$10 Volume XIX, Number 2 July 1997 August

FORTH DIMENSIONS

Linked Lists & Simple Macros

Token System for Payment Terminals

Yet Another Forth Objects Package

Adaptive Digital Filters

Pygmy Implementation of Kermit

Call for Papers: FORML Conference

New FORML dates...
...the week **before** *Thanksgiving!*

The original technical conference for professional Forth programmers and users.

19th annual FORML Forth Modification Conference
November 21 – 23, 1997

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California USA

THEME:

"Forth at the Millennium"

What are the challenges for Forth as we reach the Millennium? Will the year 2000 present problems for existing programs? Many organizations are asking for certification that software will work perfectly as we move to 2000 and beyond.

How will certification be accomplished? Encryption is required for more applications to keep transactions private. Proposals for incorporating encryption techniques are needed for current and future applications. Your ideas, expectations, and solutions for the coming Millennium are sought for this conference.

FORML is the perfect forum to present and discuss your Forth proposals and experiences with Forth professionals. As always, papers on any Forth-related topic are welcome.

Abstracts are due October 1, 1996 • Completed papers are due November 1, 1997

Mail abstract(s) of approximately 100 words to:
FORML, Forth Interest Group • 100 Dolores Street, Suite 183 • Carmel, California 93923
or send them via e-mail to FORML@forth.org

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific Ocean beach. Registration includes use of conference facilities, deluxe rooms, meals, and nightly wine and cheese parties.

Guy Kelly, Conference Chairman • **Robert Reiling, Conference Director**

Advance Registration Required—Call Today!

Conference attendee in double room $460
Non-conference guest in same room $330
Conference attendee in single room $600

Under 18 years old in same room $190
Infants under 2 years in same room – free
FIG members and guests eligible for 10% discount

phone: 408-373-6784
fax: 408-373-2845
e-mail: FORML@forth.org

FORML, Forth Interest Group
100 Dolores Street, Suite 183
Carmel, California 93923

The FORML Conference is sponsored by the Forth Interest Group.

DEPARTMENTS

EDITORIAL

Vote With Your Feet

Errata

With much fanfare in our preceding issue, we welcomed a new advertiser whose services will be of interest to many of our readers. Unfortunately, during a last-minute production frenzy as we shuffled the pagination, the ad was accidentally dropped. We offer our apologies to Kevin Martin—whose ad *does* appear in this issue—and to any readers who may have been inconvenienced.

FORML

Once again, we remind everyone that the annual FORML Conference is *not* being held on the U.S. holiday of Thanksgiving this year. (See the ad on our inside front cover.) We may seem to be belaboring the point but, for many years Forth folk from this country who have family obligations found themselves unable to attend (at least, not without some guilt). The fact of FORML during holidays became engrained in our collective consciousness, a background irritant for those who couldn't be there for that reason. Many people requested a change and, this time, the organizers were able to accommodate those requests.

We hope you will vote with your feet this year (or with your frequent flyer mileage), and show your support of the new dates with your attendance at this remarkable gathering. So much goes on at FORML that it is difficult to report in writing about it. Suffice it to say, for now, that the marvellous location is surpassed only by the conference itself.

Of course, we enjoy cross-pollination with everyone who is doing interesting new work in Forth so, if you happen to be attending EuroForth (see the ad on our back cover), we'd welcome more news about European activities! (And a hearty thank you to the European authors who have been writing lately—your contribution and influence is welcome.)

Corporate membership program

One of the relatively new ways to benefit from membership in the Forth Interest Group is via a *corporate membership*. Some companies have taken advantage of the added benefits of this level of membership, and some who previously were individual members have converted to the new program. We will be listing our corporate members in the next issue; if you are unfamiliar with the program, please contact the FIG office, who will be more than happy to explain the program to you.

Thanks for the articles, now write more...

The response to our recent request for articles has been gratifying. Thanks to those who responded, we have good material on hand for the next issue, and promises of more. Material is pledged which will interest Forth experts *and* relative beginners.

To my dismay, when I entered the editorial arena (too many years ago to discuss), I found that, just as we produced a fine issue and were ready to celebrate, another issue's deadlines were looming and advance plans had to be made for the one after that. Only with a consistent and reliable source of material can we produce the kind of publication that will serve our readers and the public, and that will represent the exciting Forth work that is being done out there.

So even while we prepare to publish the material in the next issue, and to thank you again for the contributions, we must reiterate that your articles and code are more than welcome—they are needed! And don't forget that, even in this time of on-line communications, substantive letters to the editor are also meaningful ways to contribute.

Besides, we like to get mail. Can we look forward to hearing from you?

Marlin Ouverson
editor@forth.org

Forth Dimensions
Volume XIX, Number 2
July 1997 August

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 ($53 Canada/Mexico, $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:
408-37-FORTH Fax: 408-373-2845

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 $45/53/60 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.

OFFICE NEWS

No sooner had we finished the bulk mailing of the May/June issue of *Forth Dimensions* than the July/August issue was ready to go! We're all working hard to get everything back on schedule after the move. Soon, perhaps we'll be thinking of Carmel as the home of the Forth Interest Group, rather than its "moved to" location. I'm thinking this will happen by the time of FORML! After all, the conference is the last "first" for us here at the office. After spending the first part of this year processing your memberships and orders, I hope I get to meet a good number of you at FORML.

I was able to do just that at this year's 17th annual Rochester Forth Conference, sponsored by the Institute for Applied Forth Research, Inc. This was the first year I attended, so I didn't know what to expect. It was great fun! To have the opportunity to meet and talk to the people behind the articles in *Forth Dimensions*, and to put faces to the names we've been hearing in the main offices, added a touch of humanity to the workload.

A heartfelt thank you goes to both Larry and Brenda Forsley—they are excellent hosts who go out of their way to make the Rochester Forth Conference comfortable for all who attend. If you haven't previously attended, you'll definitely want to make it a date for next year. What a wonderful opportunity it is for everyone in the Forth community to have two conferences available for you to attend.

FORML, the West Coast conference, is coming up! It's not too early to get your abstract, or even the title of your talk, to Guy Kelly, this year's Conference Chair. The e-mail site FORML@forth.org has been set up for that purpose, as well as for pre-registration and other inquiries you might have. We look forward to hearing from you!

As you no doubt know by now, the date has been changed to the week *before* the U.S. Thanksgiving—to November 21 through 23. The reason for doing this was to make it more convenient for people to attend. If you haven't been to the Monterey Peninsula during this time of year... let me tell you, it's beautiful. The weather consists mostly of bright blue skies, temperatures in the 60's... warm enough during the day that a favorite sweater or jacket is appropriate while walking around town or on the beach, yet cool enough in the evening that a fire in the fireplaces of the meeting rooms adds just the right touch to the atmosphere.

If you are wondering whether you should come as a family, please do; there are many things for families on the Peninsula. The Monterey Bay Aquarium is one of the finest in the world, and it has just finished construction of a new wing. There is a Natural History Museum in Pacific Grove that's for everyone, and it has the added benefit of being free. A new Children's Science Museum, which as of this date we haven't been to (but that will change in the next month) has just opened. The Monterey Museum of Modern Art has lovely exhibits. There are also a dozen or so great parks and playgrounds.

And, of course, there's Carmel. You can wander the quaint, winding streets and visit the Barnyard shopping center. The Peninsula also sports many brand-name outlet stores. So, if you are looking for a special gift for someone, you'll probably find it here. Or, if you'd like some local flavor, how about visiting one of the numerous winery tasting rooms? If you like good wine, this region of California, like Napa, has vineyards producing world-class wines.

Guaranteed, if you bring the family, they won't be bored! We've lived here for eight years and, even though our hearts are still rooted in New England (and, with it, the desire to move back someday), the Monterey Peninsula is a great place to visit.

More office business: new chapters are coming on board, and at the moment work is focused on revising and updating our FIG Chapter Kit. Those of you who have expressed interest: we will be getting back to you this next month. If you haven't contacted the office and you're interested in starting a chapter, now is a great time to begin. We now have Corporate Memberships, too. Our Corporate Members should have their listings of services and products in the next issue of *Forth Dimensions*.

The one thing you can count on these days is that things are constantly changing at the home office. And now is a great time to be part of that change!

Cheers 'til next time,
Trace Carter

Trace Carter • Monterey, California
office@forth.org

The final accolade:

ISO/IEC Forth

On 15th April 1997, ISO/IEC 15145 "Information technology—Programming Languages—Forth" was published. Such an event is a coming of age for Forth. Surely the world has got to take it seriously now.

In large part, ISO15145 adopts the ANS Forth document. The initial proposal was that ISO/IEC should adopt the ANS Forth standard on-block in a Fast Track procedure. The SC 22 Secretariat forwarded the final text to ITTF for publication on November 27, 1996. In the disposition of comments document (SC 22 N 2343), the Secretariat states, "The editorial comments from the Netherlands and the ISO Central Secretariat have been incorporated in the revised DIS.

"The comments from the United Kingdom concerning internationalization issues and requirements for embedded systems programmed in Forth will be addressed when the TC reconvenes in 1998. At that time, we will consider these and other needs that have arisen in this evolving technology."

So what has changed between the two documents?

On opening the ISO/IEC version and ANS versions side by side, there is an obvious difference in style for the first few sections. The contents list reflects the major differences, which all seem, at first look, to be confined to sections 1 and 2.

The ISO/IEC document's introduction provides a brief history of Forth and its progress towards international standardisation. This, in effect, replaces the very brief Purpose paragraph in the ANS document.

This shifts the remaining parts of section 1 up, although the content of each of these sections remain unchanged. The references have moved in the ISO document to section 1, but also remain unchanged apart from reordering of the listed standards. Section 2 remains unchanged, apart from the loss of the references section and the numbering of individual term definitions in section 2.1. In all other respects ISO/IEC 15145:1997(E) is the same as ANS X3.215–1994.

With the publication of the ISO/IEC Forth standard, Forth can be said to have become a truly international programming environment. What more could we want, other than that the same might happen with the Open Firmware standard, as well.

Document Reference: ISO/IEC 15145:1997(E) First Edition 1997–04–15 (ICS 35.060) 210 Pages.
Title: *Information technology – Programming languages – Forth.*

The cost of the standard in the U.K. from British Standards Institute is UKP 153.00 (non-member price, members pay approximately 50% of this).

Paul E. Bennett • peb@transcontech.co.uk
www.tcontec.demon.co.uk

ANS X3.215-1994	ISO/IEC 15145:1997(E)
	Introduction
1. Introduction	1. General
1.1 Purpose	1.1 Scope
1.2 Scope	1.1.1 Inclusions
1.2.1 Inclusions	1.1.2 Exclusions
1.2.2 Exclusions	1.2 Document organization
1.3 Document organization	1.2.1 Word sets
1.3.1 Word sets	1.2.2 Annexes
1.3.2 Annexes	1.3 Future Directions
1.4 Future directions	1.3.1 New technology
1.4.1 New technology	1.3.2 Obsolescent features
1.4.2 Obsolescent features	1.4 Normative references
2. Terms, notation and refences	2. Terms, notation and refences
2.1 Definitions of terms	2.1 Definitions of terms
2.2 Notation	2.2 Notation
2.2.1 Numeric notation	2.2.1 Numeric notation
2.2.2 Stack notation	2.2.2 Stack notation
2.2.3 Parsed-text notation	2.2.3 Parsed-text notation
2.2.4 Glossary notation	2.2.4 Glossary notation
2.3 References	

euroFORTH Conference — see back cover
EuroForth:
http://cis.paisley.ac.uk/forth/euro/index.html

EuroForth'97:
http://cis.paisley.ac.uk/forth/euro/ef97.html

A Platform-Independent Token System for Payment Terminals

Europay, the major European credit card organization (Eurocard, MasterCard in Europe, and other financial products) is developing technology to support smart cards—Integrated Circuit Cards, or ICCs—as the credit cards of the future. This will require new software in all credit card terminals, which range from 8051-based POS terminals to high-end ATMs. To facilitate this transition, they have designed a token-based system—conceptually similar to Open Firmware or Java—which is based on Forth. Using this Open Terminal Architecture (OTA), it will be possible for credit card issuers and acquirers to write application programs that will be completely platform independent and which will run on all OTA-compliant kernels.

The project has been under way for two years. FORTH, Inc. and MPE, Ltd. have been principal members of the design and development team, along with Europay. Prototype terminals were exhibited at a major Europay banking conference in June 1996, and production systems have been operating in the field in Prague, Czech Republic, since May 1997.

Background

Modern payment applications are moving to ICC technology. ICCs can significantly improve security of payment transactions by being able to manage encrypted account data offline, by participating actively in the transaction-validation process, and by being intrinsically extremely difficult to violate or reproduce. They can also contain code to enhance the transaction processing, thereby providing new opportunities for payment products and services.

Use of this new technology, however, will necessitate altering the firmware in several million terminals that will use the ICCs. To facilitate this transition, Europay is designing a standardized software system that will be compact, efficient, and easy to maintain and enhance for future payment system needs. This is the Open Terminal Architecture system.

OTA defines a software virtual machine standardized across all terminal types, described in detail in *Europay Open Terminal Architecture Specification Volume 1: Virtual Machine Specification.*[1] This virtual machine provides drivers for the terminal's I/O and all low-level CPU-specific logical and arithmetic functions. An extensive repertoire of commands specific to the needs of ICC terminals is also provided, with functions such as commands for managing databases, different languages, security algorithms, and the special data formats used by the cards. High-level libraries, terminal programs, and payment applications using standard kernel functions may be developed and compiled into token modules. These must be certified once; thereafter, they will run on any conforming terminal of the appropriate type (for example, ATM or POS) without change, regardless of the terminal's CPU type or other architectural issues. Therefore, a significant consequence of OTA is a simplified and uniform set of test and certification procedures for all terminal functions.

To provide a common means of distributing programs in a compact, standard, machine-readable form, OTA uses a token system that is, in some respects, similar to Java byte-codes. An OTA token compiler converts source code to a string of tokens that is extremely compact (and therefore easy to transmit over phone lines or to read from an ICC), and is also easy for even simple processors to interpret with minimal overhead.

