *352 pp.*

1984 FORML PROCEEDINGS **314 – $30**
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON – object-oriented programming, decompiler design, arrays and stack variables. *378 pp.*

1986 FORML PROCEEDINGS **316 – $30**
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. *323 pp.*

1988 FORML PROCEEDINGS **318 – $40**
Includes 1988 Australian FORML. Human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. *310 pp.*

1989 FORML PROCEEDINGS **319 – $40**
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. *433 pp.*

1992 FORML PROCEEDINGS **322 – $40**
Object-oriented Forth based on classes rather than prototypes, color vision sizing processor, virtual file systems, transparent target development, signal-processing pattern classification, optimization in low-level Forth, local variables, embedded Forth, auto display of digital images, graphics package for F-PC, B-tree in Forth *200 pp.*

1993 FORML PROCEEDINGS **323 – $45**
Includes papers from '92 euroForth and '93 euroForth Conferences. Forth in 32-bit protected mode, HDTV format converter, graphing functions, MIPS eForth, umbilical compilation, portable Forth engine, formal specifications of Forth, writing better Forth, Holon – a new way of Forth, FOSM – a Forth string matcher, Logo in Forth, programming productivity. *509 pp.*

1994–1995 FORML PROCEEDINGS (in one volume!) **325 – $50**

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon **201 – $90**

Annotated glossary of most Forth words in common use, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry. *504 pp.*

eFORTH IMPLEMENTATION GUIDE, C.H. Ting **215 – $25**

eForth is a Forth model designed to be portable to many of the newer, more powerful processors available now and becoming available in the near future. *54 pp.* (w/disk)

Embedded Controller FORTH, 8051, William H. Payne **216 – $76**

Describes the implementation of an 8051 version of Forth. More than half of this book is composed of source listings (w/disks C050) 511 pp.

F83 SOURCE, Henry Laxen & Michael Perry **217 – $20**

A complete listing of F83, including source and shadow screens. Includes introduction on getting started. *208 pp.*

F-PC USERS MANUAL (2nd ed., V3.5) **350 – $20**

Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools. *143 pp.*

F-PC TECHNICAL REFERENCE MANUAL **351 – $30**

A must if you need to know F-PC's inner workings. *269 pp.*

THE FIRST COURSE, C.H. Ting **223 – $25**

This tutorial goal exposes you to the minimum set of Forth instructions you need to use Forth to solve practical problems in the shortest possible time. "...This tutorial was developed to complement *The Forth Course* which skims too fast on the elementary Forth instructions and dives too quickly in the advanced topics in an upper-level college microcomputer laboratory ..." A running F-PC Forth system would be very useful. *44 pp.*

THE FORTH COURSE, Richard E. Haskell **225 – $25**

This set of 11 lessons is designed to make it easy for you to learn Forth. The material was developed over several years of teaching Forth as part of a senior/graduate course in the design of embedded software computer systems at Oakland University in Rochester, Michigan. *156 pp.* (w/disk)

FORTH NOTEBOOK, Dr. C.H. Ting **232 – $25**

Good examples and applications – a great learning aid. polyFORTH is the dialect used, but some conversion advice is included. Code is well documented. 286 pp.

FORTH NOTEBOOK II, Dr. C.H. Ting **232a – $25**

Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. *237 pp.*

"We're Sure You Wanted To Know..."

***Forth Dimensions*, Article Reference** **151 – $4**
An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1–15 (1978–94).

FORML, Article Reference **152 – $4**
An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980–92).

FORTH PROGRAMMERS HANDBOOK, **260 – $57**
Edward K. Conklin and Elizabeth D. Rather

EXCITING NEW TITLE!

This reference book documents all ANS Forth wordsets (with details of more than 250 words), and describes the Forth virtual machine, implementation strategies, the impact of multitasking on program design, Forth assemblers, and coding style recommendations.

INSIDE F-83, Dr. C.H. Ting **235 – $25**

Invaluable for those using F-83. *226 pp.*

OBJECT-ORIENTED FORTH, Dick Pountain **242 – $37**

Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. *118 pp.*

STARTING FORTH (2nd ed.) Limited Reprint, Leo Brodie **245a – $50**

LIMITED TIME!

In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. *(The original printing is now out of stock, but we are making available a special, limited-edition reprint with all the original content.) 346 pp.*

THINKING FORTH, Leo Brodie **255 – $35**

Back by popular demand! To program intelligently, you must first think intelligently, and that's where *Thinking Forth* comes in. The bestselling author of *Starting Forth* is back again with the first guide to using Forth for applications. This book captures the philosophy of the language, showing users how to write more readable, better maintainable applications. Both beginning and experienced programmers will gain a better understanding and mastery of topics like Forth style and conventions, decomposition, factoring, handling data, simplifying control structures. And, to give you an idea of how these concepts can be applied, *Thinking Forth* contains revealing interviews with users and with Forth's creator Charles H. Moore. Reprint of original, *272pp.*

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++,
Norman Smith **270 – $16**

This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation. (Guess what language!) Includes disk with complete source. *108 pp.*

WRITING FCODE PROGRAMS **252 – $52**

This manual is for designers of SBus interface cards and other devices that use the FCode interface language. It assumes familiarity with SBus card design requirements and Forth programming. Discusses SBus development for OpenBoot 1.0 and 2.0 systems. *414 pp.*

LEVELS OF MEMBERSHIP

Your standard membership in the Forth Interest Group brings *Forth Dimensions* and participation in FIG's activities—like members-only sections of our web site, discounts, special interest groups, and more. But we hope you will consider joining the growing number of members who choose to show their increased support of FIG's mission and of Forth itself.

Ask about our *special incentives* for corporate and library members, or become an individual benefactor!

Company/Corporate – $125
Library – $125
Benefactor – $125
Standard – $45 (add $15 for non-US delivery)

Forth Interest Group
See contact info on mail-order form, or send e-mail to:
office@forth.org

Fast service by fax: 831.373.2845

DISK LIBRARY
Contributions from the Forth Community

The "Contributions from the Forth Community" disk library contains author-submitted donations, generally including source, for a variety of computers & disk formats. Each file is designated by the author as public domain, shareware, or use with some restrictions. This library does not contain "For Sale" applications. *To submit your own contributions, send them to the FIG Publications Committee.*

FLOAT4th.BLK V1.4 Robert L. Smith **C001 – $8**
Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log.
★★★ IBM, 190Kb, F83

Games in Forth **C002 – $6**
Misc. games, Go, TETRA, Life... Source.
★ IBM, 760Kb

A Forth Spreadsheet, Craig Lindley **C003 – $6**
This model spreadsheet first appeared in *Forth Dimensions* VII/1,2. Those issues contain docs & source.
★ IBM, 100Kb

Automatic Structure Charts, Kim Harris **C004 – $8**
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs included in 1985 FORML Proceedings.
★★ IBM, 114Kb

A Simple Inference Engine, Martin Tracy **C005 – $8**
Based on inference engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source.
★★ IBM, 162 Kb

The Math Box, Nathaniel Grossman **C006 – $10**
Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs.
★★ IBM, 118 Kb

AstroForth & AstroOKO Demos, I.R. Agumirsian **C007 – $6**
AstroForth is the 83-Standard Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only.
★ IBM, 700 Kb

Forth List Handler, Martin Tracy **C008 – $8**
List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs.
★★ IBM, 170 Kb

8051 Embedded Forth, William Payne **C050 – $20**
8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family*. Included with item #216
★★★ IBM HD, 4.3 Mb

68HC11 Collection **C060 – $16**
Collection of Forths, tools and floating-point routines for the 68HC11 controller.
★★★ IBM HD, 2.5 Mb

F83 V2.01, Mike Perry & Henry Laxen **C100 – $20**
The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). Base for other F83 applications.
★ IBM, 83, 490 Kb

F-PC V3.6 & TCOM 2.5, Tom Zimmer **C200 – $30**
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications.
★ IBM HD, 83, 3.5Mb

F-PC TEACH V3.5, Lessons 0–7 Jack Brown **C201 – $8**
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology.
★ IBM HD, F-PC, 790 Kb

VP-Planner Float for F-PC, V1.01, Jack Brown **C202 – $8**
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking.
★★ IBM, F-PC, 350 Kb

F-PC Graphics V4.6, Mark Smiley **C203 – $10**
The latest versions of new graphics routines, including CGA, EGA, and VGA support, with numerous improvements over earlier versions created or supported by Mark Smiley.
★★ IBM HD, F-PC, 605 Kb

PocketForth V6.4, Chris Heilman **C300 – $12**
Smallest complete Forth for the Mac. Access to all Mac functions, events, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth*. Incl. source and manual.
★ MAC, 640 Kb, System 7.01 Compatible.

Kevo V0.9b6, Antero Taivalsaari **C360 – $10**
Complete Forth-like object Forth for the Mac. Object-Prototype access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Kernel source included, extensive demo files, manual.
★★★ MAC, 650 Kb, System 7.01 Compatible.

Yerkes Forth V3.67 **C350 – $20**
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual.
★★ MAC, 2.4Mb, System 7.1 Compatible.

Pygmy V1.4, Frank Sergeant **C500 – $20**
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time.
★★ IBM, 320 Kb

KForth, Guy Kelly **C600 – $20**
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs.
★★ IBM, 83, 2.5 Mb

Mops V2.6, Michael Hore **C710 – $20**
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, manual & source.
★★ MAC, 3 Mb, System 7.1 Compatible

BBL & Abundance, Roedy Green **C800 – $30**
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Incl. source & docs.
★★★ IBM HD, 13.8 Mb, hard disk required

Version-Replacement Policy

Return the old version with the FIG labels and get a new version replacement for 1/2 the current version price.

★ – Starting ★★ – Intermediate ★★★ – Advanced

MORE ON FORTH ENGINES

Volume 10 (January 1989) **810 – $15**
RTX reprints from 1988 Rochester Forth conference, object-oriented cmForth, lesser Forth engines. *87 pp.*

Volume 11 (July 1989) **811 – $15**
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility. *93 pp.*

Volume 12 (April 1990) **812 – $15**
ShBoom Chip architecture and instructions, neural computing module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pp.*

Volume 13 (October 1990) **813 – $15**
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth. *107 pp.*

Volume 14 **814 – $15**
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pp.*

Volume 15 **815 – $15**
Moore: new CAD system for chip design, a portrait of the P20; Rible: QS1 Forth processor, QS2, RISCing it all; P20 eForth software simulator/debugger. *94 pp.*

Volume 16 **816 – $15**
OK-CAD System, MuP20, eForth system words, 386 eForth, 80386 protected mode operation, FRP 1600 – 16-Bit real time processor. *104 pp.*

Volume 17 **817 – $15**
P21 chip and specifications; Pic17C42; eForth for 68HC11, 8051, Transputer *128 pp.*

Volume 18 **818 – $20**
MuP21 – programming, demos, eForth *114 pp.*

Volume 19 **819 – $20**
More MuP21 – programming, demos, eForth *135 pp.*

Volume 20 **820 – $20**
More MuP21 – programming, demos, F95, Forth Specific Language Microprocessor Patent 5,070,451 *126 pp.*

Volume 21
MuP21 Kit; My Troubles with This Darn 82C51; CT100 Lab Board; Born to Be Free; Laws of Computing; Traffic Controller and Zen of State Machines; ShBoom Microprocessor; Programmable Fieldbus Controller IX1; Logic Design of a 16-Bit Microprocessor P16 *98 pp.*

MISCELLANEOUS

T-shirt, "May the Forth Be With You" **601 – $18**
(Specify size: Small, Medium, Large, X-Large on order form) white design on a dark blue shirt or green design on tan shirt.