To summarize, OTA provides the following major benefits:

- A virtual machine with generalized ICC support functions, to be installed in each terminal only once. The kernel lifetime is expected to match that of the terminal (7–10 years).
- Terminal kernel certification independent of applications, so certification only needs to be done once for each terminal type. A terminal type is defined as a specific configuration of terminal CPU and I/O functions.
- Application certification procedures that are independent of the terminal on which the application will run, since all terminals provide the same virtual machine interface. Only one certification and validation is needed for tokenized software libraries, terminal programs, and payment applications, providing they run on certified OTA terminals.
- Standard downloading procedure for all terminal types, using compact token modules for minimum transmission time.
- Support for tokenized code on an ICC, to make maximum use of its storage capabilities and to minimize communications time between card and terminal.

OTA is based on Forth, extended with commands to facilitate development of payment applications. Forth was chosen by Europay because, of all standard interpretive-type languages, it provides the most compact and efficient means of representing both terminal programs and the code that may reside on the ICC itself. Compactness in terminal programs translates directly into reduced transmission time and cost for terminal updates, and compactness in ICC code results in increased capability and reduced transfer time between card and terminal.

For security reasons, OTA allows only run-time behavior in a terminal, so the virtual machine includes only a run-time subset of ANS Forth.

Both Forth and C compilers have been developed to support OTA tokens. VM implementations, applications, and libraries have been developed in both Forth and C.

Open Terminal Architecture Features

The specific characteristics of the architecture were designed and optimized for both compact and reasonably fast execution of typical payment functions on a wide variety of

---

[1] Version 2.2, January 29, 1997. Available from Europay Documentation Centre, 198A Chaussée de Tervuren, 1410 Waterloo, Belgium.

Elizabeth D. Rather
erather@forth.com
Manhattan Beach, CA

Stephen Pelc
sfp@mpeltd.demon.co.uk
Southampton, England

Peter Johannes
pjo@europay.com
Waterloo, Belgium

CPUs. Many design decisions were heavily influenced by the extreme need for program security in payment terminals.

Virtual Machine CPU

The OTA virtual machine is based on a multi-stack architecture, as seen in Figure One. This architecture, derived from Forth, has been further modified for portability, code density, ease of compilation, and for use with other programming languages. For example, it contains frame memory for local variables used in C. Thus, OTA token compilers can be written not only for Forth, but also for C and other languages.

The VM is a byte-addressed, 32-bit machine, with 32-bit registers and stack elements. Despite some initial trepidation about implementing a 32-bit VM on processors such as the 8051, we have found ways to do so with remarkably good run-time performance.

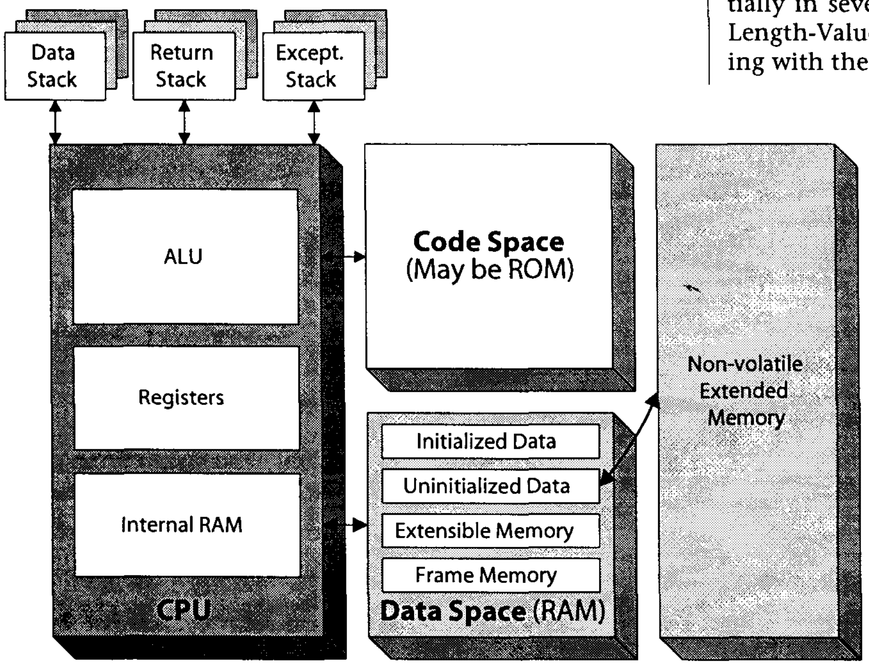
Memory

OTA defines a single address space for programs. This address space is accessible for data storage only. Programs may not assume that executable code is in this address space. Depending on the actual processor, and on the mechanism used to convert the token image into executable tokens, the executable code may be in a different address space (shown as code space in Figure One), or may be under the control of a memory management unit. In any case, programs are not permitted to access their own program memory directly, and any attempt to do so will be flagged during the program certification procedure.

Addressable memory is further divided into sections:

- Initialized data space may be preset at compile time to values that will be instantiated in the target at run time.
- Uninitialized data space will be preset to binary zeroes in the target at run time.
- Extensible memory is temporarily allocated using a rubber-band memory allocation algorithm.
- Frame memory is used by C stack frames and Forth local variables.

Figure One. The OTA Virtual Machine



In addition to directly addressable memory, the VM also manages extended memory, which is not directly available to token programs. This is used for two purposes: databases and module storage. Databases are managed by the VM as a server to client token programs. Clients may select a named database and records within that database. At any time, the client has access to a current record in a current database via named fields of various types. The module storage is not accessible by token programs directly, although modules may call functions in external modules in ways managed internally by the token interpreter in the VM. Certain tokens also support high-level management of the module storage by allowing terminal programs to add and delete modules, with appropriate security.

Programs and Tokens

The OTA token set provides program portability across multiple CPU types by passing source code programs of various types through a compiler whose output is a string of OTA tokens, which may be thought of as machine instructions for the OTA virtual machine. The tokens are organized into a module, which consists of a header, a section representing the module's data items, lists of imported and exported functions (providing links to other modules), and the tokens themselves. Target terminals then process this code by instantiating the data space associated with the module, linking the module's imports to functions exported by other modules, and finally interpreting the tokens. Figure Two illustrates this process.

The OTA token set covers three main areas. The first is the instruction set of a theoretical processor (the virtual machine), which provides the instructions necessary for the efficient execution of programs. The second supports I/O and communications functions. The third group consists of OTA-specific functions such as databases, a message database (potentially in several European languages), and support for Tag-Length-Value data formats (ISO 8825) used for communicating with the ICCs and for other data communications.

The OTA token set has been optimized for use on small terminals, with ease of compilation, ease of interpretation, and good code density. The most common functions, including most Forth primitives, are expressed in one-byte, or primary, tokens. Less frequently used functions are two-byte, or secondary, tokens. Some tokens also have associated values, for such things as literal values and branch offsets.

System Components

The purpose of OTA is to provide software to run in terminals used in payment applications. Conceptually, there are two hardware environments, and several classes of software. The hardware environments include the development system, which is based on a simple PC; and a target, which is some form of payment terminal. The entire suite of software includes:

- development software, which runs on the PC and is available in two packages, for VM and application development, respectively;
- virtual machine implementations, which include all platform-specific software in a terminal and other mandatory standard functions;
- libraries, which provide general functions to support terminal programs and payment applications;
- applications, which are the functions specific to a particular payment product;
- terminal programs, which perform general non-payment terminal functions and include high-level mechanisms for selecting and executing transactions and associated applications; and
- test suites and platforms, for both VM implementations and token programs.

Terminal Target Environments

The target system is any one of a large variety of payment terminals. Actual products range from small, hand-held devices with simple, eight-bit microprocessors (such as the 8031/51 family), to 32-bit computers running operating systems such as Windows NT. In order to simplify the production, certification, and maintenance of software on such a wide variety of targets, OTA terminal code is based on a single virtual machine. The VM consists of a standardized set of functions whose CPU-specific implementation is optimized for that specific platform. Implementations currently operating in the field on eight different devices show that this approach provides good run-time performance, even on 8051 CPUs.

Virtual Machine

The OTA VM has standard characteristics that define addressing modes, stack usage, register usage, address space, etc. The virtual machine concept makes a high degree of standardization possible across widely varying CPU types, and simplifies program portability, testing, and certification issues.

The VM instruction set includes a selected subset of ANS Forth commands, plus a number of specialized OTA functions, such as terminal I/O support and token loader/interpreter support. Since it cannot itself be tokenized, and may reside in PROM, the VM is intended to be installed once, and not changed thereafter during the lifetime of the terminal. Therefore, its functions are carefully designed to be very general in nature and as complete as possible, in order to support a wide range of present and future terminal programs and applications.

Terminal manufacturers are responsible for providing a VM implementation on their terminals. This VM is developed and certified according to the OTA Virtual Machine Specification. Standard kernel functions not appropriate to a particular terminal type (e.g., the cash dispenser function on a POS terminal) are coded as null functions for that terminal, so every kernel has an identical set of functions and the testing and certification process is simplified. These null functions add very little to system overhead and complexity, and their advantage far outweighs their cost.

The terminal's VM supports standard libraries and terminal programs and applications, which are written in high-level code for the virtual machine and are delivered as token modules, which will run on any standard VM.

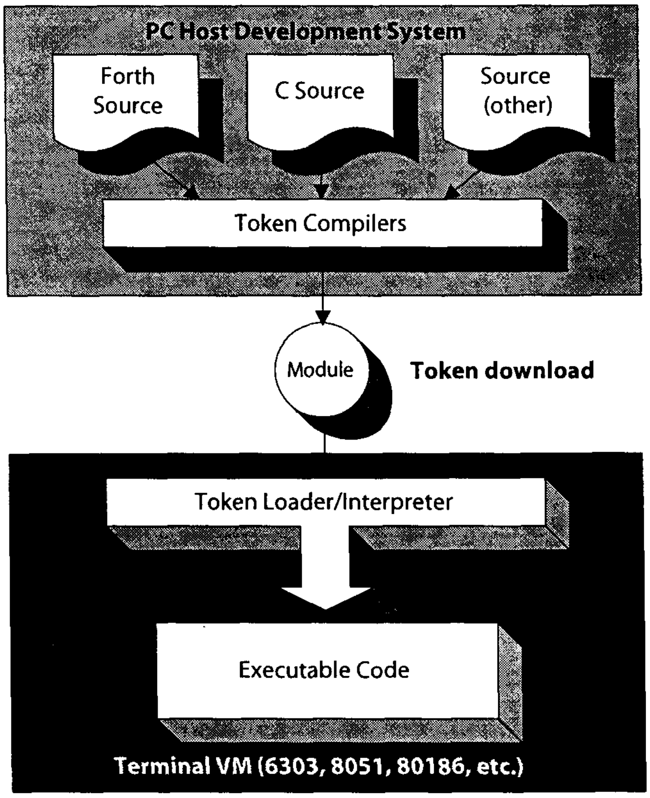
Libraries

OTA libraries contain higher-level functions that support common features of terminal programs, such as language selection, and common features of applications, such as PIN verification. A terminal may contain several libraries, some accessible to all applications, and some restricted to particular applications or payment systems. Libraries are written and tokenized for the virtual machine, using functions provided in the kernel, and therefore can be run on any terminal.

Terminal Program

A terminal program consists of the high-level personality characteristic of this terminal type (POS, ATM, etc.). This includes the functions common to all transactions (e.g., card initialization and language selection), as well as the user interface required to select an application and process a transaction. The terminal program, at the highest level, is typically triggered by a card insertion. A terminal program is written for the virtual machine and is supplied in token form. It can, therefore, be run on any terminal of the appropriate type, and is easily changed by downloading over a network at any time. However, it frequently does incorporate platform-specific features, such as customized greeting messages, knowledge of particular screen-management capabilities, etc. These features are not included in the VM, as they may change on a time-scale shorter than the design lifetime of the VM. For example, a particular make of terminal may be used by a number of different merchants, each of which may request customized user menus.

Figure Two. Tokens may be generated from a variety of source formats, downloaded to a terminal, and then converted into executable code for that terminal by any of several different methods.



Applications

A terminal transaction will select an application as part of its processing flow. Applications fall into three general areas: stored value system (such as Europay's CLIP system, VISA CASH, or Mondex), debit cards, and credit cards; applications generally will vary in their method of processing a given transaction. Versions of these applications may be provided by different payment systems and may be further customized by individual issuers, acquirers, or even individual merchants (such as large chains or department stores). Applications are supplied in token form via the communications path and, if security considerations permit, may be enhanced by token programs on an ICC.

The Token Compiler and Token Loader/Interpreter

Libraries, applications, and terminal programs are written in high-level code for the virtual machine. The OTA development system includes a special compiler for this virtual machine, whose output consists of tokens. Tokens may be thought of as machine language instructions for the virtual machine. Tokens are either one or two bytes in length, and therefore represent the program in a form that is both CPU-independent and extremely compact (far more so, for example, than compressed source text).

Each OTA virtual machine contains a token interpreter (TLI), which processes a stream of tokens into an executable form. Once the kernel is installed in a terminal, the libraries, applications, and terminal programs can be downloaded into the terminal in a variety of ways (direct connection to an OTA development host, acquirer network, modem and dial-up telephone line, ICC, etc.). Program modifications and entire new applications may be downloaded in the same manner whenever needed. The VM implementation is designed to be so general purpose in nature that a wide range of present and future terminal programs and applications can be accommodated without modifications to the VM.

ICC Functions

One function of ICCs is to improve transaction security by incorporating and managing encrypted data and participating actively in the transaction-validation process. It is a natural feature of OTA to go beyond these functions and to provide for ICCs that also contain program code to enhance a terminal's transaction processing, thereby providing new opportunities for payment products and services. To facilitate this, a few sockets have been provided that can be plugged by issuer-specific functions such as loyalty programs, which may be invoked at appropriate points in the transaction processing. Europay does not currently propose that ICCs contain entire applications, but only plug behaviors that enhance existing terminal applications.

As far as security is concerned, the presumption is that if an ICC passes the decryption and data authentication tests performed by the terminal program, whatever functions and plug behaviors are on the card have been certified and are syntactically valid. The terminal decides to allow or disallow the card's proposed actions only as controlled by the terminal access security functions.

Development Environments

An OTA development system is used to develop terminal software, either low-level VM implementations or high-level library or application software. Kernel development requires a target terminal to be connected, as the kernel is cross-compiled on the PC host and downloaded to the terminal across the Interactive Development Link. OTA libraries, terminal programs, and applications are also developed on the PC host. Because they are high-level code, they may also be executed on the host for preliminary testing, using a PC version of the standard kernel.

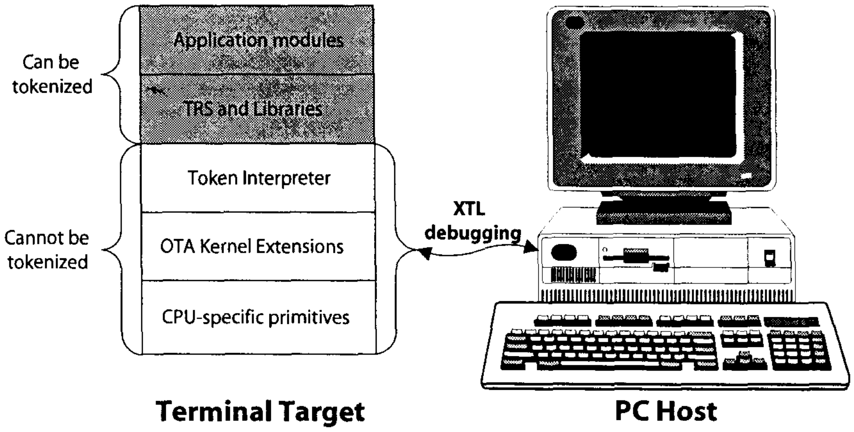
Since the requirements for developing and testing the VM implementation and high-level token modules differ significantly, two different tool chains have been developed: the Kernel Development Kit (KDK) for terminal VM implementors, and the Application Development Kit (ADK) for token program developers.

VM implementation tools

The KDK consists of a Windows-based cross-compiler and Interactive Development Environment (IDE), a test terminal consisting of a CPU, keypad, LCD display, serial ports, and memory sufficient to run a typical ICC application supplied in token form as a demo, and a VM implementation for that test terminal hardware (see Figure Three). Development software on the PC is based on ProForth for Windows, a product of MicroProcessor Engineering Ltd.

The VM implementation for the test terminal is supplied in source form, with documentation intended to support a developer porting this to a specific terminal of the same CPU type. As many terminal vendors have previously developed BIOS or OS functions (typically in C or assembler), a protocol is included that facilitates the use of this software to provide a defined set of functions (e.g., I/O functions) required by the VM. The kit also includes a terminal test suite and demo

Figure Three. A typical OTA development environment for a terminal. A small program to support the terminal end of the Cross-Target Link (XTL) protocol is included in the target during development.



application for testing the VM as it develops.

Token module development tools

There are two major components to the Application Development Kit. These are the token compiler itself, and the Token Interactive Debugging Environment (TIDE). The latter is a standard VM implementation on a Windows platform, with configuration options that enable it to simulate a wide variety of potential target terminals. A developer may use this platform to test token modules, regardless of the language in which they were written. Token compilers are available for Forth and C. The ADK also includes an optimizer, which performs a variety of post-processing functions on token modules, that can reduce a module's size by on the order of 50%. The result is a module that runs with no performance penalty; indeed, it is usually significantly faster, since there are fewer tokens to process.

Testing and validation tools

VM test suite development has progressed in parallel with VM development, and test suites are presently delivered with all KDKs. The VM test suite consists of a set of modules that exercise individual tokens, checking each token's behavior against expected results and producing a report summarizing tests passed and failed (if any). The main test platform for token modules is TIDE, although tools are also in development to provide additional test facilities.

Formal validation tools are still in development.

Project Status

The Open Terminal Architecture (OTA) is a complete system for supporting ICC-based payment terminals of the future. Its design incorporates features that will facilitate development and certification of a new, standardized kernel for all payment terminals; will support development and certification of platform-independent libraries and terminal programs; and will enable code on the ICCs themselves to provide enhanced payment products for issuers.

Extensive effort has been directed toward seeing that both code and procedures are included to ensure program security and integrity. A complete set of development tools is available through Europay to support not only Forth, the language upon which OTA is based, but also C. Reference kernels, libraries, and applications are also available.

Elizabeth D. Rather was the world's second Forth programmer. She is President of FORTH, Inc., and a member of the Board of Directors of the Forth Interest Group.

Stephen Pelc's company, Microprocessor Engineering Ltd., has been selling Forth-based hardware and software since the early 1980s.

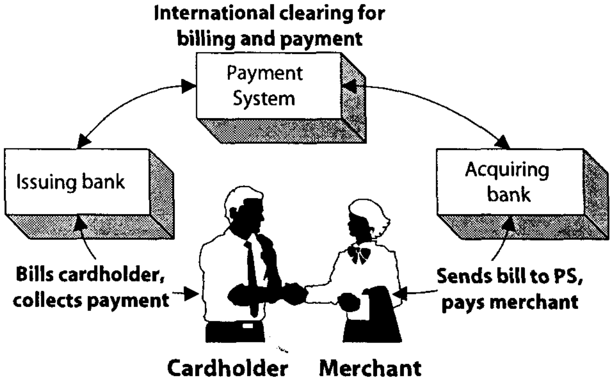
Peter Johannes has been the project manager for the Open Terminal Architecture since its inception in 1994.

Background:

Payment Systems in Europe

Banking-based payment systems in Europe are organized nationally for domestic banking products, and internationally to facilitate international consumer payments. Looking at a typical payment transaction, the parties involved are the merchant, the acquiring bank, the payment system, and the issuer. Each of these is a distinct role, although it need not be played by a different physical or corporate entity.

- Issuers provide consumers with banking products, in this context as debit, credit, or stored-value cards. These cards can still be embossed only (for paper-based transactions), with magnetic stripe (today's state-of-the-art) or with an IC. IC cards are currently used widely only in France, although pilot projects are under way in several other countries.
- Acquirers hold the commercial relationship with the merchant. They acquire the transactions, meaning they pay the merchant for the goods and route the transactions to wherever the consumer's account is.
- Merchants accept cards, and provide goods to the cardholder. Merchants are more and more convinced of the necessity to migrate toward electronic payment services (e.g., magstripe or ICC).
- Payment systems such as Europay have two primary roles: to provide specifications (and associated certification) for international payment products, and to arbitrate member conflicts. In addition, payment systems also typically provide network and security services.



An IC card (or ICC) is an active device that stores data in such a way that it can prove their authenticity. It is also capable of generating *authenticity certificates* for transactions. This means that a point-of-sale terminal now has to deal with an active device that performs complex operations. The terminal software must behave correctly, whichever ICC is used.

Current ICCs carry about 200 times the amount of information that can be present on the magstripe cards. This amount is expected to increase as ICCs get loaded with more services. The drawback is that terminal programs become more complex and bigger. Terminals typically download their software from a server via a 1200-baud communication link. The cost of this management and download is becoming prohibitive, especially in Europe.

Many existing ICC-based payment systems are "closed" (proprietary) systems. This means it is very hard to introduce new applications without getting the consent of all the parties involved. Even then, extensive changes to the software on every terminal are required, and the terminals must be recertified. With OTA, Europay is attempting to provide an open system, capable of handling multiple ICC products, card types, and systems on a single terminal with minimal changes.

—P. Johannes, Europay International

A Simple Implementation of the Kermit Protocol in Pygmy Forth

http://www.eskimo.com/~pygmy/forth/kermit.zip

Kermit in Pygmy

Frank Sergeant
pygmy@pobox.com

scr # 13000

```
KERMIT.SCR
  Contains a simple implementation of the Kermit
  file transfer protocol.

  copyright 1997 Frank Sergeant               pygmy@pobox.com
                 809 W. San Antonio St.
                 San Marcos, TX  78666

  This source code is not Public Domain or Shareware.
  You may use it freely for any private or commercial purpose
  provided you do so at your own risk.
```

scr # 13001

```
( load screen    Kermit file transfer protocol)

For the algorithm, see pp 98-113 of

_C Programmer's Guide to Serial Communications_
by Joe Campbell, Howard W. Sams & Company, 1987,
ISBN 0-672-22584-0.

Note, there are some errors in Campbell's examples.
```

scr # 13002

```
( KERMIT)

GET-Y/N   Wait for the user to press a y, n, Y, or N key.
          Return true if y or Y.  Return false if n or N.


TRY-AGAIN?
          Display a message and ask whether the user wants
          to try again.  E.g.
          " Drive not ready" TRY-AGAIN? IF ... THEN

.MSG      clears the top 2 lines of the screen and displays a
          message, leaving the cursor positioned just past the
          message.  E.g.  " Starting the transfer ..." .MSG
```

scr # 13003

```
( KERMIT)
MYMAXL maximum "len" we are willing to handle.
       The transmitted LEN field includes SEQ, TYPE, DATA, CKSUM
       fields.  94 maximum allowed under basic Kermit.  Our
       buffers must be 1 byte longer to hold the LEN field also.
OUT-BUF & IN-BUF buffers for building outgoing or receiving
```

file KERMIT.SCR

scr # 12000

```
KERMIT.SCR
  Contains a simple implementation of the Kermit
  file transfer protocol.

  copyright 1997 Frank Sergeant               pygmy@pobox.com
                 809 W. San Antonio St.
                 San Marcos, TX  78666

  This source code is not Public Domain or Shareware.
  You may use it freely for any private or commercial purpose
  provided you do so at your own risk.
```

scr # 12001

```
( load screen    Kermit file transfer protocol)

 "  *** Simple Kermit file transfer protocol Copyright (c) 1997
Frank Sergeant (pygmy@pobox.com) *** "  DROP


12002 12024 THRU
```

scr # 12002

```
( KERMIT - user interface)

: GET-Y/N ( - f)
  BEGIN KEY DUP 'Y = OVER 'y = OR IF DROP -1 EXIT THEN
            DUP 'N = SWAP 'n = OR IF       0 EXIT THEN  BEEP
  AGAIN  ;

: TRY-AGAIN? ( a - f)
  CR COUNT TYPE CR ." Try again? (Y/N) "
  GET-Y/N  ( f)  ;

: .MSG ( a -) 0 0 AT 160 SPACES  0 0 AT COUNT TYPE  ;
```

scr # 12003

```
( KERMIT)
1 CONSTANT SOH
VARIABLE SEQ    SEQ OFF
: BUMPSEQ ( -) SEQ @ 1+ 63 AND SEQ !  ;
  94  ( 35) CONSTANT MYMAXL ( fields SEQ TYPE DATA & CKSUM)
```

```
       incoming frames.  We store LEN, SEQ, TYPE, DATA, CKSUM
       fields, but not the SOH nor the ending CR.
OUTLEN & INLEN count bytes currently in the buffers
MAXL   holds agreed-upon maximum "len" value, which is
       the MIN of receiver's and sender's preferences.
```

a "character-ized" number is produced by adding a "space." The result must be <= $7E, thus the original number must be <= $5E (ie 94 decimal).

```
CREATE OUT-BUF  MYMAXL ( 1+) 2 + ALLOT     VARIABLE OUTLEN
CREATE  IN-BUF  MYMAXL ( 1+) 2 + ALLOT     VARIABLE INLEN

: ?NUM-OK ( n -) $5E > ABORT" too big"  ;
: CHAR ( n - c) DUP ?NUM-OK ( n) $20 +  ;
: UNCHAR ( c - n) $20 - ( n) DUP ?NUM-OK  ;
```

scr # 13004

(KERMIT)

MAXL, QCTL, etc are the agreed-upon protocol parameters for the session. INIT-LIMITS initializes these to the values we would prefer to use. The sender and receiver exchange an S-frame and an ACK-frame listing their preferences. We then compromise by taking the MIN between them.

scr # 12004

```
( KERMIT  - protocol parameters)
VARIABLE MAXL
VARIABLE QCTL
VARIABLE NPAD
VARIABLE PADC
VARIABLE EOLC
VARIABLE TIMEOUT

: INIT-LIMITS ( -)
  MYMAXL MAXL !  ( maximum "len" value)
      '# QCTL !  ( control code escape character)
       0 NPAD !  ( number of pad characters)
       0 PADC !  ( pad character)
     $0D EOLC !  ( end of line character)
    4 TIMEOUT !  ( timeout in seconds)     ;   INIT-LIMITS
```

scr # 13005

(KERMIT)

We make >LEN, >TYPE, etc relative to the start of the buffer so we can use the same definitions for both the receiving and sending buffers. >CKSUM assumes the LEN byte has been initialized.

scr # 12005

```
( KERMIT  - address of fields in buffers)
: FIELD: ( offset -) ( buff - a) CREATE C,   DOES> C@ +  ;

0 FIELD: >LEN
1 FIELD: >SEQ
2 FIELD: >TYPE
3 FIELD: >DATA
: >CKSUM ( buff - a) >LEN DUP C@ UNCHAR  +  ;

3 FIELD: >MAXL
4 FIELD: >TIME
5 FIELD: >NPAD
6 FIELD: >PADC
7 FIELD: >EOLC
8 FIELD: >QCTL
```

scr # 13006

(KERMIT - compromise on the parameters)

COMPROMISE assumes we have an S frame in one buffer and its ACK frame in the other buffer. We don't care whether we are the sender or receiver. The compromise takes the more conservative setting from each buffer as the actual protocol parameter to use.