BIBLIOGRAPHY OF FORTH REFERENCES **340 – $18**
(3rd ed., January 1987)
Over 1900 references to Forth articles throughout computer literature. *104 pp.*

Last 5

DR. DOBB'S JOURNAL back issues

Annual Forth issues, including code for Forth applications.

September 1982, September 1983, Sepember 1984 (3 issues) **425 – $10**

FORTH INTEREST GROUP

100 Dolores St., Suite 183 • Carmel, California 93923 • office@forth.org

For credit card orders or customer service:
Phone Orders ***831.37.FORTH***
weekdays *831.373.6784*
9.00 – 1.30 PST ***831.373.2845*** *(fax)*

Name _____
Company _____
Street _____ voice _____
City _____ fax _____
State/Prov. _____ Zip _____ e-mail _____
Nation

Non-Post Office deliveries: include special instructions.	The amount of your sub-total...	...shipping & handling
Surface U.S. & International	Up to $40.00	$7.50
	$40.01 to $80.00	$10.00
	$80.01 to $150.00	$15.00
	Above $150.00	10% of Total
International Air		40% of Total
Courier Shipments		$15 + courier costs

Item	Title	Quantity	Unit Price	Total

☐ CHECK ENCLOSED *(payable to: Forth Interest Group)*
☐ VISA/MasterCard:

Card Number _____ exp. date _____

Signature _____

sub-total	
10% Member Discount Member#	
Sales tax* on sub-total *(California only)*	
Shipping and handling *(see chart above)*	
Membership* in the Forth Interest Group ☐ New ☐ Renewal	
TOTAL	

✖ MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a worldwide, non-profit, member-supported organization with over 1,000 members and 10 chapters. Your membership includes a subscription to the bi-monthly magazine *Forth Dimensions*. FIG also offers its members an on-line data base, a large selection of Forth literature and other services. Cost is $45 per year for U.S.A; all other countries $60 per year. This fee includes $39 for *Forth Dimensions*. No sales tax, handling fee, or discount on membership.

When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. Your membership entitles you to a 10% discount on publications and functions of FIG. Dues are not deductible as a charitable contribution for U.S. federal income tax purposes, but may be deductible as a business expense.

PAYMENT MUST ACCOMPANY ALL ORDERS

PRICES: All orders must be prepaid. Prices are subject to change without notice. Credit card orders will be sent and billed at current prices. Checks must be in U.S. dollars, drawn on a U.S. bank. A $10 charge will be added for returned checks.

SHIPPING & HANDLING: All orders calculate shipping & handling based on order dollar value. *Special handling available on request.*

SHIPPING TIME: Books in stock are shipped within seven days of receipt of the order.
SURFACE DELIVERY: U.S.: 10 days other: 30–60 days

***CALIFORNIA SALES TAX BY COUNTY:**
7.75%: Del Norte, Fresno, Imperial, Inyo, Madera, Orange, Riverside, Sacramento, Santa Clara, Santa Barbara, San Bernardino, San Diego, and San Joaquin; **8.25%**: Alameda, Contra Costa, Los Angeles San Mateo, San Francisco, San Benito, and Santa Cruz; **7.25%**: other counties.

Fast service by fax: 831.373.2845

narrowly focused, preemptive task. The answer to when this "task switch" occurs is whenever the hardware signals an interrupt (providing the interrupt has not been disabled). ISRs can coexist happily with a cooperative multitasker. Upon the hardware signal, the currently executing task is interrupted. The ISR runs, then control is returned to the task that was interrupted. The ISR is typically written in CODE to perform low-level hardware work, such as reading a serial port or timer or some other device. It knows exactly which registers it uses, and only those registers need to be saved and restored. A nice balance is to use an ISR only when timing is critical. Let the ISR handle only the core of the time-critical task, such as reading a new character from a serial port and stuffing it into a queue. Then, let normal, cooperative multitasking handle everything else. The less you do in an ISR, the better.

Who's next?

Computer science literature is littered with papers describing complex task-scheduling mechanisms, such as *priority queues*, where different tasks not only have different priorities but have dynamically changing priorities. Low-priority tasks sometimes have their priority raised to prevent them from slowing down higher-priority tasks. To a certain degree, this literature is solving problems created by the complex approaches to task scheduling. It is common for preemptive multitaskers to have more complex scheduling algorithms than cooperative multitaskers.

In some—perhaps many—cases, task scheduling can be simplified dramatically. The traditional Forth scheduling method is *round robin*. Each task gets its turn, in order. Each task retains control until it voluntarily relinquishes control by executing PAUSE. A task may execute PAUSE directly, as in

```
: TST1    BEGIN
            PAUSE  DO-SOMETHING
          AGAIN  ;
```

or it may execute PAUSE indirectly by executing another word which executes PAUSE, such as

```
: TST2    BEGIN
            100 MS  DO-SOMETHING
          AGAIN  ;
```

where the word MS executes PAUSE periodically as it kills time.

I/O words execute PAUSE. Typically, the only time you need to use an explicit PAUSE is when you are running a loop that contains no I/O or delay.

This scheduling algorithm is almost too simple to call an algorithm. Each task runs in turn. There are several benefits to this. First, the multitasking code is simpler and, therefore, more likely to be bug-free. Second, the multitasking code takes less space in memory. Third, and perhaps most important, this method runs *fast*. The overhead of task switching is small compared to preemptive methods with complex scheduling algorithms. In the time it takes for other methods to meet and discuss which task should run next, Forth just runs the tasks. It is simpler to do it than to argue about it.

Classic Forth multitasking

Classic Forth multitasking is a cooperative, round-robin system where all the tasks remain in a linked list, regardless of whether they are active or not. Each task gets control in turn. A sleeping task immediately jumps to the next task, without bothering to restore its state first, much as you might reach over to turn off an alarm clock without really waking up. This is usually done by a sleeping task's STATUS field containing a jump to the next task in the list. A task that is awake has a STATUS field that does not jump to the next task, but instead calls a routine to restore the task's state. SLEEP and WAKE change the task's STATUS field appropriately. The code to be executed is assigned to a task within a colon definition, following the word ACTIVATE.

Pygmy multitasking

Pygmy multitasking is very similar to classic Forth multitasking. The main difference is that only active tasks are kept in the task list. Thus, a sleeping task adds zero overhead to the task-switching mechanism. Instead of altering a STATUS field, SLEEP removes the task from the list and WAKE inserts the task into the list. The code to be executed by a task is defined in its own word, not by using ACTIVATE as described in the previous paragraph.

USER variables

Each task has its own private data area which can be accessed via USER variables. A USER variable is defined with an offset relative to the start of the task's private data area. Thus, the actual address returned will be different for each task in the system, yet the variables are each defined just once. For example, BASE is a USER variable. Three different tasks could be printing numbers in three different bases without interfering with each other. Other USER variables contain initial data stack and return stack pointer values, the current data stack pointer for sleeping tasks, the current video cursor address, and a link to the next task in the active task list. When only one task is awake, its link variable points back to itself. It is possible for one task to access the USER variables of another task with the word LOCAL. For example, the main terminal task could change task T1's BASE to octal with

```
8   T1 BASE LOCAL   !
```

The state you save may be your own

When a task switch occurs, the state of the task giving up control must be saved and the state of the task about to run must be restored. Cooperative multitasking makes it easier and faster to save and restore state, because task switches occur only at predictable points. Task switches occur only when a task executes PAUSE. They never occur at random points in a task's code because of a timer going off, as can happen in preemptive multitasking.

Timing issues

I divide all timing requirements into two categories: tight and sloppy. Either the timing requirements are very precise and critical, or they are not. Fortunately, almost everything fits in the sloppy category.

Obviously, the speed of the computer has a bearing on this. Generally, very few items fit in the first category. Interrupt service routines can handle them. Everything else, including the non-time-critical parts of critical tasks, is handled by ordinary round-robin Forth tasks. Since interrupt handlers are more difficult to code and test, reducing the number and

complexity of interrupt handlers increases reliability and reduces development time. The goal is to divide tasks up in the simplest way possible that does not violate your application's timing constraints.

Foreground versus background tasks

In Pygmy, there is a single foreground task that communicates with the user. Other tasks run in the background. The foreground task is called a *terminal task* because it is the task connected to the user's terminal.

Separation of concerns

Multitasking simplifies programming an application because it focuses concentration on a smaller part of the bigger problem. With small-enough pieces, we can hope to achieve "proof of correctness by observation." Don't try to hold all of a complex application in your head at once.

Simple example

TASK: is used to create a task. The variables #SP, #RP, and #USER hold the number of bytes to be reserved in each task for the data stack, the return stack, and the user variables. The default for a 68HC11 might be six user variables and 16 items on each of the stacks. For example, to set up three tasks, each of which requires (6 + 16 + 16) 38 cells, followed by four tasks, each of which requires (6 + 16 + 32) 54 cells, you could type

```
TASK: T1     TASK: T2     TASK: T3
16 CELLS #RP !     32 CELLS #SP !
TASK: T4     TASK: T5     TASK: T6
```

In deciding on the size of the stacks to be allocated to each task, consider whether the operating system or interrupts will be using the same stack. If so, be sure to leave sufficient room for that purpose, as well as for your own code.

Once a task has been created, it must be assigned a word to execute. Such a word should be an endless loop, or should explicitly put itself to sleep with STOP. If task T1 should increment a variable about once a second, first define the word that will do the real work.

```
VARIABLE SECONDS
: COUNT-SECONDS ( -)
  BEGIN   1 SECONDS +!  1000 MS  AGAIN  ;
```

Then, initialize T1 so it will execute COUNT-SECONDS.

```
' COUNT-SECONDS  T1  TASK!
```

T1 does not start running until it is made active, i.e., awakened:

```
T1 WAKE
```

Actually, all WAKE does is link the new task into a list of active tasks. For T1 to execute, you must also enable the multitasker itself by typing

```
MULTI
```

Now T1 is running COUNT-SECONDS in the background and the terminal task is still accepting input from you via the keyboard. Occasionally type SECONDS @ U. to verify the seconds are counting.

At the heart of the multitasking system, the word (PAUSE switches tasks. PAUSE is a vectored word which executes either (PAUSE or NOP. When PAUSE executes NOP, no task switching occurs. When PAUSE executes (PAUSE, the state of the current task is saved and control is transferred to the next task in the list of active tasks. The word MULTI sets PAUSE to (PAUSE. The word SINGLE sets PAUSE to NOP.