For now, we will ignore all the settings except for MAXL and TIMEOUT, taking the MIN of MAXL and the MAX of TIMEOUT.

scr # 12006

```
( KERMIT - compromise on the parameters)

: COMPROMISE ( -)
  OUT-BUF IN-BUF ( a a)
  OVER >MAXL C@ UNCHAR  OVER >MAXL C@ UNCHAR
      MIN  MAXL ! ( a a)
  OVER >TIME C@ UNCHAR  OVER >TIME C@ UNCHAR
      MAX  TIMEOUT ! ( a a)    2DROP  ;
```

scr # 13007

```
MYMENU    cheap error handling in the case where the user
          chooses to abort the file transfer process.  Set up
          your own menu ( ' MYREALMENU IS MYMENU ) or allow the
          default 'no vector' error to occur.

KSER-IN   gets a serial character and tests whether it is SOH,
          all the while checking for a time-out.  Returns
          character and SOH-flag (true if character is SOH).
          In case of time out, return up an extra level,
          putting a 'V on the stack as the dummy frame type
          indicating a time out followed by a true flag
          indicating a 'good' check sum.
          Note, KSER-IN is only called by GETFRAME and so is
          always called with the correct stack depth.  To test
          it standalone, nest it once in a test word, as shown
          in TEST-IN.
```

scr # 13008

```
( KERMIT)
We "controlify" a control code (0-$1F, $7F) by flipping bit 6
and preceding it with the QCTL character (usually '#).  The
QCTL character itself is escaped.  We count QCTL as a control
character in CTRL? so we can escape it, but we only flip bit
6 for actual control characters.  Also, consider $7E (~) to
be a control character, as it is used for repeat counts

(KEMIT puts a character into OUT-BUF and increments the count
KEMIT writes a character into OUT-BUF, escaping it if necessary.
ROOM? says whether there is room in the buffer for another
      character.  We require 2 bytes available in case the
      next character needs to be escaped.  If we allowed
      high-bit escpaping we would require 3 bytes instead.
```

scr # 13009

```
( KERMIT)
CK%%  converts the raw checksum of all the bytes
      after SOH into a checksum character by wrapping
      and character-izing it according to the KERMIT algorithm.

CKSUM  calculates a checksum on a buffer by adding the bytes
       in the LEN SEQ TYPE & DATA fields and applying CK%%.
       The LEN field must include the cksum byte.

CKSUM? Calculate the checksum character for the input frame
       and compare it to the transmitted checksum character.
       Return true if the checksum is good.
```

scr # 13010

```
MODEM! sends a character to the modem.  We defer it to make
      testing easy.

DATA! builds an entire data field, stopping either when out
      of source characters or out of room in OUT-BUF.
```

scr # 12007

```
DEFER MYMENU
: KSER-IN ( - c f)
  TIMEOUT @ 1000 * ( ms)
  BEGIN  KEY? IF KEY DROP CR
                    ." Abort file transfer (Y/N)? "  GET-Y/N CR
                    IF ." Transfer aborted -- press "
                       ." any key to return to menu"
                       KEY DROP MYMENU
                    ELSE ." Transfer continuing "
              THEN  THEN
    SER-IN? IF ( ms) DROP SER-IN DUP SOH = ( c f) EXIT THEN
    ( ms) 1-  DUP 0= IF POP 2DROP 'V  -1 EXIT THEN     1 MS
  AGAIN  ;

: TEST-IN ( - c f)  KSER-IN  ;
```

scr # 12008

```
( KERMIT )
: CTRL ( c - c')
  DUP QCTL @ = OVER '~ = OR IF EXIT THEN  $40 XOR  ;

: CTRL?  ( c - f)
  DUP $20 < OVER QCTL @ = OR OVER $7E = OR SWAP $7F = OR  ;

: (KEMIT ( c -) OUT-BUF  OUTLEN @ + C! (  ) 1 OUTLEN +!  ;

: KEMIT   ( c -)  PAUSE ( just in case)
  ( c) DUP CTRL? IF  QCTL @ (KEMIT  CTRL  ( c) THEN (KEMIT  ;

: ROOM?  ( - u) MAXL @ 1- OUTLEN @  >  ;
```

scr # 12009

```
( KERMIT )
: CK%% ( u - c)
  DUP $C0 AND 2/ 2/ 2/ 2/ 2/ 2/ + $3F AND CHAR ;

: CKSUM ( buffer - c) >LEN DUP C@ UNCHAR ( a #) 0 ROT ROT
  FOR ( sum a) C@+ +UNDER NEXT DROP  CK%% ( c)  ;

: CKSUM?  ( - f)
  IN-BUF CKSUM ( c)  IN-BUF >CKSUM C@ ( c c)  =  ;
```

scr # 12010

```
DEFER MODEM!
( ' EMIT)  ' SER-OUT IS MODEM!

: DATA! ( a # - a' #')    SWAP   ( # a)
  BEGIN ( # a) OVER 0= ROOM? 0= OR ( ie out of source or room)
```

```
( a #) ;
BUILD-FRAME  Given the address and length of data to be
      transferred and the type of the frame, put as much of
      the data as will fit into a frame and return the address
      and length of the remaining (i.e. unsent) data.
```

scr # 13011

```
( KERMIT - debugging aids)

.FRAME .INB .OUTB  are used for testing to dump the contents
U.R
                    of the buffers to the screen.

TEST1 TEST2  provide some test data



.FRAME THEN ;
```

scr # 13012

```
( KERMIT)

SENDFRAME sends an entire header, from SOH through 1-byte
UNCHAR 1+
      checksum and ending carriage return, to the "modem."
      It sends SOH, sends LEN+1 characters in the OUT-BUF,
      and then sends a carriage return.
```

scr # 13013

```
( KERMIT)

LIMITS provides data for use in building either an S-frame
       or its ACK frame for purposes of negotiating
       the protocol as to maximum frame length, etc.
work)
       Note that PADC is controlified, but seems not to
       be "escaped" -- after all, we haven't agreed upon
       the escape character at the time of sending the
       S-frame.  We build this frame directly into OUT-BUF
       to prevent DATA! from escaping any characters.
       We say we'll use (~) as the repeat character, but we
       will _not_ use repeat counts when we transmit, but we
       _will_ handle them when we receive.  If the sender does
       not escape actual tildes, then we will have a problem.
```

```
   NOT WHILE ( # a) C@+ KEMIT  -1 +UNDER  REPEAT SWAP

: BUILD-FRAME ( a # type - a' #')   OUTLEN OFF
  0 ( ie dummy len) CHAR (KEMIT    SEQ @ CHAR (KEMIT
  (KEMIT ( a #) DATA! ( a' #')
  OUTLEN @ CHAR  OUT-BUF >LEN C!  ( a #)
  OUT-BUF CKSUM  OUT-BUF >CKSUM C! ( a #)
```

scr # 12011

```
( KERMIT - debugging aids)

: .FRAME ( buf -) ." len = "  C@+ UNCHAR DUP PUSH 2

  ."  seq = " C@+ UNCHAR 2 U.R SPACE SPACE
  ." myseq = " SEQ @ 2 U.R SPACE SPACE
  POP 1- TYPE  CR  ;

: .INB ( type -) .S 3 SPACES
   'V = IF ." V-frame "  CR ELSE ."  IN: " IN-BUF


: .OUTB ( -) .S 3 SPACES ." OUT: " OUT-BUF .FRAME  ;

" WHAT DOES THE SYMBOL # STAND FOR?" CONSTANT TEST1

" as much labor for the study of its" CONSTANT TEST2
```

scr # 12012

```
( KERMIT)

: SENDFRAME ( -) SOH MODEM!  OUT-BUF >LEN DUP C@

 FOR ( a) C@+ MODEM!  NEXT DROP (  )  $0D MODEM!  ;
```

scr # 12013

```
( KERMIT)
: LIMITS ( type -)
  SEQ OFF  PUSH
  '~  ( the repeat char)
  '1  ( 1-byte chksum, either '1 or 1 CHAR seems to

  'N  ( no hi-bit prefix)
  QCTL @           EOLC @ CHAR  PADC @ CTRL
  NPAD @ CHAR  TIMEOUT @ CHAR  MAXL @ CHAR    POP
  SEQ @ CHAR   12 ( len) CHAR
  OUT-BUF  12 FOR DUP PUSH C!  POP 1+ NEXT DROP  (  )
  OUT-BUF CKSUM  OUT-BUF >CKSUM C!  ;
```

scr # 13014

```
( KERMIT)
KINIT sends the 'send-init' frame.  It must have sequence zero.
      This is the 'S' frame sent by sender in response to the
      receiver's initial NAKs.
KINITACK sends a reply to a 'send-init' frame.  Before sending
      KINITACK (if we are receiving) or after receiving
      KINITACK (if we are sending), we must adjust our settings
      to the minimum of the sender's and the receiver's requests
      Note complex handling of COMPROMISE.
FILEHEADER sends the file name of the file to be transmitted.

EOF is sent at the end of each file we send.  EOT is sent after
we finish sending all the files.  Receiver sends ACK or NAK
after each frame is received, depending on whether chksum is
ok.  ERROR is sent to abandon the session.  I think we will
ignore an ATTRIB frame.
```

scr # 12014

```
( KERMIT)
: BUILD/SEND   ( a # type -) BUILD-FRAME  SENDFRAME 2DROP  ;
: KINIT ( -) 'S LIMITS  SENDFRAME  ;
: KINITACK ( -) 'Y LIMITS COMPROMISE 'Y LIMITS SENDFRAME  ;
: FILEHEADER ( a # -) " sending file " .MSG 2DUP TYPE
  ( a #)   'F BUILD/SEND  ;
: EMPTY-FRAME ( type -) (  ) CREATE C,
  DOES> C@ 0 0 ROT BUILD/SEND  ;

'Y EMPTY-FRAME (ACK      'N EMPTY-FRAME (NAK
'Z EMPTY-FRAME EOF       'B EMPTY-FRAME EOT
'A EMPTY-FRAME ATTRIB    'E EMPTY-FRAME ERROR

: ACK ( seq# -) SEQ @ SWAP SEQ ! (ACK  SEQ !  ;
: NAK ( seq# -) SEQ @ SWAP SEQ ! (NAK  SEQ !  ;
```

scr # 13015

```
( KERMIT)

EXPECTED   holds the count of bytes we expect to receive
           following the length byte.


SETLENGTH  handles the length count for an incoming frame,
           initializing EXPECTED and INLEN and putting the
           length byte into the input buffer.

PUT-IN-BUFFER  puts input bytes into the buffer and returns
               a flag that is true when all the expected bytes
               have arrived.
```

scr # 12015

```
( KERMIT)

VARIABLE EXPECTED

: INBUF! ( c -)  IN-BUF INLEN @ + C!  1 INLEN +!  ;

: SETLENGTH ( clength -)
  INLEN OFF  DUP INBUF!  ( c)   UNCHAR  EXPECTED !  ;

: PUT-IN-BUFFER ( c - f)  INBUF! INLEN @ EXPECTED @ > ;
```

scr # 13016

```
GETFRAME is closely tied to KSER-IN and is the only word that
 should ever call KSER-IN, as KSER-IN returns upward an extra
 level in case of a timeout, supplying the type and cksum flag
 (ie 'V -1).  So, GETFRAME always succeeds, returning a type
 and flag.  It watches for an SOH in the middle of a frame and
 starts over.  What makes GETFRAME tricky is it needs to handle
 the usual case as well as a timeout at any time as well as an
 unexpected SOH at any time.  What makes it simpler is pushing
 some of the logic down to the word KSER-IN and letting KSER-IN
 terminate not only itself but also GETFRAME in the case of a
 timeout, thus producing a dummy V-frame.  After that we no
 longer have a timeout as a special case, we simply have an
 additional "frame" type (i.e. a timeout frame).
```

scr # 12016

```
( KERMIT)

: GETFRAME ( -  type f)
  BEGIN KSER-IN NIP UNTIL (  )  ( ie await SOH)
  BEGIN
     BEGIN KSER-IN WHILE DROP REPEAT ( c) SETLENGTH (  )
     BEGIN KSER-IN NOT WHILE ( c) PUT-IN-BUFFER ( f)
      IF  IN-BUF >TYPE C@ CKSUM?
          OVER 'E = OVER AND ABORT" Fatal Error in Kermit"
          EXIT THEN (  )
     REPEAT ( c) DROP
  AGAIN  ( type f)  ;
```

scr # 13017

```
( KERMIT)

GET-GOOD-FRAME  continues to try to get a frame until one
                arrives with a good checksum.  It will try
                forever unless the user aborts the transfer.
                (See KSER-IN for test for user abort.)


IN-SEQ     sequence number of the frame in the input buffer

GOOD-SEQ?  true if the input frame's sequence number is the
           expected sequence number.
```

scr # 12017

```
( KERMIT)

: GET-GOOD-FRAME ( - type)
  BEGIN  GETFRAME ( type cksumflag) NOT WHILE
       ."  bad cksum " DROP  REPEAT  ;


: IN-SEQ ( - u)  IN-BUF >SEQ C@ UNCHAR  ;

: GOOD-SEQ? ( - f)  IN-SEQ SEQ @ =  ;
```

scr # 13018

```
( KERMIT)

(GETACK keeps getting frames until one comes in with a good
        checksum.  V-frames are ok.

GETACK keeps getting ack frames, handling or ignoring each, as
       appropriate.  It re-sends the data frame in case of a
       V-frame (timeout) or a NAK with the correct sequence
       number.  It is used only by the sender.  Later, it
       could bail out if too many NAKs or timeouts occur in a
       row, etc.

READ  load up the buffer from the file in preparation for
      transmitting it via the serial port
```

scr # 12018

```
( KERMIT)
: (GETACK ( - type)
  BEGIN GETFRAME ( type f) NOT WHILE DROP REPEAT ( type)  ;

: GETACK ( -)
  BEGIN   (GETACK  ( type)
    'Y OF  GOOD-SEQ? ( f) DUP IF BUMPSEQ THEN    ( f)  ELSE
    'N OF  GOOD-SEQ? IF SENDFRAME THEN              0  ELSE
    'V OF  SENDFRAME                                0  ELSE
    ( default) DROP  0 [  3 ] THENS ( f)
  UNTIL  ;

: READ ( h - a #) PUSH 32767 BUFFER ( ie dummy buffer)
  DUP 1024 POP FILE-READ #BYTES-READ @  ;
```

scr # 13019

```
( KERMIT)

GET-FIRST-NAK ignores timeouts and sequence numbers and waits
      for a NAK from the receiver.

SEND  wait for the prompting NAK frame from receiver
      send S-frame ( ie KINIT)
      reset serial in to throw away any extra prompting NAKs
      get S-frame ACK for SEQ 0
      send the entire file, one D-frame at a time
      close the file
      send end of file and end of transmission
```

scr # 12019

```
( KERMIT)

: GET-FIRST-NAK ( -) BEGIN (GETACK 'N = UNTIL  ;

: SEND ( name -) CLS " Waiting to send " .MSG INIT-LIMITS
  DUP FOPEN IF CR ." cannot open input file" CR  EXIT THEN
  ( name h) 1000 MS  ( name h)   GET-FIRST-NAK
  ( n h)  KINIT RESET-SER-IN  GETACK
  COMPROMISE  SWAP COUNT ( h a #) FILEHEADER ( h) GETACK
  BEGIN ( h) DUP READ DUP WHILE ( h a #)
   BEGIN 'D BUILD-FRAME SENDFRAME GETACK  '. EMIT
   DUP 0= UNTIL 2DROP
  REPEAT 2DROP ( h) FCLOSE (   ) EMPTY-BUFFERS ( just in case)
  EOF GETACK  EOT GETACK  ;
```

scr # 13020

```
( KERMIT)

IN-DATA  is a buffer for holding the UNCTRL'd data field.  Make
         it big in case lots of repeat counts are present.

C!+ stores a character and bumps the address (similar to C@+)
```

scr # 12020

```
( KERMIT)

CREATE IN-DATA    MYMAXL 3 / 94 * 2 +  ALLOT

: C!+ ( c a - a+) DUP PUSH C! POP 1+  ;

: C@+- ( fr # - fr # c) 1- PUSH C@+ POP SWAP  ;
```

```
C@+-   gets a character from the 'from' address, increments
       the 'from' address and decrements the count of remaining
       characters.

UNCTRL'd if the current character is the QCTL escape character,
         get another character and unescape it.


 scr # 13021

REPEAT'd The most recent character was the tilde (~), indicating
         the beginning of a 3 or 4 character repeat sequence.
         Get the next character as the count and then the next 1
         or 2 (if escaped) to find the value to be repeated, &
         expand that repeated character into destination buffer.

UNCTRL copy the escaped and repeated source buffer,
       unescaping and expanding as appropriate, to the
       destination buffer.

>IN-DATA  copies IN-BUF's data field, which may contain
          escaped characters, to IN-DATA with escaped characters
          converted to their actual values (and repeated counts
          expanded).


 scr # 13022
( KERMIT)

BUILDFNAME extracts name of file to be received from an
           input F frame and stores it in our KFNAME buffer
           as a counted string (and an asciiz string suitable
           for passing to DOS for creating the file).

RCVNAME  this is what we do in response to an F-frame:
         save the file name in the KFNAME buffer as
         a counted, asciiz string, then create the file and
         save the handle.


 scr # 13023
( KERMIT)

GET-NEXT  Get the next frame we are expecting, ACKing or NAKing
          as appropriate.
          Always ack with the seq number we received, even if
          it wasn't the seq number we expected, thus allowing
          sender to continue.  But, throw away frames that
```

```
: UNCTRL'd ( from # c - from # c)
  DUP QCTL @ - IF EXIT THEN    DROP C@+- CTRL   ;



scr # 12021
( KERMIT)
: REPEAT'd ( to from # - to from #)  ROT PUSH ( fr #)
  C@+- UNCHAR PUSH C@+- ( fr # c) UNCTRL'd ( fr # c)
  POP POP ( ie rpt# to) 2DUP + PUSH ( fr # c rpt# to)
  SWAP ROT FILL ( fr #) POP ROT ROT ( to fr #)  ;

: UNCTRL ( from to # - a #)
  ROT PUSH PUSH DUP POP POP SWAP ( to to from #)
  BEGIN DUP WHILE ( to to fr #)
      C@+- DUP '~ = IF ( to to fr # c) DROP REPEAT'd
      ELSE UNCTRL'd PUSH ROT POP SWAP C!+ ROT ROT  THEN
  REPEAT ( to to fr 0) 2DROP  OVER -  ( a #)  ;

: >IN-DATA ( - a #) IN-BUF >DATA IN-DATA ( from to)
  IN-BUF C@ UNCHAR 3 - ( from to #) UNCTRL ;


scr # 12022
( KERMIT)
VARIABLE KHANDLE
CREATE KFNAME 50 ALLOT

: BUILDFNAME ( -)
  >IN-DATA ( a #) DUP PUSH KFNAME 1+ SWAP CMOVE (  )
  0 KFNAME R@ + 1+ C!  ( make name into an asciiz string)
  POP KFNAME C!   ;

: RCVNAME ( -)
  BUILDFNAME   KFNAME FMAKE ( h f)
  ABORT" cannot open output file" ( h) KHANDLE !
  " receiving file " .MSG KFNAME COUNT TYPE SPACE  ;


scr # 12023
( KERMIT)
: GETNEXT ( - type)
  BEGIN GETFRAME ( type f)
     IF ( type) DUP 'V =
        IF ( type) SEQ @ NAK    -1  ( type f)
        ELSE ( type) IN-SEQ DUP ACK ( type seq) SEQ @ -
        THEN
```

```
          do not have the expected seq number.  Except, if
          V-frame (ie timeout) or if a bad checksum, then
          NAK with our expected sequence number.
          It is possible a D-frame should not be ACK'd until
          after we have written it to disk, in case disk writes
          interfere with servicing the serial port.

WRITE     Append input data to the file.

 scr # 13024
( KERMIT)

RECEIVE  send NAK every second until we see SOH, then get
         the rest of that first frame -- until we get the
         S-frame.  Then compromise on settings and send
         an ack for the S-frame.  Then, handle the frame
         types, getting file name and opening it for an
         F-frame, writing D-frames to the file, closing
         the file upon getting a Z-frame, and exiting upon
         getting a B-frame (EOT).
```

```
      ELSE
         ( ie bad cksum)
         SEQ @ NAK  -1  ( type f)  ( -1 ABORT"  BAD CKSUM" )
      THEN
     WHILE DROP
    REPEAT   BUMPSEQ  ;

: WRITE ( -) >IN-DATA KHANDLE @ ( a # h) FILE-WRITE   ;

scr # 12024
( KERMIT)

: RECEIVE ( -) CLS  " Waiting to receive " .MSG
  RESET-SER-IN    INIT-LIMITS
  BEGIN 0 NAK  1000 MS  GET-GOOD-FRAME  'S = UNTIL
  (  ) KINITACK  BUMPSEQ
  BEGIN  GETNEXT ( type) ( DUP EMIT )
     'D OF  WRITE  '. EMIT        0  ELSE
     'F OF  RCVNAME               0  ELSE
     'Z OF  KHANDLE @ FCLOSE      0  ELSE
     'B OF                       -1  ELSE
     ( otherwise)  DROP           0 [  4 ]  THENS
  UNTIL  (  )  ;
```

FREEWARE & SHAREWARE

Win32Forth 3.5 has been released to the community, and is available on the Forth Interest Group compilers page at:
ftp://ftp.taygeta.com/pub/Forth/Win32For/w32for35.exe

The Forth Interest Group's home page is located at:
http://www.forth.org/fig.html

The primary changes to Win32Forth include enhancements to the WinView editor that allow editing multiple files at the same time, and a more intelligent screen-refresh algorithm. The class library has been generalized, and now supports more control classes.

Be sure to get the latest update file numbered like 32UPDTxx.EXE, where *xx* is a number 01, 02, etc. Intermediate update files will not be needed, each successive update will include all the previous updates.

3.5 will be the last public-domain release of Win32Forth, making this the basis for future user work. Bug fixes and minor revisions will be released periodically, in the form of an update to the base system.

I would like to give special thanks to all the industrious Forthers who have contributed bug reports, fixes, and examples to this project.

—Tom Zimmer

Pygmy 1.5 is now on my web site for your downloading convenience. Get the file pygmy15.zip on my Forth page at:
http://www.eskimo.com/~pygmy

Version 1.5 has minor typo and documentation clean-ups, plus *multitasking and the ability to be embedded in a C wrapper* in order to access C library routines, give C access to Forth routines, an implementation of the Kermit file-transfer protocol, etc.

I have updated the bonus disk, too, available only via uuencoded e-mail. Previous bonus disk customers, please e-mail me your preferred e-mail address, and I'll send the *updated bonus disk free.* (Potential bonus disk customers, see details in the pygmy15.zip file.) The bonus disk now includes the updated 68HC11 (cross) assembler and updated serial port routines, both interrupt-driven and multitasked polling.

If anyone does not have web access to get pygmy15.zip, e-mail me and I will be glad to send it as uuencoded e-mail.

My **"3-instruction Forth" for the 68HC11** is now on my Forth web page). It includes the original article from the 1991 FORML Conference Proceedings, with variants of the code customized for several different versions of the 68HC11 (i.e., 'E1, 'A1, 'D0).

The article describes how to use a single-board computer with Forth on a host computer. Examples include code for Pygmy on the PC and a minimal "3-instruction Forth" monitor downloaded to the 68HC11's RAM. For development, this gives nearly the full convenience of a full Forth on the target, but using somewhere between 32 bytes (the smallest I have used) to 66 bytes of RAM on the 68HC11.

This may be of use even to those who do not use Forth to develop for the 'HC11, as it provides a convenient wrapper for writing, running, and testing assembly language routines on the 'HC11. The package is freely distributable and usable, under the condition that I bear no responsibility for any damages due to its use or misuse.

The **Bare Bones EPROM Programmer** materials are also on my web site. This includes the executable file, the schematic and printed circuit board artwork (as .bmp files), and instructions for building and using the programmer. It doesn't include source. The programmer uses the MC68HC11D0P and the 4049 or 74HC4049, and is controlled from a PC serial link. The programmer only programs 2kx8, 8kx8, 16kx8, and 32kx8 EPROMs (and probably EEPROMs). I am making this old project available for "historical" interest.

—Frank Sergeant • pygmy@pobox.com

STRETCHING STANDARD FORTH - #15

Linked Lists

A *linked list* is a *list* (i.e., sequence) of cells, called *nodes*, such that each node somehow contains the address of the next node. The address of the first node of the list is contained in a cell, the *head* of the list. The last node contains a programmer-chosen value to indicate the end of the list. This value is usually zero, and that's what we'll use here.

When the list is empty, the head contains the zero.

The nodes themselves don't contain any information. Information is associated by a programmer-chosen convention to be physically near its node.

The information could be anything: a number, an execution token, a string, a node or head of another list, or whatever. More than one item of information may be associated with a node.

In the applications that interest us now, the primary information will be kept just after the node, and will be a counted string. Use `CELL+` to go from a node to its information.

To get the address of the next node from a node, use `node@`. To set the contents of a node, use `node!`.

These can be defined:

```
: node@ @ ;
: node! ! ;
```

We use `node@` and `node!` so we can change the definition if we want.

To create a new list, we define its head.

```
: LIST: ( "<spaces>name" -- )
  CREATE 0 , ;
( name Execution: -- node )
```

To create a new node, which will be put in a list:

```
: New-Node ( -- node )
  ALIGN  HERE 0 ,  ;
```

Information should go into dataspace just after the new node.

To put a node, new or old, into a list [see Figure One]. This puts the node at the beginning of the list. To put a node at the end of the list [see Figure Two].

Any node in a list can be considered to be the head of a list. So to put a node after a node in a list, also use `link-node`.

All nodes in a list should be unique. We'll ensure this by only putting newly created nodes in a list, or moving a node from one list to another.

Information doesn't have to be unique, but generally will be.

[See Figure Three.]

`STRING` is from *Starting Forth*. [See Figure Four.]

To remove information from a list, remove its node.

Sometimes we'll want to remove a whole bunch of information that was recently added. We do this by removing all nodes within a range of addresses. Use `DUP CELL+` as the range to remove a single node.

Space is not recovered by this. That must be done some other way. [See Figure Five.]

Continued on page 27

Figure One

```
: link-node                        ( node list -- )
    OVER 0=
    ABORT" link-node: Mustn't link NIL node. "
    2DUP node@ SWAP node! node!
;
```

Figure Two

```
: append-node                      ( node list -- )
    BEGIN  DUP node@ WHILE  node@  REPEAT  node!
;
```

Figure Three

```
( Create a list )

LIST: A-List

( A-List --> 0 )

( Put "Larry" on the list. )

New-Node A-List link-node  ," Larry"

( A-List --> Larry --> 0 )

New-Node A-List link-node  ," Curly"

( A-List --> Curly --> Larry --> 0 )

A-List node@  New-Node SWAP link-node  ," Moe"

( A-List --> Curly --> Moe --> Larry --> 0 )
```

Figure Four

```
: PLACE ( caddr len addr -- )
  2DUP 2>R  CHAR+ SWAP  CHARS MOVE  2R> C!  ;
: STRING ( char "ccc<char>" -- )
  WORD COUNT  HERE  OVER 1+ ALLOT  PLACE  ;
: ," [CHAR] " STRING ;
```

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

WIL BADEN is a professional programmer with an interest in Forth. For a copy of the source for this article, send e-mail requesting *Stretching Forth #15: Linked Lists.*

Figure Five

```
(  cull-nodes    Remove nodes that are within a range. )

: cull-nodes    3 needed        ( lower upper list -- )
    ROT ROT 2>R              ( node) ( R: lower upper )
        BEGIN                       ( node)
            DUP node@ ?DUP
        WHILE
            DUP 2R@ WITHIN IF
                node@ OVER node!
            ELSE
                NIP
            THEN
        REPEAT                      DROP
    2R> 2DROP
;
```

Figure Six

```
: .text ( caddr len -- ) TYPE  SPACE ;
: ?text ( addr -- ) ?DUP IF  COUNT .text  THEN ;
: BEGINS-WITH ( a1 u1 a2 u2 -- a1 u1 flag ) DUP >R 2OVER R> MIN
COMPARE 0= ;
```

Figure Seven

```
: items ( list -- ) ITEM> ( item) ?text ( ) ITERATE ;
```

Figure Eight

```
: #items ( list -- n ) 0 SWAP ITEM> DROP  1+  ITERATE ;
```

Figure Nine

```
:: Some-List ITEM>
    COUNT S" ANS" BEGINS-WITH IF  .text  ELSE  2DROP  THEN
ITERATE ;;
```

Figure Ten

```
( item-search )

    : ?node?     ( flag node -- node 0 | node' node' | 0 0 )
        SWAP IF
            FALSE     ( node 0)
        ELSE
            node@ DUP ( node' node' | 0 0 )
        THEN
    ;

MACRO ITEM>   " >R FALSE BEGIN R> ?node? WHILE DUP >R CELL+ "

MACRO ITERATE " FALSE REPEAT DROP "
```

THE VIEW FROM GOAT HILL - #02

The Search Paradigm

Standard Forth provides a basic pattern for the general problem of searching. [Figure One]

When components are empty, `0= WHILE REPEAT` can be replaced by `UNTIL`, and `ELSE THEN` by `THEN`.

By writing `test-for-more` and `test-for-match` appropriately, the Search Paradigm can be written. [Figure Two]

This can be made strictly structured. [Figure Three]

Now make a simple definition and macro. [Figure Four] See "Tool Belt #2" in this issue for `MACRO`.