Consider the example, several paragraphs above, for the definition of COUNT-SECONDS. Although it does not contain the word PAUSE, it contains the word MS which contains the word PAUSE, so all is well. Generally, all the I/O words (and delay words such as MS) contain PAUSE.

Also, note that other tasks can access SECONDS without fear of catching it in the middle of an update. This is because a task switch never occurs because of a timer going off, only when a task executes PAUSE. Thus, COUNT-SECONDS never relinquishes control within +! and no locking mechanism or semaphore is needed, as it might be in a preemptive multitasking system.

The T1 task may be taken out of the active task list by

```
T1 SLEEP
```

A task may put itself to sleep by executing the word STOP. Note that the main terminal task would ordinarily never try to put itself to sleep.

Example: multiple-channel analog to digital converter

Think of an ADC (analog-to-digital converter) as basically a volt meter without the display. It is an electronic circuit, usually in a single chip, that translates a voltage on its input into a number on its output such that the number is proportional to the input voltage. A multi-channel ADC is a chip containing more than one ADC.

Let's suppose you have a four-channel 16-bit ADC from which you need to collect data. Assume it is memory-mapped starting at address $0800. Thus, the first channel is at $0800 and $0801, the second at $0802 and $0803, etc. Further, to smooth the data, you need to keep a running average of the four most recent readings on each channel.

Instead of trying to handle all four channels, start by writing a word SAMPLE to handle a single channel. When it is correct, set up four tasks and let each of them run SAMPLE. Remember to Keep It Simple. Don't start by writing the entire application at once. Instead, write the very simplest, pale shadow of its future self. Write the skeleton and flesh it out later. First write a routine to test the ADC interactively at the keyboard. Connect a variable voltage source to one of the ADC inputs, for example a potentiometer between 5 volts and ground, with the wiper connected to the first ADC's input. Then, read the ADC by typing

```
$0800 C@  $100 *     $0801 C@  +   U.
```

to see what number you get. As a cross-check, use a real volt meter to measure the voltage source. How does the ADC number compare to the volt meter's reading? Does the number make sense? Look at the data sheet. Did you get the byte order right, with the MSB (most significant byte) at the lower address and the LSB (least significant byte) at the higher address, or is it the other way around? Instead of typing the

above line over and over, create a shorthand word named V@ ("voltage fetch") to do it for you.

```
: V@ ( - u)   $0800 C@ $100 *   $0801 C@ + ;
```

Then you can say

```
V@ U.
```

Do various readings at various potentiometer settings and compare with the volt meter and the data sheet until you have a handle on how it works. In other words, spend a little time playing in order to gain certainty and confidence that you are on the right track. Pay particular attention to the readings at zero and at 5 volts. Should you convert the raw number to a voltage? Yes, at least for testing. Even at this beginning stage of playing, scaling the raw number to a voltage makes it easier to compare the reading with the volt meter.

Let's assume a reading of 0 represents zero volts and a reading of 65535 represents 5 volts. .Volts scales the raw number returned by the ADC and displays it as a voltage with two decimal places. (Pygmy's #> also types the result to the screen.) [See Listing One.]

Then, use .V ("print voltage") for further testing with the potentiometer.

```
: .V  ( -)  V@  .Volts  ;
```

With the potentiometer near mid-scale, you might get something like 2.48.

By taking these baby steps, you move right along. If you try to take giant steps, you might bog down in the mire.

The definition of V@ is hard-coded to read the first ADC channel. Once it is working, modify V@ so it can read any of the channels.

```
$0800 CONSTANT AtoD

( read a channel where the )
( channel number is 0 to 3 )
: V@ ( channel - raw)
  2* AtoD + DUP C@ $100 *
  SWAP C@ +  ;
```

Then try it at the keyboard as before.

What do you do with the readings? Store them in RAM or on disk? Transmit them via the serial port? For now, just write the raw numbers to the screen. This makes a four-channel, on-screen volt meter of sorts.

Assume the ADC is always ready with the current value. In real life, you might need to start a conversion, then wait for a period of time or for a ready signal. Those refinements can be added easily. The application culminates in SAMPLE which takes a channel to read and screen coordinates where the result should appear.

```
: SAMPLE  ( channel y x -)
  AT  ( chan)  V@  5 U.R  ;
```

U.R right-justifies the number in a five-character field. This prevents previous values from confusing the viewer. Try it from the keyboard. For this to work correctly, so multiple tasks can share the same display, each task must keep track of its own cursor. For example, in Pygmy, a version of EMIT could be written that used the USER variable RCURSOR for this purpose. Otherwise, when multiple tasks display the ADC reading, the digits will likely become jumbled with one another.

```
0  3 15 SAMPLE
```

Or, add KEY DROP to prevent scrolling.

```
0  3 15 SAMPLE    KEY DROP
```

If all is well, set up four tasks and let each of them run the very same SAMPLE word, but with different channel numbers and different screen coordinates.

In Pygmy, each task needs its own self-contained word that takes no parameters. This is the word that does the real work. First, put SAMPLE in a loop and delay a little between readings. Since SAMPLE contains U.R, which contains PAUSE, an explicit PAUSE is not needed. 200 MS delays for about 200 milliseconds. (MS also contains the word PAUSE.) [See Listing Two.]

Create the four tasks and assign the above words to the tasks.

```
( create) ( assign word to task) ( awaken)
TASK: S0   ' SAMPLE0  S0  TASK!    S0 WAKE
TASK: S1   ' SAMPLE1  S1  TASK!    S1 WAKE
TASK: S2   ' SAMPLE2  S2  TASK!    S2 WAKE
TASK: S3   ' SAMPLE3  S3  TASK!    S3 WAKE
```

Only after the simple version works should you try smoothing the data. You could save the three previous readings to average with the new reading, but only the previous average needs to be saved. Multiply the previous average by three and add the new reading. Divide the sum by four to get the new average. [See Listing Three.]

If you want to try running this example on a PC, you likely will not have a real ADC at address $0800. You might not even have a real ADC at all. You can fake it by reading a timer, instead. Use the following definition of V@ to simulate reading an ADC.

```
: V@ ( channel - raw) DROP  T0@  ;
```

Summary

If you are new to Forth multitasking, I hope this discussion has encouraged you to give it a try. Further, I hope it encourages us all to consider how to partition an application into simpler pieces, leading to more reliable applications which can be developed more rapidly.

Listing One

```
( Pygmy 1.5 needs U*/ for .Volts to work)
CODE UM* ( u u - ud)
  AX POP,  BX MUL,  AX PUSH,  DX BX MOV,  NXT,  END-CODE

: U*/ ( u1 u2 u3 - u1*u2/u3) PUSH UM* POP UM/MOD NIP ;

( Convert to a voltage and display with two decimals)
: .Volts ( u -)   500 65535  U*/  <# # # '. HOLD #S #>   ;
```

Listing Two

```
( Make it easy to save two words to the return stack
  and then recover them.  Consider rewriting these
  words in CODE.)
: 2PUSH ( a b -) POP  SWAP PUSH   SWAP PUSH   PUSH  ;
: 2POP  ( - a b) POP  POP SWAP    POP SWAP    PUSH  ;
: 2R@   ( - a b) POP   2POP 2DUP 2PUSH    ROT PUSH  ;

: SAMPLES ( chan  y x -)  2PUSH
  BEGIN ( chan)  DUP 2R@ ( chan chan y x) SAMPLE  200 MS AGAIN  ;

: SAMPLE0 ( -)  0   2 15 SAMPLES  ;
: SAMPLE1 ( -)  1   4 15 SAMPLES  ;
: SAMPLE2 ( -)  2   6 15 SAMPLES  ;
: SAMPLE3 ( -)  3   8 15 SAMPLES  ;
```

Listing Three

```
: AVERAGE ( previousAverage  new  - newAverage)
  4 /  ( prev new/4)  SWAP  3 4 */  ( new/4  3/4) +  ;

: SAMPLES ( chan y x -)
  2PUSH ( chan)
  DUP V@  ( fake previous average)
  BEGIN ( chan prevAverage) OVER V@ AVERAGE
        2R@ AT   DUP 5 U.R   200 MS
  AGAIN  ;
```

Forth and Functional MRI

Functional Magnetic Resonance Imaging (fMRI) is a new branch of biophysics which studies brain function via magnetic resonance imaging (MRI). A circular definition? Perhaps, but it will become clearer with a little description of the process. Specifically, in this article we will look a little at what MRI is and what the word "functional" means in regards to MRI. Then we will take a closer look at a Forth program developed for the analysis of functional MRI data. I believe this to be the first use of Forth in this area.

Magnetic Resonance Imaging

One of the wonders of modern medicine, magnetic resonance imaging has the ability to peer into the body in a non-invasive manner which does not rely on ionizing radiation (x-ray, CAT scan) or ingestion of radioactive isotopes (PET imaging). Instead, MRI uses magnetic fields and radio-frequency pulses to cause water molecules in the body to "resonate," which in turn generate a detectable radio-frequency signal. By careful adjustment of the magnetic field, it is possible to encode the output signal with information related to the position in the body and, hence, to construct an image which is related to the number of water molecules in small regions of space termed *voxels* (volume elements). Interested readers can find the "Basics of MRI" web site (www.cis.rit.edu/htbooks/mri/), which contains an online book describing MRI in much more detail than is possible here.

So, then, what is the meaning of the word "functional" in fMRI? For diagnosic purposes, MRI has been in use for roughly 20 years. As a tool for radiologists, is it indispensible for its speed, safety, and flexibility. MRI can see soft tissue that standard x-ray cannot see, but clinical MRI images are just that, static images. The body is, naturally, a living system, so it would be nice to use MRI to see it function. This is exactly what fMRI does with a focus on brain function. As a subfield, it is less than eight years old, with much of the initial work performed at the Medical College of Wisconsin, where I am currently located.

Functional MRI

Functional MRI makes static MRI dynamic by acquiring multiple images of a specific area, or *slice*, over time while the subject is performing some sort of mental task. The task might be as simple as tapping his fingers or watching a flashing checkerboard; or as complex as thinking about tapping his fingers, or memorizing words or faces and placing them in sequence. The key to functional MRI is the fact that, when neurons in the brain become active in response to a stimulus (tapping fingers for example), they extract oxygen from the surrounding blood. This, in turn, causes an increase in blood flow, which causes a decrease in the amount of deoxyhemoglobin (red blood cells without an oxygen atom attached) in the area. Deoxyhemoglobin is paramagnetic, which interferes with the magnetic resonance signal. A decrease in deoxyhemoglobin concentration will, therefore, cause an increase in the magnetic resonance signal from that voxel in the image. These blood-oxygenation-level-dependent (BOLD) signal changes are small, typically on the order of 2–5% of the voxel signal, and averaging over many trials is often necessary to get good data.