This special instance of the Search Paradigm can now be written:

```
SPECIAL-SEARCH test-for-match? REPEAT
IF      what-to-do-if-found.
ELSE    what-to-do-if-not-found.
THEN
```

If you don't do anything special for *found/notfound*, you can write:

```
SPECIAL-SEARCH test-for-match? REPEAT DROP
```

If `test-for-match` is not really a test but an action, you want to do on every element:

```
SPECIAL-SEARCH do-it. FALSE REPEAT DROP
```

This is so common, another macro is appropriate.

```
MACRO ITERATE " FALSE REPEAT DROP "
SPECIAL-SEARCH do-it. ITERATE
```

Example. Searching a file. [Figure Five]

Now we can write:

```
: LISTING ( -- ) LINES> TYPE CR ITERATE ;
```

Or use nonce words:

```
( List the file. )
:: LINES> TYPE CR ITERATE ;;

( Count lines. )
0 :: LINES> 2DROP 1+ ITERATE ;; U.

( List comments. )
: BEGINS-WITH ( a1 k1 a2 n2 -- a1 k1 flag )
        DUP >R 2OVER R> MIN S=
;

:: LINES>
    S" ( " BEGINS-WITH IF
        TYPE CR
    ELSE 2DROP THEN
ITERATE ;;
```

(Helpful definitions.)

```
: .LINE TYPE CR ;

MACRO ?? " IF \ THEN "
```

(Simple FGREP)

```
:: LINES>
   S" pattern" SEARCH ?DUP AND ?DUP ?? .LINE
   ITERATE ;;
```

(That just prints the rest of the line. For the whole line:)

```
:: LINES> 2DUP S" pattern" SEARCH NIP NIP
?DUP AND ?DUP ?? .LINE ITERATE ;;
```

Let's capture that in another definition.

```
: ?MATCH                    ( a1 u1 a2 u2 -- )
  2>R 2DUP 2R> SEARCH NIP NIP
  ?DUP AND ?DUP ?? .LINE
;

:: LINES> S" pattern" ?MATCH  ITERATE ;;
```

In a long printout, you may want to pause, or terminate prematurely.

```
MACRO FAILURE   " FALSE EXIT "
MACRO SUCCESS   " TRUE  EXIT "

: QUIT?                        ( -- flag )
( If you haven't hit a key, )
( return FALSE immediately. )
   KEY? 0= ?? FAILURE
( If you hit a key other than )
( the space bar, return TRUE. )
   KEY BL = NOT ?? SUCCESS
( Otherwise, pause and return )
( FALSE when you hit any key. )
   KEY DROP
   FAILURE
;
```

It's used in `?ITERATE`, an alternative to `ITERATE`:

```
MACRO ?ITERATE " QUIT? REPEAT DROP "

( 'LISTING' redefined. )
: LISTING  LINES> .line ?ITERATE ;
```

Neil Bawd • Goat Hill, California
NeilBawd@forth.org

Neil Bawd is known by FORML Conference attendees; he should not be mistaken for his evil twin. For the source for this article, request "The View from Goat Hill #2: The Search Paradigm."

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, data-base 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

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 an educational forum for discussion of the 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 bases 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 **NEW**

Fast service by fax: 408-373-2845

BOOKS ABOUT FORTH

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

Annotated glossary of most Forth words in common usage, 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 the name of a Forth model designed to be portable to a large number 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 contains 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.*

THE FIRST COURSE, C.H. Ting 223 – $25

This tutorial's goal is to expose you to the very 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 a 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, called *The Forth Course*, 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 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. Great learning aid. polyFORTH is the dialect used. 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.*

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 the inner workings of F-PC. *269 pp.*

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.), Leo Brodie 245 – $37

In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. *346 pp.*

THINKING FORTH, Leo Brodie 255 – $35

BACK BY POPULAR DEMAND! The bestselling author of *Starting Forth* is back again with the first guide to using Forth to program applications. This book captures the philosophy of the language to show users how to write more readable, better maintainable applications. Both beginning and experienced programmers will gain a better understanding and mastery of such topics: 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 real-life users and with Forth's creator Charles H. Moore. To program intelligently, you must first think intelligently, and that's where *Thinking Forth* comes in. 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 written 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. The material covered discusses SBus development for both OpenBoot 1.0 and 2.0 systems. *414 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).

FIG has reserved 100 complete sets of FD Volume XI

Available on a first-come, first-served basis while supplies last.

For a little less than a year's memberhsip, you can have all this Forth knowledge on your bookshelf, for immediate reference or leisurely study. Your member discount applies, naturally. The total member price of just $39.50 *includes* shipping and handling (non-members pay $42.50; California residents add the amount of sales tax for your area before the shipping and handling—see the mail-order form).

Forth Interest Group
100 Dolores Street, Suite 183 • Carmel, California 93923
voice: 408-373-6784 • fax: 408-373-2845
e-mail: office@forth.org

Fast service by fax: 408-373-2845

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 determined 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 incl. in 1985 FORML Proceedings.
★★ **IBM, 114Kb**

A Simple Inference Engine, Martin Tracy C005 – $8
Based on inf. 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-Std. 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.

MORE ON FORTH ENGINES

Volume 10 (January 1989) RTX reprints from 1988 Rochester Forth conference, object-oriented cmForth, lesser Forth engines. *87 pp.*	810 – $15
Volume 11 (July 1989) RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility. *93 pp.*	811 – $15
Volume 12 (April 1990) ShBoom Chip architecture and instructions, neural computing module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pp.*	812 – $15
Volume 13 (October 1990) PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth. *107 pp.*	813 – $15
Volume 14 RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pp.*	814 – $15
Volume 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.*	815 – $15
Volume 16 OK-CAD System, MuP20, eForth system words, 386 eForth, 80386 protected mode operation, FRP 1600 – 16-Bit real time processor. *104 pp.*	816 – $15
Volume 17 P21 chip and specifications; Pic17C42; eForth for 68HC11, 8051, Transputer *128 pp.*	817 – $15
Volume 18 MuP21 – programming, demos, eForth *114 pp.*	818 – $20
Volume 19 More MuP21 – programming, demos, eForth *135 pp.*	819 – $20
Volume 20 More MuP21 – programming, demos, F95, Forth Specific Language Microprocessor Patent 5,070,451 *126 pp.*	820 – $20
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" (Specify size: Small, Medium, Large, X-Large on order form) white design on a dark blue shirt or green design on tan shirt.	601 – $18
BIBLIOGRAPHY OF FORTH REFERENCES (3rd ed., January 1987) Over 1900 references to Forth articles throughout computer literature. *104 pp.*	340 – $18

Last 5

DR. DOBB'S JOURNAL back issues

Annual Forth issues, including code for various 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	***408.37.FORTH***
weekdays	*408.373.6784*
9.00 – 1.30 PST	***408.373.2845** (fax)*

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

Non-Post Office deliveries: include special instructions.	The amount of your ⇓ **sub-total...**	...the **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

PRICES MAY CHANGE WITHOUT NOTICE

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 $45/53/60	
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. & Canada surface; $53 Canada air mail; 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: 408-373-2845

Figure One

```
BEGIN     test-for-more?                 \ More?
WHILE     test-for-match?                \ Found?
0= WHILE  advance-to-next-element.       \ Next.
REPEAT    what-to-do-when-found.         \ Found.
ELSE      what-to-do-when-not-found.     \ Not found.
THEN
```

Figure Two

```
BEGIN     test-for-more?                 \ More?
WHILE     test-for-match?                \ Found?
UNTIL     what-to-do-when-found.         \ Found.
ELSE      what-to-do-when-not-found.     \ Not found.
THEN
```

Figure Three

```
FALSE BEGIN
        ?DUP IF    FALSE
        ELSE test-for-more? DUP
        THEN
    WHILE   test-for-match?
    REPEAT
    IF      what-to-do-when-found.
    ELSE    what-to-do-when-not-found.
    THEN
```

Figure Four

```
: ?test-for-more?
    ?DUP IF FALSE ELSE
        test-for-more? DUP
    THEN
;

MACRO SPECIAL-SEARCH " FALSE BEGIN ?test-for-more? WHILE "
```

Figure Five

```
: ?lines>?          ( flag -- caddr len true | flag false )
    ?DUP IF FALSE ELSE
        Line-Buffer DUP MAXLINE THE-FILE
            READ-LINE checked ?DUP 0= IF
                2DROP FALSE DUP  THE-FILE REWIND
            THEN
    THEN
;

MACRO LINES> " FALSE BEGIN ?lines>? WHILE "
```

STANDARD FORTH TOOL BELT – #02

Simple Macros

In "Tool Belt #1," a couple of definitions have the form:

```
: name S" blahdy-blahdy-blah " EVALUATE ;
IMMEDIATE
```

Available in any Standard Forth

This is a way to provide simple macros for Forth. Typing or writing *name* is the same as typing or writing *blahdy-blahdy-blah*.

There are many places where simple macros would be useful. So let's abstract the defining process.

Using `STRING` from *Starting Forth*, `MACRO` can be defined:

```
: MACRO  CREATE IMMEDIATE  CHAR STRING
DOES> COUNT EVALUATE  ;
```

On systems where `WORD`'s buffer happens to be at `HERE`, `STRING` can be defined:

```
: STRING ( char -- )  WORD C@ 1+ ALLOT ;
```

In Forth, Inc.'s Power MacForth, `MACRO` can be defined:

```
: MACRO
  CREATE IMMEDIATE  CHAR ?PARSE CS,
  DOES>  COUNT EVALUATE  ;
```

Here is a definition using `PLACE` from F83.

```
: PLACE
  2DUP >R >R
  CHAR+ SWAP  CHARS MOVE
  R> R> C!  ;

: STRING
  WORD COUNT  HERE  OVER 1+
  CHARS ALLOT  PLACE  ;
```

The stack effects are:

```
PLACE    caddr u addr --
STRING   char "ccc<char>" --
MACRO    "name<spaces><char>ccc<char> --
```

The definition can be written using required words only—no Core Extension words or other optional word sets—and will work in any Standard Forth. No transient area, other than that used serially by `WORD`, and no temporary storage are used.

Because the text for macros is acquired by `WORD`, macro definitions must be on one source line. You can have more than one macro definition on a line, though. The need for long macros can be satisfied by using macros within macros.

For portability, stick to one line.

Some examples of macro definitions are given in Figure One. Comment out definitions that don't interest you.

`ANEW` can be defined:

```
: ANEW
  >IN @ BL WORD FIND
  IF EXECUTE ELSE DROP THEN
  >IN ! MARKER ;
```

How about a parameter?

Much can be done with macros without parameters. But it would be nice to have a parameter when you need one. So here is a redefinition of `MACRO` that allows parameters occurring once in the macro. It also gives you a hook to compile long macros—up to 255 characters.

A parameter must be a single word, and it follows the use of the macro. \, which is otherwise useless in a macro, is used in the macro as the placeholder for a parameter. In a macro, the parameter cannot be quoted or be the name of a word being defined, because of parsing restrictions in `EVALUATE`d strings. Placeholders are replaced by parameters, in the order in which they occur.

Think of a macro with parameters as a stencil with slots to be filled.

You can use \ as a parameter to leave a slot empty. [Figure Two]

If you don't already have `/STRING` from the String-Handling word set, it can be defined:

```
: /STRING  ( a n k -- a+k n-k )
  2>R  R@ CHARS +  2R> - ;
```

Some examples of simple macros using parameters are shown in Figure Three.

When should I use a macro?

1. Re-writing the examples without using `EVALUATE` will show you one reason.

   For instance, define `??` without `EVALUATE`. Be sure you can do everything the macro will do.

   Even something as simple as `MACRO AGAIN " 0 UNTIL "` is an advanced topic without `MACRO`.

   Many definitions are much easier using macros, rather than considering how to use `LITERAL` and `POSTPONE` on each word.
2. A second reason is when you want to eliminate the overhead of subroutine nesting. You want to trade *space* for *time*.

   Be selective. `BOUNDS` defined as `MACRO BOUNDS " OVER + SWAP "` in general won't do noticeably better than :

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

Wil Baden is a professional programmer with an interest in Forth. For a copy of the source for this article, send e-mail requesting *Tool Belt #2: Simple Macros.*

STANDARD FORTH TOOL BELT - #02

Figure One

```
MACRO ::      " ANEW NONCE : NONCE-DEF "
MACRO ;;      " ; NONCE-DEF NONCE "
MACRO S=      " COMPARE 0= "
MACRO TH      " CELLS + "
MACRO ANDIF   " DUP IF DROP "     MACRO ORIF " ?DUP 0= IF "
MACRO =IF     " OVER = IF DROP "
MACRO S=IF    " 2OVER S= IF 2DROP "
MACRO SUCCESS " TRUE EXIT "     MACRO FAILURE " FALSE EXIT "
MACRO NOT     " 0= "
MACRO 2>R     " SWAP >R >R "
MACRO 2R>     " R> R> SWAP "
MACRO 2R@     " R> R@ SWAP DUP >R "
MACRO ENDIF   " THEN "
MACRO UNLESS " 0= IF "
MACRO AGAIN   " 0 UNTIL "
MACRO FOR     " BEGIN ?DUP WHILE 1- >R "     MACRO NEXT " R> REPEAT "
MACRO #DO     " 0 ?DO "
MACRO ?REPEAT " [ 0 CS-PICK ] UNTIL "
MACRO LINES>  " FALSE BEGIN ?lines>? WHILE "
MACRO ITEM>   " >R FALSE BEGIN R> ?node? WHILE DUP >R CELL+ "
MACRO ITERATE " FALSE REPEAT DROP "
```

Figure Two

```
: split-at-char               ( a n char -- a+k n-k a k )
   >R  2DUP                      ( a n a+k n-k)( R: char)
      BEGIN  DUP WHILE  OVER C@ R@ =
      0= WHILE  1 /STRING  REPEAT THEN
   R> DROP                           ( R: )
   DUP >R  2SWAP  R> -           ( a+k n-k a k)
;

: DOES>MACRO DOES>
   COUNT BEGIN                      ( caddr u)
      [CHAR] \ split-at-char  2SWAP 2>R   ( caddr u)
      EVALUATE                            ( )
   R@ WHILE
      BL WORD COUNT EVALUATE
      2R> 1 /STRING                       ( caddr u)
   REPEAT                                 ( )
   2R> 2DROP
;

: MACRO   CREATE IMMEDIATE  CHAR STRING  DOES>MACRO  ;
```

Figure Three

```
MACRO ??     " IF \ THEN "
MACRO ?LEAVE " ?? LEAVE "
MACRO H:     " <hex> \ </hex> "
MACRO 'TH    " CELLS \ + "
MACRO TIMES " #DO \ LOOP "
MACRO TIME   " ::  N 0  COUNTER >R  DO  \ \   LOOP  R> TIMER  CR  ;; "
MACRO THE    " [ ALSO \ ] \ [ PREVIOUS ] "
MACRO CLEAR " DEPTH TIMES DROP "
```

`BOUNDS OVER + SWAP ;`. That's because it appears outside a loop, not inside.