Another key to the development of fMRI is known *as echo-planar imaging* (EPI), which is a way to make a single image very rapidly—as quickly as 54 ms. at our lab, which allows one to get many data points in the time domain to watch signal changes within voxels. Traditional MRI techniques take far too long to produce images for functional MRI to work. While industry is catching up with research and beginning to offer EPI abilities in scanners, until recently each site had to construct its own EPI hardware, which was time consuming and expensive.

Functional MRI is a new and growing area which promises to aid greatly in furthering medical advances in basic research and in diagnosis and treatment of disease. Already fMRI has had important success in understanding the effects of cocaine abuse and in diagnosis of schizophrenia. Recent work is demonstrating key elements of the process of memorization, and the potential for applications is virtually unlimited and untapped.

A typical fMRI experiment

A typical block-designed functional MRI experiment proceeds as follows. The subject is placed within the scanner and images of the same portion of the brain are taken at regular intervals. The time between images is typically on the order of one second but may be significantly shorter. During image acquisition, the subject is performing some sort of mental or physical task, as described above.

When the scanning session is complete, the resulting images contain a record of any variation in the magnetic resonance signal which may have occurred in connection with the applied stimulus. This data is then given to the fMRI analysis program for evaluation. The program correlates the time course of each pixel with the applied stimulus or reference function, which is often represented as a box car function with a value of 1 when the stimulus was present and a value of 0 otherwise. The results of this correlation calculation give an indication of which regions of the brain responded, or became "active," when the stimulus was applied. Activated regions are defined as those which have a cross-correlation value above some chosen threshold. The higher the cross-correlation value between the reference function and the pixel time series, the more the two are "alike," implying a response to the stimulus.

Ronald T. Kneusel • Milwaukee, Wisconsin
rkneusel@mcw.edu

Ron is a computer programmer for the Medical College of Wisconsin, and is a part-time biophysics graduate student.

Forth enters the picture

The output of an fMRI experiment is a series of hundreds, even thousands, of individual EPI images. These images form a three-dimensional data set, with the X and Y axes being the image axes, and the Z axis being time. A typical analysis would then need to look at specific images in the series and also plot time courses of individual pixels (voxels and pixels are frequently interchanged as, in a displayed image, each pixel corresponds to the signal intensity from a voxel). In order to determine which areas of the image became active with the stimulus applied during the experiment, a calculation is performed which correlates the pixel time courses with the applied stimulus time course. The result of this calculation indicates regions in the image associated with the stimulus and allows these pixel time courses to be extracted and analyzed further, if need be.

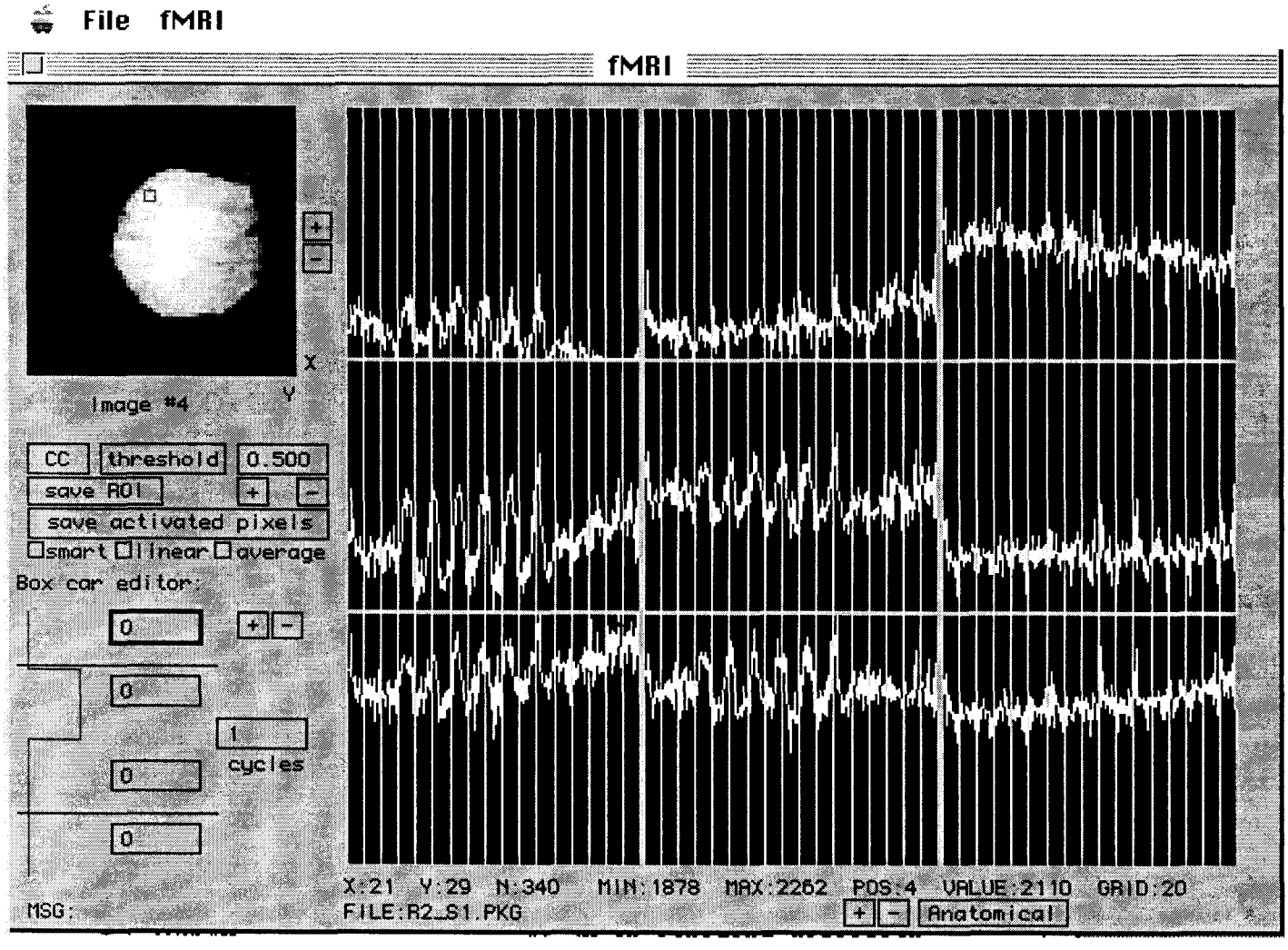
To accomplish this, I wrote a Forth program for the Apple Macintosh using Chris Heilman's freeware Pocket Forth. I described the general procedure for creating applications with Pocket Forth in *Forth Dimensions* XIX.3 and will, therefore, concentrate here on the specifics of the fMRI program.

But, why choose Forth to begin with? Forth is, of course, suitable to most any task, but in this case it was particularly suitable for several reasons. Typical MRI images are stored as 16-bit integers, which is ideal for a small Forth to handle. Additionally, an fMRI dataset is easily stored as a package with all the images in a single file, making image access no different than accessing elements of an array, each element of which is 8192 bytes in size. With only a handful of exceptions, all calculations can be handled by integer arithmetic, thereby greatly increasing the speed of the program, especially on older hardware. Finally, Forth's interactive nature aided greatly in program development by allowing rapid testing of new words and ideas.

The application

Figure One is a screen shot of the fMRI application with some experimental data loaded. The application uses only one window, which is divided into three main sections. The first shows the actual images in the dataset, using internal greyscale or reverse greyscale palettes or an external user-defined color palette (256 RGB entries). The second section shows the time courses of the nine pixels within the rectangle superimposed upon the image. It can be thought of as viewing the column of images from the side. This area of the window also shows the reference function, if any, and is used to show expanded graphs or images. The third section of the window contains buttons for controlling the cross-correlation and threshold, and for saving regions of interest (ROI) or activated pixels. In this context, "saving" means writing

Figure One. *[Color images reproduced here in grayscale.–Ed.]*



the actual pixel time courses, or an average, out to disk as tab-delimited ASCII text. Additionally, this area of the application window contains the box car editor, which is useful for defining simple on/off reference functions.

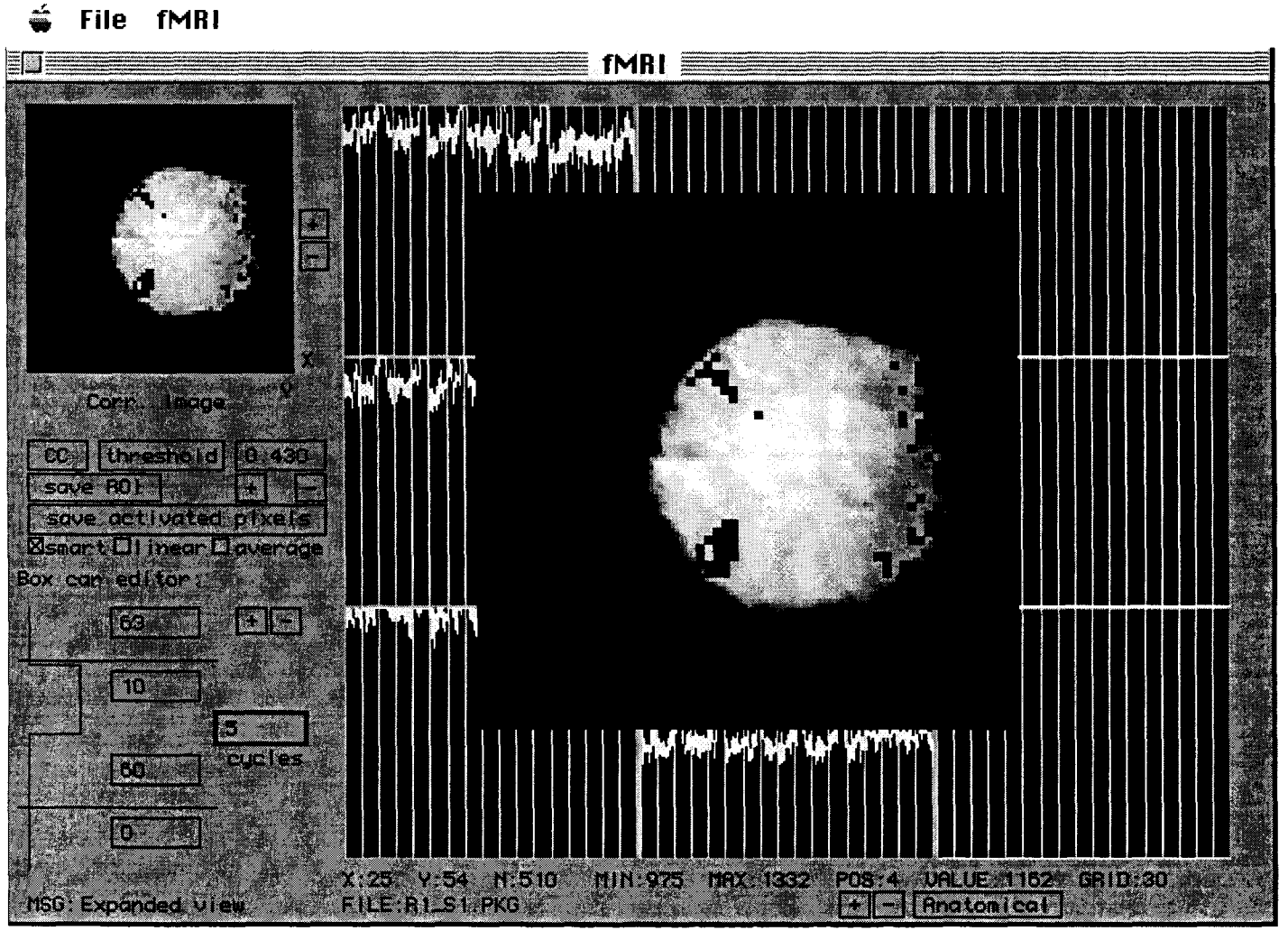
The three checkboxes—*smart*, *linear*, and *average*—are used to modify the actions of the buttons above them. The *smart* checkbox implements an attempt to speed the cross-correlation calculation by only computing the correlation for pixels which have a value that is at least 1/4 the value of the mean pixel intensity of the first image. In this way, the program does not waste time calculating a value for areas of the data which are outside the brain itself. The *linear* checkbox is often necessary. It removes any linear trend from a pixel time course before calculating the cross-correlation with the reference function. Subject motion is one source of this linear drift. Lastly, the *average* checkbox is used to modify the behavior of the "save ROI" and "save activated pixels" buttons. In place of a large output file containing all the data from all activated pixels, it will write a single, average time series.

The application also has two menus, File and fMRI. The File menu is used to load image datasets (packages), *anatomicals* (high-resolution MRI images), and palette files, or to write the currently displayed image or time series to disk. The fMRI menu is used to set up for functional analysis. It uses the current settings of the box car editor, or the current pixel time course, to define the reference function. Additionally, it is used to load a reference function from an ASCII disk file or to sum multiple pixels into the reference function. Three-channel binomial smoothing of the reference function is also available.

Figure Two shows the result of a correlation calculation between a selected pixel as a reference function and the reset of the data set. The expanded image plot shows areas of the brain which became active in response to the stimulus. In this case, the subject was instructed to watch a flashing checkerboard image and to tap his fingers when the checkerboard was present. This is an axial image with the forehead on the left. The two smaller activated regions on the left correspond to the primary motor cortex region of the brain. Had the subject been tapping the fingers on his left hand only, there would be a single active region near the top of the image (the right side of the brain). The larger activated region on the right corresponds to the primary visual cortex and is the result of the flashing checkerboard stimulus. These regions, or a portion of them, can be written to disk and analyzed further.

Functional MRI is new and exciting and a prime opportunity for Forth to make an entrance. This example is the first of what I hope to be several Forth-based fMRI analysis tools.

Figure Two. Result of a correlation calculation (shown in black and white).



The Problem with Buffers

Let us say that we have a program on a microcontroller which inputs data and does something with this data. The data comes to us in bursts, so we need to buffer it. Overall, we process the data faster than it arrives, but when we get a burst of input, all we have time to do is store it in memory.

We make our buffer as large as possible, filling our entire memory, in order to reduce the chance that our program will fail by not having enough buffer space available to hold a burst of data when it comes in. We have two pointers, TODO and PAST. TODO points to the data which awaits processing, and PAST points to just past this data. Our buffer initially looks like Figure One.

CBUFFER and BEYOND are fixed pointers which never change during the course of the program's execution. Let's say that we get a burst of data. We put this data in our buffer and move PAST forward to the next available location [see Figure Two].

We now have some data to process (in our diagrams, valid data is represented by shaded areas). We do some processing and we consume some of this data, moving TODO forward [Figure Three].

At this point, we get another burst of data. We place this data past our present data, moving PAST forward. Our data comes in sequentially (from a serial port, perhaps), so we can wrap around the end of the buffer [Figure Four].

So far, everything has been easy and obvious. We will now introduce a complication. We were able to wrap around the end of the buffer when we input our data, but let us say that we can't when we process it. Our data is composed of records which we must process as whole units. The routine which processes the records expects them to occupy contiguous addresses.

Let us say that, in the above diagram, we have a record to process which currently has its front half at the back of the buffer (pointed to by TODO) and its back half at the front of the buffer (pointed to by CBUFFER). Our data must be contiguous before we can begin processing it. We need to rotate our data within the buffer so that TODO ends up pointing to the beginning of the buffer (equal to CBUFFER) [Figure Five].

An obvious solution would be to copy the data from TODO through BEYOND into a temporary storage location. We would then copy the data from CBUFFER through PAST to an address equal to CBUFFER + (BEYOND – TODO). We would then copy our data from the temporary storage location to CBUFFER. This won't work, though. Our buffer pretty much fills the available RAM; we don't have room for temporary storage anywhere. We don't want to shrink the size of the buffer to make room for a temporary storage area, because the buffer needs to be as big as possible to prevent data loss during a burst of input.

The next section of this article will discuss a solution to the

Figure One

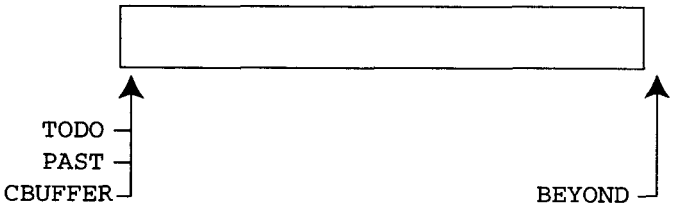


Figure Two

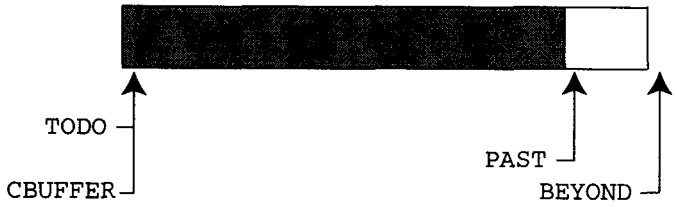


Figure Three

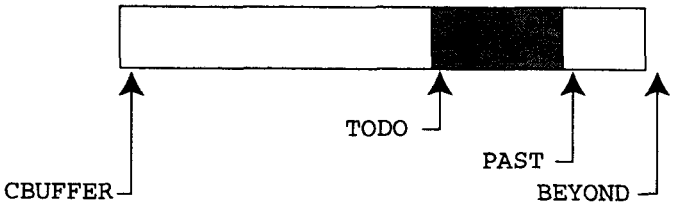


Figure Four

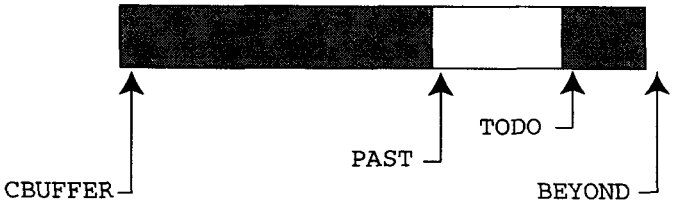
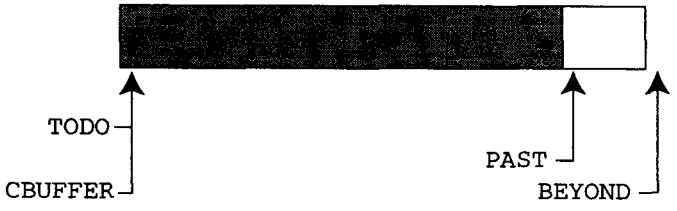


Figure Five



Hugh Aguilar • vaguilar@dancris.com
Phoenix, Arizona

Hugh Aguilar has worked mostly as an assembly language programmer, but has also held jobs using C++ and Forth. He is not currently programming.

problem of rotating data within a buffer; the reader may want to take a moment now to figure out a solution of his own.

Buffer "normalization"

Listing One provides a solution to the problem of rotating data within a buffer. We employ an algorithm which is the author's own invention. The code is written in UR/Forth from Laboratory Microsystems, Inc. (both the 16-bit and 32-bit MS-DOS versions), but should run unmodified on any Forth-83 system. For demonstration purposes, we have a small buffer of twenty-three bytes which we display as an ASCII string before and after the rotation. Our algorithm should work with any size buffer up to almost the full address space.

We have a routine called INIT_CBUFFER which initializes the CBUFFER and also the TODO and PAST pointers. We also have a routine called DISPLAY_CBUFFER which displays the CBUFFER and the two pointers on the screen. The routine DEMONSTRATE demonstrates the whole process; this is the routine the user should run from his console.

The meat of our program lies in the routines NORMALIZE_TODO and ADJUST_PTRS. NORMALIZE_TODO rotates the data in the buffer, and ADJUST_PTRS adjusts the values of TODO and PAST. We will first examine NORMALIZE_TODO which is the more complicated.

We have a double variable called DISTANCE which is the difference betwean TODO and CBUFFER; this is how far the data must be rotated. PREP_ROTATE sets DISTANCE and also provides our initial destination and source addresses. The destination is initially our TODO value, and the source is the destination value plus the difference. Note that we don't simply add or subtract values to pointers.

We have to be careful about falling off the edges of the buffer. Our routines ADVANCE_PTR and RETARD_PTR add and subtract the DISTANCE value to a pointer.

Our ROTATE_CBUFFER routine takes the character at the source and stores it into the destination. At this point, it calls ADVANCE_PTRS which produces new destination and source values by advancing the old values forward by the DISTANCE value. Note that, since the destination and the source are already separated by the DISTANCE value, we don't have to call ADVANCE_PTR on the destination value. We just discard the old destination value and use the old source value as the new destination value. We do have to call ADVANCE_PTR on the old source value to get the new source value. ROTATE_CBUFFER iterates until it has been through all of the combinations of destination and source values that it can do. If ROTATE_CBUFFER iterated one more time, it would find that its source location no longer holds the correct value; this location was the original destination location and got plugged with a new value at that time. ROTATE_CBUFFER doesn't iterate this last time, and this one location we must handle explicitly.

Prior to calling ROTATE_CBUFFER we fetch the original character from the destination location and we hang onto it (on the R-stack). After ROTATE_CBUFFER finishes, we store this character into what would have been ROTATE_CBUFFER's next destination location (the first character of the buffer). This completes NORMALIZE_TODO's work.

ADJUST_PTRS is fairly simple. We just call RETARD_PTRS for the TODO and PAST pointers, adjusting them so that they correspond to the new orientation of the data in CBUFFER. Note that all of these operations could have been done in the opposite directions. We set DISTANCE to TODO minus CBUFFER. In NORMALIZE_TODO we called ADVANCE_PTR to iterate through our pointers. In ADJUST_PTRS we called RETARD_PTR to adjust our pointers. We could have set DISTANCE to BEYOND minus TODO and then used RETARD_PTR in NORMALIZE_TODO and ADVANCE_PTR in ADJUST_PTRS. The effect is the same, it is just that everything is going in the opposite directions. The reader may want to rewrite the program in this manner to be sure that he understands the process.