Sometimes the system you're working with has a different word that does the same thing as the word you'd like to use. For example, `<ROT` instead of `-ROT`, or `UPCASE` instead of what you like for conversion to uppercase, or something that does the same as a standard word. So define `MACRO -ROT " <ROT "`, etc. The run–time overhead of your definition is eliminated.

3. Use a macro for short definitions when the compiler might give better results for in-line code.

```
1024 CONSTANT 1K
MACRO K " 1K * "
```

I expect `401 K` to be a literal, and most other ways of using `K` to become an in-line shift.

4. In a target compiler, macros that work just for source code can be used to provide the convenience of a definition without having the overhead in the target.

Thus, if the target doesn't have `NIP`, define `MACRO NIP " SWAP DROP "`.

Never define `: -ROT ROT ROT ;`. A high-level definition of `-ROT` destroys any advantage `-ROT` might have—it adds the overhead of nesting to the two `ROT`s. If you want to use `-ROT`, define it as `MACRO -ROT " ROT ROT "`. A friendly compiler should do the right thing for `ROT ROT`, even when it doesn't have `-ROT`.

Appendix

```
MACRO ::     " ANEW NONCE   : NONCE-DEF "
```

Start compiling a nonce word. Used extensively for data initialization and testing. Use `;;` to complete the definition, execute it, and forget it.

Nonce words let you execute loops from the keyboard. They also allow you to initialize data structures programmatically.

Nonce words wouldn't be needed with self–compiling `IF`, `BEGIN`, `DO`, `?DO`.

From *The Random House Dictionary of the English Language*: *nonce word*, a word coined and used only for the particular occasion.

```
MACRO ;;     " ;  NONCE-DEF  NONCE "
```

Finish compiling a nonce word, execute it, and forget it. If a syntax error occurs in compilation, just start over. If other corrections are needed, type `NONCE` to recover, make the corrections, and start over.

```
MACRO ??     " IF \ THEN "
```

For the frequent situation of `IF` being used with a single word.

`?? a-word` becomes `IF a-word THEN`.

Examples: `?? LEAVE`, `?? NEGATE`, `DEPTH 0= ?? 0`, `/MOD SWAP ?? 1+`, `@ ?DUP ?? EXECUTE`.

The single word may be a macro, which will expand.

```
MACRO 2>R    " SWAP >R >R "
MACRO 2R>    " R> R> SWAP "
MACRO 2R@    " R> R@ SWAP DUP >R "
MACRO R+!    " R> + >R "
MACRO R!     " R> DROP >R "
```

Words manipulating the return stack are easier to define with macros. Such definitions would only be used if they were not already defined directly.

```
MACRO ENDIF  " THEN "
MACRO AGAIN  " 0 UNTIL "
MACRO UNLESS " 0= IF "
MACRO #DO    " 0 ?DO "
```

Words involving the control-flow words are easier to write and understand with macros.

```
MACRO FOR    " BEGIN ?DUP WHILE 1- >R "
MACRO NEXT   " R> REPEAT "
```

That implements the `FOR ... NEXT` loop used in some Forth systems.

```
MACRO ANDIF " DUP IF DROP "
MACRO ORIF  " ?DUP 0= IF "
```

Short circuit conditionals.

`test-a ANDIF test-b THEN`: if `test-a` is false, don't bother executing `test-b`.

`test-a ORIF test-b THEN`: if `test-a` is true, don't bother executing `test-b`.

```
MACRO =IF   " OVER = IF DROP "
```

A common sequence. Like a `CASE` statement.

```
MACRO S=IF  " 2OVER S= IF 2DROP "
```

A `CASE` statement for strings.

```
MACRO NOT    " 0= "
MACRO TH     " CELLS + "
MACRO S=     " COMPARE 0= "
```

The above definitions were made with macros to make it easier for the compiler to optimize, and to avoid the overhead of nesting functions.

```
MACRO TIMES " #DO \ LOOP "
```

Execute the next word *n* times. For very short `DO` loops. Note the use of macro `#DO`.

[See Figure Four] A rare, simple macro with two parameters. `COUNTER` and `TIMER` are the words used for timing in the system I'm using. You should do what works with your system. The second parameter is there to clean up or complete the work done by the first parameter.

For example, `TIME RANDOM DROP`.

You will have to define `N` before using `TIME`. I generally define it as a value word, so I can experiment with the number of iterations.

To time an empty loop: `TIME \ \`

```
MACRO THE   " [ ALSO \ ] \ [ PREVIOUS ] "
```

STANDARD FORTH TOOL BELT - #02

Another simple macro with two parameters. For the Search-Order Extension word set.

`THE` is used like `THE EDITOR I` or `THE FORTH I`, where the first parameter is a vocabulary, and the second is a word in that vocabulary.

```
MACRO  H:  " <hex> \ </hex> "
```

Used to change the `BASE` to hex at run time for the next word only. It needs supporting definitions like the ones in Figure Five, which will be explained in later articles.

Figure Four

```
MACRO TIME  " ::  N 0  COUNTER >R  DO  \ \  LOOP  R> TIMER  CR  ;; "
```

Figure Five

```
    VARIABLE Temp
    : <hex>   ( -- ) BASE @ Temp !  HEX ;
    : </hex>  ( -- ) Temp @ BASE ! ;

MACRO LINES>  "  FALSE BEGIN ?lines>? WHILE "
MACRO ITEM>   "  >R FALSE BEGIN R> ?node@? WHILE DUP >R CELL+ "
MACRO ITERATE "  FALSE REPEAT DROP "
```

STRETCHING STANDARD FORTH - #15

Cotinued from page 20

To understand the next set of definitions, here are some examples of its use.

First some useful preliminary definitions. [See Figure Six.]

`?text` is to `.text` as `?` is to `.`.

Given a list, we will want to "list" its items. So we go through the list, displaying every item. [Figure Seven]

Count the number of items in a list. [Figure Eight]

List items in `Some-List` that begin with "ANS" using a nonce word. [Figure Nine]

The definitions of `ITEM>` "item-search" and `ITERATE` are given in Figure Ten.

`ITEM>` and `ITERATE` are macros and expand to the code shown in Figure Eleven.

This gives us a wrapper for doing things with the items of a list. {Figure Twelve]

To search a list, see Figure Thirteen.

The `...` represents what you do with an item, leaving a flag on the stack—true to quit or false to continue.

After `REPEAT` there will be zero on the stack, if you went all the way through the list. If you quit before the end of the list, the address of the node where you quit will be on the stack.

The simplest use is to see whether a string is an item in a list [Figure Fourteen].

Figure Eleven

```
                                 ( list)
   >R FALSE                      ( flag) ( R: node)
   BEGIN R>                      ( flag node)
      ?node@?                    ( node 0 | node' node' | 0 0)
   WHILE                         ( node')
      DUP >R  CELL+              ( item) ( R: node)

   FALSE REPEAT DROP
```

Figure Twelve

```
   ( a-list) ITEM> ( item) "do-something" ITERATE ( )
```

Figure Thirteen

```
   ( a-list) ITEM> ( item) ... ( done?) REPEAT ( 0|node)
```

Figure Fourteen

```
: item? ( caddr u list -- 0|node )
   ITEM> COUNT 2OVER COMPARE 0= REPEAT NIP NIP
;
```

(Next article: "Ordered Lists.")
For *nonce word* and *macro* see Tool Belt #1 and #2.

MPE's Forth Coding Style Standard

[Continued from the preceding issue. Portions of this document, including parts of some of the examples, were edited lightly by FD *for publication in this format. —Ed.]*

End of definition

At the end of the definition, the final word, the semi-colon, or `end-code` will not be indented, and will be at the left-hand end of the line. This ensures that the ends of definitions can be found. It also helps when code is added to the end of a word, by avoiding the possibility of having several semi-colons at or near the end of the word.

The move to literate programming ... will lead to increased use of tools to produce formatted output of the design and source code

```
: WORD1      \ n1 n2 - n3 ; function to ...
  ...
  ...
;

CODE WORD2   \ n1 n2 - n3 ; function to ...
  ...
  ...
END-CODE
```

Immediate Declaration

If a word is to be made `IMMEDIATE`, the word to make it so should appear just after the semi-colon or `END-CODE`:

```
: WORD1      \ n1 n2 - n3 ; function to ...
  ...
  ...
; IMMEDIATE
```

If the word were to be placed on the line following the end of the definition, though legal, there would be a possibility of another word being inserted between the two, and the first word then losing its immediate or public status.

Comments

The function of a comment is to explain why, not what. Comments are written for people, *not* just the original programmer. Sometimes it hurts to come back to code you wrote a few years ago. From observation, code that was commented when it was written is more reliable than code that was commented afterwards.

The comments can even be written first using a PDL (program description language). Algol was first designed as a language for writing algorithms before it was implemented as a computer language. The move to literate programming and including the design within the code will lead to increasing use of tools that can process source files to produce a formatted output of the design and code using different fonts and layout rules.

Microprocessor Engineering, Ltd.
Southampton, England

MPE house rule

Comment as you write.

Definitions should be commented as well as possible. In a text file, there is no excuse for not having enough room to write comments, so comments should be used liberally. In a definition, there should be comments down the right-hand side of the page, in parallel with the code. These comments should start in a uniform column, which should as far as possible be consistent throughout the file. This column should be further to the right than the starting column for the stack comment and short description, usually about half way across the page.

```
: WORD1       \ n1 n2 - n3 ; function to ...
  ...        \ get the pointer
  ...        \ modify the address
;
```

Note that comments in a word body should be vertically aligned. Assuming that the paper is 80 columns wide, comments can often start at column 41, a convenient tab stop.

Line comments are best started with the `\` word—comment to end of line. This is in preference to the `(` word, which must be terminated with a `)`. This last is easily forgotten. These comments should not be on the stack-detail level, though this may be appropriate in certain cases. They should, however, give descriptive information on the state of the system at that point—describing the overall action of the line of code, of the phrase. Needless to say, comments should also be correct.

On a point of style, it is better if the editor inserts tabs between the code and the comment than a series of spaces. This leaves less tidying-up to do after small changes to a line of code. It also makes the source file more compact on disc and faster to load.

Defining Words

Defining words present a special case of definition. This is because, as the word breaks down into two parts, more care should be given to the indentation:

```
: WORDN \ n1 - ; - n3 ; function class to ...
  CREATE       \ defining portion
  ...         \ lay down data
  DOES>>       \ execution portion
  ...         \ get the data ...
;
```

The CREATE and DOES> words should be indented to below the name of the word. The code in the CREATE and DOES> portions should then be indented by further spaces. The layout of DOES> and ; also applies to ;CODE and END-CODE. It is also often found useful to document the stack action of the relevant portion of the word on the line with the CREATE and DOES> words:

```
: WORDN  \ n1 — ; — n3 ; function class to ...
 CREATE     \ n1 — ; defining portion
 ...      \ lay down data
 DOES>>     \ — n3 ; execution portion
 ...      \ get the data ...
;
```

Control Structure Layout

Control structures should be laid out for ease of understanding, and to easily spot overlapping or incomplete structures. To this end, indenting and the use of many lines makes the layout easy. Again, there is no lack of space on a page, and this should be used to advantage.

Flags and Limits

As Forth uses a postfix notation, the flag used to control program flow is specified before the structure or test which uses it. The flag should be identified on the line immediately preceding the test which will use it, as should loop limits:

The contrary view is that most bugs in code appear at structure and procedure boundaries.

```
VAR @         \ get flag
IF            \ if set ...
 ...
ENDIF

VAR @ 0       \ make loop limits
DO            \ for each ...
 ...
 LOOP

VAR1 @ VAR2 @ AND \ we need these because ...
VAR3 @ OR         \ or this because ...
 IF
 ...
ENDIF
```

Indenting

For ease of reading, the start and end of a control structure should be placed on lines by themselves. This makes them easy to spot—for presence or absence. Modern editors with automatic colouring such as Ed4Windows, WinEdit, and CodeWright can do this automatically for Forth. The code within the structure should then be indented by a uniform amount, normally two spaces:

```
...
DO
 ...
LOOP
```

```
             \ phrase comment as question?
IF
              \ yes:
 ...          \ and indent the comment too
ELSE          \ no:
 ...         \ and indent too
ENDIF  \ — x y ; good place to show result

BEGIN        \ x y — ; show stack
 ...       \ question?
     WHILE \  indentation shows structure
 ...
REPEAT

CASE                   \ description needed
 xx OF ... ... ENDOF \ case 1
 yy OF ... ENDOF \ case 2
 zz OF ...           \ big case 3
 ...             \ consider factoring
 ENDOF
 ...             \ default
ENDCASE
```

At the end of a control structure, the structure termination word will be without indentation and back below the start of the structure, ensuring that starts and ends of structures are vertically aligned, so that it is easy to see an unbalanced structure or piece of code. See above for examples.

The multiple exit control structures introduced by ANS Forth are deprecated at MPE because their intention is to allow the user to associate several exit conditions with exit actions. However, the ANS format separates these visually, leaving a horrible job for program maintenance. A simple extension to CASE ... ENDCASE provides the visual match.

```
CASE                     \ description needed
  xx OF ... ... ENDOF \ case 1
  yy OF ... ENDOF \ case 2
        ...              \ default
NEXTCASE \ branches back to just after CASE
```

Short Control Structures

If the code within a control structure is very short, then it is good practice to leave the start and end of the structure on one line, with the body of the structure. However, what constitutes a short structure is very subjective.

```
...
DO  I  .LOOP
```

Note that there is more than one space between the DO and the I . and again to the LOOP. This helps the code to retain phrasing.

The contrary view is that most bugs in code appear at structure and procedure boundaries. These boundaries need to be made more visible, even if costs a few more lines on the screen.

Layout of Code Definitions

The layout of code definitions will be slightly different from the layout of high-level definitions. For a start, the layout will be more vertical than the corresponding high-level code. If a word is being defined, the top line of the definition will reflect that of any other word—i.e., it will have a stack comment and a brief description. If a label is being defined, then there may not be a stack effect, but there will still be a brief description of the function of the procedure or sub-routine. The code that then follows may be very vertical, or may be phrased more:

```
CODE WORD1          \ n1 – n2 ; word to ...
 MOV AX, BX
 ADD BX, # 03
 XCHG BP, SP BP INC XCHG BP, SP
 ...
END-CODE
```

Of course, there will still be plenty of in-line comments. Where code is being converted from a conventional assembler, it may be useful to retain the common tabbed layout of many assembler programmers. This also makes it easier for assembler programmers to read the Forth assembler. In most cases, ease of reading or writing code leads to reliability.

```
CODE WORD1     \ n1 – n2 ; word to …
   MOV  AX, BX \ BX=TOS, so get n1 from TOS
   ADD  BX, # 03 \ this is the increment to…
   XCHG BP, SP \ swap stacks in order to
   PUSH AX     \ hold this out of the way
   XCHG BP, SP \ for later
 ...
END-CODE
```

Data Structure Layout

Data structures should be laid out in a consistent fashion, not unlike the layout of definitions. However, there will be certain differences.