Notice that the size of the buffer must be a prime number. This ensures that ROTATE_CBUFFER iterates through all of the locations. In other words, we want to be assured that DISTANCE will never be an even divisor of CBUFFER_SIZE. Because TODO can be anywhere in CBUFFER, DISTANCE can be any number from zero to CBUFFER_SIZE-1. We have to make CBUFFER_SIZE a prime number to ensure that no number in the range 0,CBUFFER_SIZE) evenly divides it.

A lot of programmers tend to take a few successful runs of a program as "proof" that the program is correct. The author of this article is guilty of this as often as anybody. It is a bad habit. Consider a programmer who might write the above program but fail to realize that CBUFFER_SIZE must be a prime number. This programmer might, by happenstance, set CBUFFER_SIZE at 64009, which is an "almost-prime" number. This value is just short of 64K, which is in keeping with our idea of making the buffer almost fill available memory. The only time the program would fail would be when TODO happened to be 253 bytes away from CBUFFER. This would only happen, on the average, one out of every 64009 executions of NORMALIZE_TODO, or about 0.00156% of the time. Rare bugs like this don't generally show up during testing, only after a program has gone into production. This is a good illustration of the danger in relying on testing as a substitute for thinking things through.

Pointers are integers

Notice that, in our program, we converted our pointers to double-precision integers inside of ADVANCE_PTR and RETARD_PTR. At the end of these routines, we converted our values back into single-precision pointers. We did this because we expected our buffer to be almost filling our available memory. Let us say that we are working on a 16-bit computer and have 64K of memory available. If our buffer is almost 64K, then our DISTANCE value, added or subtracted from some pointer within the buffer, could easily overflow our single-precision arithmetic operations. In fact, if our buffer size is over 21K (one third of our address space size), we can guarantee that we will overflow single-precision arithmetic somewhere within our program. Even with a buffer size under 21K, we will still overflow single-precision arithmetic unless our buffer is exactly centered in the middle of the address space. Small buffer sizes are clearly not in keeping with the normal usage of buffers, in which bigger is better. Even on a 32-bit machine, we could overflow a 32-bit integer if our buffer was fairly big and was relatively close to the bottom of the address space. The only good solution to overflowing arithmetic is to type cast to a greater -precision arithmetic. This is what we did in ADVANCE_PTR and RETARD_PTR.

Although this publication is concerned with the Forth programming language, let us now quote from *The C Programming Language* (second edition) Ritchie. On page 102, K&R say:

> Pointers and integers are not interchangeable. Zero is the only

> exception... Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array (there is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Bjarne Stroustrup has a similar speech to give in *The C++ Programming Language* (page 57):

> The pointer values were converted to the type `long` [in the example program] before the subtraction using explicit type conversion. They were converted to `long`, and not to the "obvious" type `int`, because, in some implementations of C++ a pointer will not fit into an `int` (that is, `sizeof(int)<sizeof(char*)`).
>
> Subtraction of pointers is defined only when both pointers point at elements of the same array (although the language has no way of ensuring that is the case). When subtracting one pointer from another, the result is the number of array elements between the two pointers (an integer). One can add an integer to a pointer or subtract an integer from a pointer; in both cases, the result is a pointer value. If that value does not point to an element of the same array as the original pointer or one beyond, the result of using that value is undefined.

ADVANCE_PTR and RETARD_PTR cannot be ported directly from our program into C or C++ because our pointer value may be outside of the CBUFFER array briefly. In our program, we test to see if the pointer is outside of the array, and wrap it around if it is. This cannot be done directly in C or C++ because a pointer outside of an array is an undefined value, according to Stroustrup. To convert to C or C++, we would need to write the ADVANCE_PTR and RETARD_PTR routines as shown in Listing Two (note the use of long arithmetic, as per Stroustrup's advice).

It is the author's contention that this is obscure and inobvious code. It is probably also less efficient because we are doing three long arithmetic operations (four, if the conditional is true), compared to the Forth version which does two long arithmetic operations (three, if the conditional is true). Mostly, though, our complaint is the fog-factor that comes with writing C code; it is an error-prone business.

Note that we are not saying that a language definition should indicate how many bits each data type takes (like Java does). C and C++, like Forth, did the right thing in leaving this undefined. Defining the data type size largely ties a language to a processor size. Our Forth program, like almost all Forth programs, will run equally well on both 16-bit and 32-bit UR/Forth.

On a 16-bit Forth, our program uses 16-bit single-precision and 32-bit double-precision, which is optimum for that architecture. Java, because it specifies integers and pointers as 32-bit, must necessarily run on a 32-bit or bigger machine. This largely prevents Java from being used in embedded controllers, which was supposedly Java's original application. Forth programs can easily port without modification between a 16-bit embedded controller and a 32-bit desktop machine.

Forth defines integers and pointers as being equal in size and being the natural size of data for a machine. C pointers may or may not be equal in size to integers. They might be equal in size to long integers. For safety, pointers should always be type-cast to long integers. This is a problem because there is nothing bigger than a long integer and so there is no easy method of dealing with overflow.

Listing One

```
screen 1.
\ data declarations                                  05:16 05/10/97

23 CONSTANT CBUFFER_SIZE     \ must be a prime number!

CREATE CBUFFER  CBUFFER_SIZE ALLOT  HERE CONSTANT BEYOND

VARIABLE TODO    \ pointer to beginning of valid data
VARIABLE PAST    \ pointer to just past the valid data

2VARIABLE DISTANCE  \ distance between TODO and CBUFFER

screen 2.
\ INIT_CBUFFER DISPLAY_CBUFFER                       05:09 05/10/97

: INIT_CBUFFER  \ --  \ sets CBUFFER, TODO, and PAST
    " fghijklmnopqrst123abcde" COUNT
    DUP CBUFFER_SIZE <> ABORT" string is wrong length"
    CBUFFER SWAP CMOVE
    CBUFFER 18 +  TODO !     \ adr of the 'a'
    CBUFFER 15 +  PAST ! ;   \ adr of the '1'

: DISPLAY_CBUFFER  \ --  \ needs CBUFFER, TODO and PAST
    CR  8 SPACES  CBUFFER CBUFFER_SIZE TYPE
    CR  TODO @ CBUFFER - 4 + SPACES  ." TODO^"
    CR  PAST @ CBUFFER - 4 + SPACES  ." PAST^"  CR ;

screen 3.
\ ADVANCE_PTR RETARD_PTR ADJUST_PTRS                 05:16 05/10/97

: ADVANCE_PTR  \ ptr -- new_ptr      \ by DISTANCE
    0  DISTANCE 2@  D+  \ D --
```

```
    2DUP  BEYOND 0  D>= IF  CBUFFER_SIZE 0  D-  THEN
    0<> ABORT" we aren't back in our address space" ;

: RETARD_PTR  \ ptr -- new_ptr       \ by DISTANCE
    0  DISTANCE 2@  D-  \ D --
    2DUP  CBUFFER 0  D< IF  CBUFFER_SIZE 0  D+  THEN
    0<> ABORT" we aren't back in our address space" ;

: ADJUST_PTRS  \ --  \ adjusts TODO and PAST
    TODO @ RETARD_PTR TODO !  PAST @ RETARD_PTR PAST ! ;
```

screen 4.

```
\ MOVE_DATUM ADVANCE_PTRS ROTATE_CBUFFER          02:54 05/10/97

: MOVE_DATUM  \ dst src --
    C@  SWAP C! ;

: ADVANCE_PTRS  \ dst src -- new_dst new_src
    NIP  \ discard dst, our old src will be our new dst
    DUP ADVANCE_PTR ;

: ROTATE_CBUFFER  \ dst src --
    OVER >R  BEGIN
        2DUP MOVE_DATUM  ADVANCE_PTRS
        DUP R@ = UNTIL  R> DROP
    2DROP ;  \ these are just short of their original values
```

screen 5.

```
\ PREP_ROTATE NORMALIZE_TODO DEMONSTRATE          05:16 05/10/97

: PREP_ROTATE  \ -- dst src  \ sets DISTANCE
    TODO @ CBUFFER -  0  DISTANCE 2!
    TODO @  DUP ADVANCE_PTR ;

: NORMALIZE_TODO \ --  \ adjusts TODO and PAST
    PREP_ROTATE          OVER C@ >R
    ROTATE_CBUFFER       R> CBUFFER C! ;

: DEMONSTRATE  \ --
    INIT_CBUFFER                    DISPLAY_CBUFFER
    NORMALIZE_TODO  ADJUST_PTRS     DISPLAY_CBUFFER ;
```

Listing Two

```
// assume CBuffer and Beyond are char * types
// assume Distance is an unsigned long int

char *AdvancePtr( char *Ptr)
{
    unsigned long int TrialDist= (long)Beyond -(long)Ptr;
    if( TrialDist <= Distance)  // we will wrap!
    {
        return( CBuffer +( Distance -TrialDist));
    } else {
        return( Ptr +Distance);
    }
}

char *RetardPtr( char *Ptr)
{
    unsigned long int TrialDist= (long)Ptr -(long)CBuffer;
    if( TrialDist < Distance)  // we will wrap!
    {
        return( Beyond -( Distance -TrialDist));
    } else {
        return( Ptr -Distance);
    }
}
```

Reed-Solomon Error Correction

This article describes how to do Reed-Solomon error correction in Forth. Reed-Solomon is a type of forward error correction used in disk drives, CDs, satellites, and other communication channels. *Forward error correction* means that redundancy (extra bytes, for example) is added to a block of data before sending. At the destination, these extra data are used to determine if an error has occurred and to correct the error, if possible. Using forward error correction reduces the necessity of retransmitting data in error. In some cases, such as for disk drives, by the time an error is detected, the original data is not available and forward error correction must be used.

Reed-Solomon Error Correction Codes (ECC) use a type of arithmetic called *finite-field* or *Galois field* arithmetic. Galois was a French mathematician who advanced the field of finite math an entire generation in a single night. Challenged to a duel over a woman, he wrote all he could on the subject the night before, and was killed the next morning. Mathematicians have enjoyed speculating what he might have subsequently contributed, had he survived.

This article is presented in two parts. The first is an introduction to finite field arithmetic, and shows how to generate finite fields, and how to use them. The second tells how to design and use Reed-Solomon ECC.

Finite fields are called such because, unlike the set of integers or real numbers, they contain a finite set of elements. The elements in a finite field are called *symbols*, and most symbols used today are base two, or binary numbers. All base two finite fields have 2^n symbols, where n is a positive integer. Thus, a 2^8 field will have 256 symbols, and they will be the 256 bytes we are familiar with. Today's algorithms become unwieldy for fields larger than about 2^{16}, and most fields are in the 2^6–2^{12} range.

Addition, subtraction, multiplication, and division operations are defined for finite fields, so we will define Forth words for these:

```
FF+    ( n1 n2 – n3)
FF-    ( n1 n2 – n3)
FF*    ( n1 n2 – n3)
FF/    ( n1 n2 – n3)
```

(`n1`, `n2`, and `n3` are all symbols in the finite field)

Any of these operations will give a result that is also in the finite field (as with familiar number systems, division by zero is undefined). We can, therefore, express the finite field in terms of addition, subtraction, multiplication, division, and even log and antilog tables. Tables for a 2^4 field are shown in Figure One. You may find patterns in the numbers in this simple field, but larger fields may look like gibberish. Multiplying by one or zero always gives results we are familiar with, but the patterns for other entries in the tables are not obvious.

Note that a subtraction table is not given. In finite field arithmetic, addition and subtraction are exactly the same operation: XOR! You can verify that this is works by selecting two arbitrary numbers and adding them (table lookup), then subtract one of the original numbers (again with table lookup). It works every time (as it should, since we `know n1 n2 XOR n2 XOR` gives `n1` back: in finite field parlance, we are adding `n2` to `n1`, then subtracting `n2` from the result). Try some multiply and divide examples as well.

The properties of commutation, association, and distribution all work with finite field arithmetic. And we can add another property: subtraction is also commutative, since it's the same as addition.

Like the magic squares (matrices whose rows, columns, and diagonals add to the same number), we played with as children, only certain symbol arrangements work for a finite field. For a four-bit field, there are only six possible distinct arrangements, not counting rotations. For an eight-bit field, there are thirty-eight. The larger the field, the more arrangements there are that work.

How do we generate a finite field? The field is generated using XOR and shift operations with irreducible polynomials (finite field polynomial math is a big part of ECC). The irreducible polynomials are tough to find, but tables of them exist. Listing One gives some for symbol sizes through twelve bits. For each irreducible polynomial, there is a corresponding finite field. These fields can also be "rotated" by specifying an offset in the log/antilog tables.

The code in Listing One will generate a finite field, given an irreducible polynomial. It also defines the three arithmetic operators (`FF-` is skipped; use `FF+`), log/antilog, and power operations.

Listing Two gives utility words that display the Figure One tables for the field you generated. Of course, for large fields, the tables will not display properly, due to their size.

To generate a finite field, choose the size by setting the number of bits in the `MASK` constant. Then choose the generating polynomial from the list given. There are subtle differences in the performance of different polynomials; for our purposes, any in the list will work.

Once the code is loaded and the verifier has given a message that it has successfully completed, you are ready to do finite field arithmetic. Do it just like regular integer arithmetic, on the stack. Note that, without special hardware, it's most efficient to do `FF*` and `FF/` using log/antilog table lookups. The log of a finite field symbol is a regular integer, so to multiply, standard addition (mod the field size) of the logs is used. See the definitions of `FF*` and `FF/`.

In the next article, we will use the finite field operators to make a Reed-Solomon encoder and decoder, and show how to do the math to detect and correct errors.

Glenn Dixon • Roy, Utah
Dixong@iomega.com

Glenn Dixon works at Iomega Corporation. He served as channel engineer for the Zip 100 and Zip 250, and uses Forth for anything the company will allow. When not Forthing, he writes murder mysteries.

Figure One

```
POLYNOMIAL (LESS TOP TERM)= 3 MASK= F
U=Undefined
```

Antilog table

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	4	8	3	6	C	B	5	A	7	E	F	D	9	U

Log table

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U	0	1	4	2	8	5	A	3	E	9	7	6	D	B	C

Addition table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	0	3	2	5	4	7	6	9	8	B	A	D	C	F	E
2	2	3	0	1	6	7	4	5	A	B	8	9	E	F	C	D
3	3	2	1	0	7	6	5	4	B	A	9	8	F	E	D	C
4	4	5	6	7	0	1	2	3	C	D	E	F	8	9	A	B
5	5	4	7	6	1	0	3	2	D	C	F	E	9	8	B	A
6	6	7	4	5	2	3	0	1	E	F	C	D	A	B	8	9
7	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8
8	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7
9	9	8	B	A	D	C	F	E	1	0	3	2	5	4	7	6
A	A	B	8	9	E	F	C	D	2	3	0	1	6	7	4	5
B	B	A	9	8	F	E	D	C	3	2	1	0	7	6	5	4
C	C	D	E	F	8	9	A	B	4	5	6	7	0	1	2	3
D	D	C	F	E	9	8	B	A	5	4	7	6	1	0	3	2
E	E	F	C	D	A	B	8	9	6	7	4	5	2	3	0	1
F	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

Multiplication table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	3	1	7	5	B	9	F	D
3	0	3	6	5	C	F	A	9	B	8	D	E	7	4	1	2
4	0	4	8	C	3	7	B	F	6	2	E	A	5	1	D	9
5	0	5	A	F	7	2	D	8	E	B	4	1	9	C	3	6
6	0	6	C	A	B	D	7	1	5	3	9	F	E	8	2	4
7	0	7	E	9	F	8	1	6	D	A	3	4	2	5	C	B
8	0	8	3	B	6	E	5	D	C	4	F	7	A	2	9	1
9	0	9	1	8	2	B	3	A	4	D	5	C	6	F	7	E
A	0	A	7	D	E	4	9	3	F	5	8	2	1	B	6	C
B	0	B	5	E	A	1	F	4	7	C	2	9	D	6	8	3
C	0	C	B	7	5	9	E	2	A	6	1	D	F	3	4	8
D	0	D	9	4	1	C	8	5	2	F	B	6	3	E	A	7
E	0	E	F	1	D	3	2	C	9	7	6	8	4	A	B	5
F	0	F	D	2	9	6	4	B	1	E	C	3	8	7	5	A

Division table

Numerator:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	9	1	8	2	B	3	A	4	D	5	C	6	F	7	E
3	0	E	F	1	D	3	2	C	9	7	6	8	4	A	B	5
4	0	D	9	4	1	C	8	5	2	F	B	6	3	E	A	7
5	0	B	5	E	A	1	F	4	7	C	2	9	D	6	8	3
6	0	7	E	9	F	8	1	6	D	A	3	4	2	5	C	B
7	0	6	C	A	B	D	7	1	5	3	9	F	E	8	2	4
8	0	F	D	2	9	6	4	B	1	E	C	3	8	7	5	A
9	0	2	4	6	8	A	C	E	3	1	7	5	B	9	F	D
A	0	C	B	7	5	9	E	2	A	6	1	D	F	3	4	8
B	0	5	A	F	7	2	D	8	E	B	4	1	9	C	3	6
C	0	A	7	D	E	4	9	3	F	5	8	2	1	B	6	C
D	0	4	8	C	3	7	B	F	6	2	E	A	5	1	D	9
E	0	3	6	5	C	F	A	9	B	8	D	E	7	4	1	2
F	0	8	3	B	6	E	5	D	C	4	F	7	A	2	9	1

^*Denominator*

Listing One. (Electronic copies of the listings are available by e-mail request to the author.)

```
\ FILE FiniteField.t to implement and test finite fields of GF(2^K) symbols.
\ Will handle up to K=12 or so depending on your memory.
\ This should work on any ANSI forth, 16 or 32 bit, with no dependencies.

DECIMAL

\ A SHORT LIST OF IRREDUCIBLE POLYNOMIALS [ 1] :  $ means hex
\ the numbers below represent polynomials.  The actual polynomial represented is
\ x^k + x^(k-1)*bit(k-1)+x^(k-2)*bit(k-2)+ . . . +x^0*bit0
\ example, the five-bit poly $1D is
\ x^5+x^4+x^3+x^2+1
\ 3 bit: 3 ; 4 bit: 3  $17  ; 5 bit: $5 $1D $17
\ 6 bit: 3 $17 $27 ; 7 bit: 9 $f $1D  ; 8 bit: $1D  $77  $F3
\ 9 bit: $11 $59 $131 ; 10 bit: 9  $F  $10D  11 bit: 5  $125  $8D
\ 12 bit: $53  $45B  $4D

\ Define your finite field as follows: Choose the number of bits in your symbol
\ then set MASK to the all-1's symbol
\ Example: For a 5 bit symbol, mask=$1F, For an 8 bit symbol, MASK=$FF
\ Then set POLYNOMIAL to one of the polynomial numbers in the above table.

HEX F DECIMAL CONSTANT MASK \ MASK=2^k-1 where k=symbol size
3 CONSTANT POLYNOMIAL  \ From the above table, 4 bit symbol

\ The field can also have an offset, which basically rotates the entries:
0 CONSTANT M0  \ M0 is an offset. In some cases, a non-zero offset is used to
\ increase calculation efficiency. A non-zero value rotates the field.

\ we start by making an antilog table.  This is accomplished with shift-XOR
\ operations on the chosen polynomial.

CREATE ANTILOG MASK 1+ cells ALLOT  \ An antilog table
: FILL-ANTILOG
   1 MASK 0 DO DUP I CELLS ANTILOG + !  \ Store tha value in the table
       MASK U2/ MASK XOR OVER AND IF    \ Test top bit
         2* POLYNOMIAL XOR              \ Shift and XOR if set
       ELSE 2*
       THEN MASK AND                    \ Calculate next shifted value
       LOOP DROP ;
FILL-ANTILOG                            \ Fill the table

\ We now make a log table by reading the antilog table.

: >ANTILOG  ( n1--n2) \ Find the antilog of N1 using table lookup
\ Note: multiply and divide operations can give inputs exceeding table indices.
\ Wraparound with the MOD operation fixes this.
   MASK MOD CELLS ANTILOG + @ ;

CREATE LOG MASK 1+ CELLS ALLOT

\ FILL-LOG just looks through antilog table and puts corresponding entry in log table.

: FILL-LOG
     MASK  0 DO I >ANTILOG 1- CELLS LOG +  I  SWAP ! LOOP ;
```

```
FILL-LOG

: >LOG ( n1--n2) \ find log of n1  returned number is an integer (ordinal)
   DUP 0= ABORT" ATTEMPTED TO FIND LOG OF 0 IN >LOG!!!"
    1- CELLS LOG + @ ;

: FF+ ( N1 N2--N) \ Finite field add (and subtract!)
  XOR ;
\ If your system has SYNONYM, use it for FF+ to improve efficiency

: FF*   ( n1 n2--N)  \ Finite field multiply using logs
   DUP 0<> >R        \ If N1 or N2 are zero, log method won't work
   OVER 0<> R> AND IF
   >LOG SWAP >LOG +    \ Just add logs to multiply.
   >ANTILOG
   ELSE 2DROP 0 THEN
   ;

: FF/  ( N1 N2--N1/N2)  \ Finite field divide using logs
   DUP 0= ABORT" ATTEMPT TO DIVIDE BY 0 IN FF/!"
   OVER 0<> IF  \ zero numerator must have special handling.
   >LOG SWAP >LOG SWAP -
   >ANTILOG
   ELSE 2DROP 0 THEN ;

: FF^   ( n1 POWER--n2) \ n1 RAISED TO A POWER: Power is a normal ordinal integer
   DUP 0= IF 2DROP 1 ELSE
    1 SWAP 0 DO  OVER FF* LOOP SWAP DROP THEN \ brute force.
    ;

\ If an offset was specified, the log/antilog tables must now be redone:
M0 #IF
\ this replaces the b^i antilog table (offset 0)  with the A^i table (offset M0)
CREATE NANTILOG MASK CELLS ALLOT
: FILL-NANTILOG
   MASK 0 DO I >ANTILOG M0 FF^  I CELLS NANTILOG + ! LOOP ;
 FILL-NANTILOG
 NANTILOG ANTILOG MASK 1+ CELLS CMOVE \ replace the antilog table
 FILL-LOG
  #THEN
```

Listing Two

```
\ Utilities to display finite field tables
\ Utilities only work for symbol size 8 bits or less.

: H.R ( n1 n2--)  \ Hex .r
  BASE @ >R HEX .r r> BASE ! ;

: .TOPLINE
   CR 4 SPACES
   MASK 1+ 0 DO I 2 H.R SPACE LOOP
   CR 4 SPACES
   MASK 1+ 0 DO ." ===" LOOP  ;
```

```
: +TABLE
   CR ." ADDITION TABLE    "
   .TOPLINE
   MASK 1+ 0 DO
    CR I 2 H.R 2 SPACES
   MASK 1+ 0 DO I J FF+ 2 H.R 1 SPACES LOOP
    LOOP CR CR
  ;

: *TABLE
   CR ." MULTIPLICATION TABLE    "
   .TOPLINE
   MASK 1+ 0 DO
    CR I 2 H.R 2 SPACES
   MASK 1+ 0 DO I J FF* 2 H.R 1 SPACES LOOP
    LOOP CR CR
  ;

: /TABLE
   CR ." DIVISION TABLE    "
   CR ." NUMERATOR:"
   .TOPLINE
   MASK 1+ 0 DO
    CR I 2 H.R 2 SPACES
   MASK 1+ 0 DO I J DUP 0<> IF FF/ 2 H.R ELSE 2DROP ."  U" THEN 1 SPACES LOOP
    LOOP
    CR ."  ^DENOMINATOR" CR
   ;

: ANTILOG-TABLE
     CR ." ANTILOG TABLE"
     .TOPLINE
     CR 4 SPACES
   MASK 0 DO
    I >ANTILOG 2 H.R SPACE LOOP
    ."  U " CR CR ;

: LOG-TABLE
     CR ." LOG TABLE "
     .TOPLINE
     CR ."      U "
   MASK 1+  1 DO
     I >LOG 2 H.R SPACE LOOP CR CR ;

: TABLES.
    CR  ." POLYNOMIAL (LESS TOP TERM)=" POLYNOMIAL 2 H.R
    ."  MASK=" MASK 2 H.R
    CR ." U=Undefined"
    CR CR
   ANTILOG-TABLE
   LOG-TABLE
   +TABLE
   *TABLE
   /TABLE ;
```

SPONSORS & BENEFACTORS

The following are corporate sponsors and individual benefactors whose generous donations are helping, beyond the basic membership levels, to further the work of *Forth Dimensions* and the Forth Interest Group. For information about participating in this program, please contact the FIG office (office@forth.org).

Corporate Sponsors

AM Research, Inc. specializes in Embedded Control applications using the language Forth. Over 75 microcontrollers are supported in three families, 8051, 6811 and 8xC16x with both hardware and software. We supply development packages, do applications and turnkey manufacturing.

Clarity Development, Inc. (http://www.clarity-dev.com) provides consulting, project management, systems integration, training, and seminars. We specialize in intranet applications of Object technologies, and also provide project auditing services aimed at venture capitalists who need to protect their investments. Many of our systems have employed compact Forth-like engines to implement run-time logic.

Computer Solutions, Ltd. (COMSOL to its friends) is Europe's premier supplier of embedded microprocessor development tools. Users and developers for 18 years, COMSOL pioneered Forth under operating systems, and developed the groundbreaking chipFORTH host/target environment. Our consultancy projects range from single chip to one system with 7000 linked processors. www.computer-solutions.co.uk.

Digalog Corp. (www.digalog.com) has supplied control and instrumentation hardware and software products, systems, and services for the automotive and aerospace testing industry for over 20 years. The real-time software for these products is Forth based. Digalog has offices in Ventura CA, Detroit MI, Chicago IL, Richmond VA, and Brighton UK.

Forth Engineering has collected Forth experience since 1980. We now concentrate on research and evolution of the Forth principle of programming and provide Holon, a new generation of Forth cross-development systems. Forth Engineering, Meggen/Lucerne, Switzerland - http://www.holonforth.com.

FORTH, Inc. has provided high-performance software and services for real-time applications since 1973. Today, companies in banking, aerospace, and embedded systems use our powerful Forth systems for Windows, DOS, Macs, and micro-controllers. Current developments include token-based architectures, (e.g., Open Firmware, Europay's Open Terminal Architecture), advanced cross-compilers, and industrial control systems.

The iTV Corporation is a vertically integrated computer company developing low-cost components and information appliances for the consumer marketplace. iTVc supports the Forth development community. The iTVc processor instruction set is based on Forth primitives, and most development tools, system, and application code are written in Forth.

Keycorp (www.keycorp.com.au) develops innovative hardware and software solutions for electronic transactions and banking systems, and smart cards including GSM Subscriber Identification Modules (SIMs). Keycorp is also a leading developer of multi-application smart card operating systems such as the Forth-based OSSCA and MULTOS.

www.kernelforth.com

An interactive programming environment for writing Windows NT and Windows 95 kernel mode device drivers in Forth.

MicroProcessor Engineering supplies development tools and consultancy for real-time programming on PCs and embedded systems. An emphasis on research has led to a range of modern Forth systems including ProForth for Windows, cross-compilers for a wide range of CPUs, and the portable binary system that is the basis of the Europay Open Terminal Architecture. http://www.mpeltd.demon.co.uk

www.theforthsource.com

Silicon Composers (web site address www.silcomp.com) sells single-board computers using the 16-bit RXT 2000 and the 32-bit SC32 Forth chips for standalone, PC plug-in, and VME-based operation. Each SBC comes with Forth development software. Our SBCs are designed for use in embedded control, data acquisition, and computation-intense control applications.

T-Recursive Technology specializes in contract development of hardware and software for embedded microprocessor systems. From concept, through hardware design, prototyping, and software implementation, "doing more with less" is our goal. We also develop tools for the embedded marketplace and, on occasion, special-purpose software where "small" and "fast" are crucial.

Tateno Dennou, Inc. was founded in 1989, and is located in Omecity Tokyo. Our business is consulting, developing, and reselling products by importing from the U.S.A. Our main field is DSP and high-speed digital.

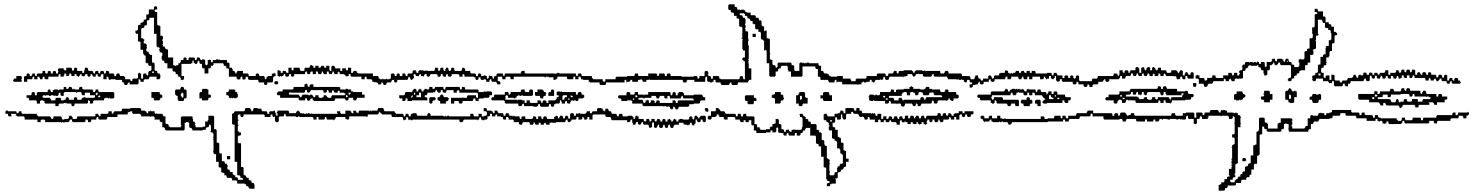
ASO Bldg., 5-955 Baigo, Ome,Tokyo 198-0063 Japan
+81-428-77-7000 • Fax: +81-428-77-7002
http://www.dsp-tdi.com • E-mail: sales@dsp-tdi.com

Taygeta Scientific Incorporated specializes in scientific software: data analysis, distributed and parallel software design, and signal processing. TSI also has expertise in embedded systems, TCP/IP protocols and custom applications, WWW and FTP services, and robotics. Taygeta Scientific Incoporated • 1340 Munras Avenue, Suite 314 • Monterey, CA 93940 • 408-641-0645, fax 408-641-0647 • http://www.taygeta.com

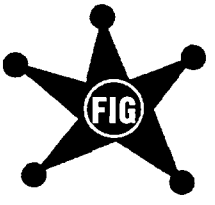
Triangle Digital Services Ltd.—Manufacturer of Industrial Embedded Forth Computers, we offer solutions to low-power, portable data logging, CAN and control applications. Optimised performance, yet ever-increasing functionality of our 16-bit TDS2020 computer and add-on boards offer versatility. Exceptional hardware and software support to developers make us the choice of the professional.

Individual Benefactors

Makoto Akaishi
Everett F. Carter, Jr.
Edward W. Falat
Michael Frain
Guy Grotke
Bjorn Gruenwald
John D. Hall
Guy Kelly
Zvie Liberman
Marty McGowan
Gary S. Nemeth
Marlin Ouverson
John Phillips
Thomas A. Scally
Werner Thie
Richard C. Wagner



WANTED



BY THE FORTH INTEREST GROUP

"Silicon Slick" (an alias)

...and any and all Forth programmers and other SOFTWARE RENEGADES roaming the range in pioneer territories...

...to write articles about their DISCOVERIES & TECHNIQUES, PERILOUS MISADVENTURES, and MYSTIFYING ENCOUNTERS with STRANGE CHARACTERS and with FORTH FEATURES obvious and subtle.

REWARD

To recognize and reward authors of Forth-related articles, the Forth Interest Group (FIG) has adopted the following Author Recognition Program.

Articles

The author of any Forth-related article published in a periodical or in the proceedings of a non-Forth conference is awarded one year's membership in the Forth Interest Group, subject to these conditions:

a. The membership awarded is for the membership year following the one during which the article was published.
b. Only one membership per person is awarded in any year, regardless of the number of articles the person published in that year.
c. The article's length must be one page or more in the magazine in which it appeared.
d. The author must submit the printed article (photocopies are accepted) to the Forth Interest Group, including identification of the magazine and issue in which it appeared, within sixty days of publication. In return, the author will be sent a coupon good for the following year's membership.
e. If the original article was published in a language other than English, the article must be accompanied by an Engish translation or summary.

The fastest, most convenient way for us to receive your material is via e-mail (a vast improvement over the telegraph, a.k.a "talking wire") to the editor@forth.org address. Binary (e.g., formatted text) files must be uuencoded to be sent as e-mail, but ASCII files can be sent as-is.

Letters to the Editor

Letters to the editor are, in effect, short articles, and so deserve recognition. The author of a Forth-related letter to an editor published in any magazine except *Forth Dimensions* is awarded $10 credit toward FIG membership dues, subject to these conditions:

a. The credit applies only to membership dues for the membership year following the one in which the letter was published.
b. The maximum award in any year to one person will not exceed the full cost of the FIG membership dues for the following year.
c. The author must submit to the Forth Interest Group a photocopy of the printed letter, including identification of the magazine and issue in which it appeared, within sixty days of publication. A coupon worth $10 toward the following year's membership will then be sent to the author.
d. If the original letter was published in a language other than English, the letter must be accompanied by an English translation or summary.