Where several similar items are being defined in one go, only one stack comment is required for the block. This should be at the start of the line above the items:

```
\ – addr ; the variables for input data
VARIABLE BILL           \ contains ...
VARIABLE BEN            \ defines the ...
```

Constants

Constants require a value as part of their definition. This value should appear at the left-hand end of the line. If several constants are being defined at one time, the words CONSTANT should line up vertically.

```
\ – n ; constants for destinations
 3 CONSTANT BILL        \ constant for ...
23 CONSTANT BEN         \ constant for ...
```

It is good practice to define all constants in one place in the file, or in one file in the set. See the earlier section on the layout of a file.

Variables

Variables should be defined at the left-hand end of the line, and should be followed by an appropriate comment. A general stack comment may precede the whole block of variables:

```
\ – addr ; variables
VARIABLE BILL                \ contains …
VARIABLE BEN                 \ defines the
...
```

If the variables are to be pre-initialised to anything other than zero, the initialisation value should follow the definition of the variable:

```
DECIMAL
VARIABLE BILL 25 BILL !  \ contains …
VARIABLE BEN                \ contains …
 34 BEN !     \ default to 34 because …
```

The use of program initialisation procedures has much to recommend it. It is surprising how often an application works after first being compiled, but not when run a second time. This situation is usually caused by the program not defining its required state within the initialisation code.

Buffers

A buffer may be defined and require more than one or two bytes of dictionary space. This space may be pre-initialised, or it may be a scratch area, or otherwise filled by the application. The buffer should be defined and any pre-initialisation should immediately follow its definition:

```
CREATE BILL               \ the ...
 10 ALLOT                 \ for a PC Forth
 "" this" BILL $MOVE      \ preset to 'this'

VARIABLE BILL
 10 ALLOT-RAM      \ for a ROM/RAM target
```

Tables

A table may be predefined—such as a look-up table. This will usually be created in the dictionary, and will include its data. The important point is consistency and ease of reading:

```
CREATE TABLE  \ – addr ; bit-pattern table
 1 C, 2 C, 4 C, 8 C,          \ b0..b3
 16 C, 32 C, 64 C, 128 C,         \ b4..b7
```

Note that the numbers and the commas line up. This makes reading easy.

To be continued in the next issue...

FORTHWARE-#9

Adaptive Digital Filters

Introduction

This month, we turn to the topic of *adaptive digital filters.* Just as we improved our ability to apply controls to external processes by introducing a *closed loop,* we will improve our filtering by closing the loop and using the filter output to modify the filter in real-time.

Adaptive filters can either be IIR (Infinite Impulse Response) or FIR (Finite Impulse Response). There are also *nonlinear* filter types, but we will not consider them here. Generally, the form of the filter remains fixed as it runs, but a special output channel of the filter (usually called the error output) is fed into a process which recalculates the filter *coefficients* in order to produce an output that is closer to the desired form (see Figure One). If it is properly designed, the result of such a filter is an output that enhances the desired component over a wide range of conditions.

As with ordinary digital filters, a vast amount of written material is available on the topic of adaptive filters. We will only give a limited introduction to the topic here. You should also be aware that a proper treatment of adaptive filters requires a good deal of calculus and linear algebra; I will mostly spare you all the math, in order to provide an introduction to the types of adaptive filters and their applications.

What Can Go Wrong

When you think about it, adaptive filters are a little scary. They process a signal and then decide to adjust themselves in order to alter the signal's characteristics. How can you be sure the filter makes the right decision and doesn't make things worse? Mathematically, what we are concerned with here is the *stability* of the filter. The question of the filter's stability arises because of two related properties of adaptive filters. First, the feedback of an error term makes the filter behave as a differential equation. It is perfectly valid for a differential equation to have an unstable solution. If we have happened to design an adaptive filter whose underlying differential equation has an unstable solution, we are in trouble. Second, since we are implementing our filter in the discrete time domain, the filter implementation becomes a *finite difference approximation* to the differential equation. Finite difference equations have their own stability constraints, such that it is possible to have numerically unstable solutions even when the analytic solution is stable.

Analyzing the stability of an adaptive filter tends to be a deep exercise in mathematics (involving such things as Laplace transforms and numerical analysis). Having a mathematical software package such as Mathematica or Macsyma helps a great deal, but often, in the real world, such filters are just empirically tested. Hopefully, a proper analysis is used for filters that will be installed into safety-critical systems.

General Principles

Unlike nonadaptive filters, which just pass their input data through the filtering mechanism, adaptive filters also apply a system model. The estimate of the system is compared against the filter output to create an error signal; this error signal is then used in a *cost function* of some kind, which indicates how well (or badly) we are doing with the current parameters of the filter.

The Kalman Filter

The simplest error signal we can generate is just the difference between what we are getting out of the filter, $\hat{x}_t$, and what we expect to see, x_t,

$$e_t = x_t - \hat{x}_t \qquad (1)$$

For many problems, an overshoot is just as bad as an undershoot, so we can use the *mean square error* as a cost function. There are lots of other cost functions we could use, but this one is particularly convenient mathematically. The earliest adaptive filter derived from this error and cost function is the *Wiener* filter. Unfortunately, it is in FIR form and has coefficients that extend infinitely back in time. Except for certain periodic systems, this filter is not very practical. The filter can be re-derived in IIR form—this is known as the *Kalman* filter. The discrete time, linear, form of this equation looks like,

$$\hat{x}_t = b_t \hat{x}_{t-1} + K_t z_t \qquad (2)$$

b_t is a model of how the system goes from one time interval to the next. It is our best understanding of how the ideal system goes from a value at time $t-1$ to a value at time t.

K_t is the *Kalman gain.* It is not controlled by the measurements directly, but instead is determined by how good you think the model b is, compared to the quality of the observations.

z_t is called the *innovation.* It is an estimate of what you think the error *will be* at time t, given a measurement at time t, y_t, and a prediction of x at time t based upon the best estimate of x at $t-1$ extrapolated to time t by applying our model function, b. The simplest example of how to calculate the innovation is,

$$z_t = y_t - H b_{t-1} \hat{x}_{t-1} \qquad (3)$$

where H is a function that may be necessary to convert the components of x to the components of y. (An example is the meteorological case of estimating the humidity—the x—based upon wet-bulb and dry-bulb temperature measurements—the y. The H function would have one component that converts humidity to wet-bulb temperature, and one for the conversion to dry-bulb). In an application, the innovation is a known

Skip Carter • Monterey, California
skip@taygeta.com

Skip Carter, a scientific and software consultant, and leader of the Forth Scientific Library project, maintains www.taygeta.com. He is also the President of the Forth Interest Group.

function, like above, and the function b is known. What is *not* known is the gain, K; this must be calculated in parallel with the model estimation. The time varying gain is where the adaptive nature of the Kalman filter expresses itself.

In order to determine the equation that gives us the gain function, we have to spend some time with optimal estimation theory. I will not spend the time on this here, but just show the result. In the scalar case, the gain function is:

$$K_t = \frac{H\left[b^2 p_{t-1} + \sigma_g^2\right]}{\sigma_v^2 + H^2\sigma_g^2 + H^2 b^2 p_{t-1}} \qquad (4)$$

Two of the new quantities, σ_g^2 and σ_v^2, are the noise or error variances for the model, b, and the measurements, respectively. The first is a statement about how good you believe the model of the system is. The second quantifies how good you think your measurements are. Both of these quantities are presumed to be known. The third quantity, p_{t-1}, is the error covariance of *the filter*; it gives effectively the error bars of the current model output. This can be calculated given the gain,

$$p_t = \frac{1}{H}\sigma_v^2 K_t \qquad (5)$$

To use the filter, each time a new observation (y_t) becomes available, we calculate (3) and (4), and then use that information in (2) and (5).

The Kalman filter is frequently applied to systems where x and y are *multi-channel* or *vector* systems. In this case, the equations (2) through (5) are rewritten as matrix equations.

The LMS Adaptive Filter

An important general form for adaptive filters is the Least Mean Squares (LMS) filter. This type of adaptive filter is easy to implement and is widely used because of this. The filter uses a gradient search technique in order to determine how to improve the filter coefficients. This gradient search is also the reason for the basic weakness of the filter: it has a relatively slow convergence rate.

Suppose we consider the Nth-order FIR filter,

$$y_n = h(1)\, x_{n-1} + h(2)\, x_{n-2} + \ldots + h(n) x_{n-N} \qquad (6)$$

the filter error is,

$$e_n = x_n - y_n \qquad (7)$$

With the LMS filter, we adjust the values of the coefficients, h, proportional to the error, e,

$$h_i(n) = h_{i-1}(n) + 2\mu x_{n-i} e_n \qquad (8)$$

where μ is a "learning factor" which controls how strongly the error is weighted. This equation is the consequence of the requirement to minimize the mean square value of e, hence the name of the filter.

The RLS Filter

The Recursive Least Squares (RLS) filter is, in theory, a better filter than the LMS filter, but it is not used as often as it could be because it requires more computational resources. (The LMS filter requires $2N+1$ operations per filter update, whereas the RLS filter requires $2.5N^2+4N$). It has been successfully used in system identification problems and in time series analysis, where its real-time performance is not an issue.

There are several forms of the RLS filter, but all of them are similar to the Kalman filter, in that a covariance is calculated along with the regular filter output. This covariance is then used to calculate a new filter gain.

Listing One shows a Forth implementation of the square-root form of the filter. This form of the filter calculates the square root of the covariance, instead of the covariance directly, in order to improve the numerical stability of the filter. The filter has two tunable parameters: *sigma* is the square root of the variance of the input data, and *lambda* is a weighting coefficient that controls how strongly to perturb the coefficients for a given filter error. These two coefficients need to be carefully considered in order to ensure that the resulting filter is stable. Note that, in this example, the filter error is just the straight difference between the input sample and the convolution of the filter weights and the previous inputs; this could be made to be different for different types of problems.

Application Example—Echo Cancellation

As example of how to apply an adaptive filter to a real-world problem, let's consider the problem of echo cancellation. In this problem, there is a signal we want to recover, x, that we measure—but it is contaminated with multiple copies of itself. These extra copies overlap with unknown delay times and amplitudes. Mathematically, the measurements consist of the series, y, which is the convolution of the clean signal x and the unknown reflection coefficient series h,

$$y = h * x \qquad (9)$$

The goal of the filter is to estimate the unknown reflection coefficients h.

This type of problem is sometimes solved by direct mathematical inversion of the known quantities x and y. In the engineering literature, this is sometimes called the *system identification problem*; in geophysics, it's known as *deconvolution*. An adaptive filter can be used to solve this problem in real-time, but it takes some amount of time for it to converge (particularly if the signal power is low). The adaptive filter approach will also work when h is slowly varying with time; this is something standard deconvolution does not work well with.

We can use any adaptive filter for this; if we use an LMS filter, the coefficients will get adapted by the application of equation (8). If we do this for the echo cancellation problem, we have to replace e_n in that equation with the difference between the convolution of the reference signal, x with the coefficients h and the actual filter input data s:

$$e_n \leftarrow s - y \qquad (10)$$

Conclusion

Take a look at where we have been in this series: we have considered how to generate signals that can be used to manipulate the outside world, how to measure how the outside world is responding to our signals, and how to create a closed-loop controller to make the external system behave in the particular manner we desire. In recent columns, we have looked at how to handle signals that are time varying and how to modify those signals with filters. This time, we enhanced our filtering ability to be able to adaptively modify a signal.

Next time, I am going to shift gears a bit and spend some time considering the software development process itself.

Please don't hesitate to contact me through *Forth Dimensions* or via e-mail if you have any comments or suggestions about this or any other Forthware column.

References

Cowan, C.F.N. and P.M. Grant, (eds.) 1985. *Adaptive Filters.* Prentice-Hall, Englewood Cliffs, NJ, 308 pages. ISBN 0-13-004037-1.

du Plessis, R.M., 1967. "Poor man's explanation of Kalman Filters, or How I stopped worrying and learned to love matrix inversion." Reprinted 1997 by Taygeta Scientific Inc., Monterey, CA.

Listing One.

```
\ rls.fth    An implementation of the square root form of an RLS adaptive filter
\ This is an ANS Forth program requiring:
\         1. The Floating point word set
\         2. The Forth Scientific Library Array words
\
\   There is an environmental dependency in that it is assumed
\   that the float stack is separate from the parameter stack

\ This code is released to the public domain  September 1996   Taygeta Scientific Inc.
\ $Author:   skip  $
\ $Workfile:   rls.fth  $
\ $Revision:   1.1  $
\ $Date:   16 Jul 1997 23:19:58  $
\ ================================================================
\ the Array words from the Forth Scientific Library
S" /usr/local/lib/forth/fsl-util.fth" INCLUDED
\ ================================================================

8 CONSTANT N       \ the filter order

FVARIABLE sigma    \ square root of the initial data variance
FVARIABLE lambda   \ weighting coefficient

FVARIABLE Err       \ filter error output
FVARIABLE Gain      \ filter gain

N Float Array K{         \ filter gain components
N Float Array Phi{       \ filter data
N Float Array H{         \ filter coefficients

\ internal filter coefficient data
FVARIABLE y_old
N Float Array F{
N Float Array V{
N Float Array Alpha{

\ U{ and D{ are actually diagonal matricies such that the
\ filter covariance is U * D * Transpose( U )
N Float Array U{
N Float Array D{

: F+! ( addr -- , F: x -- )  \ increment a Float variable
    DUP F@ F+ F!
;

: initialize ( -- , F: sigma lambda -- )  \ one time initialization
```

```
    lambda F!
    N 0 DO
        0.0E0 Phi{ I } F!
        0.0E0 H{ I } F!
      1.0E0 U{ I } F!
        FDUP D{ I } F!
    LOOP
    sigma F!
;

: }shuffle ( x{ -- )   \ slide all the data values down by one
    1 N 1- DO
            DUP I 1- } F@ DUP I } F!
    -1 +LOOP
    DROP
;

: do_filter ( -- , F: yn --  )  \ apply the RLS filter on the current data
    0.0E0
    N 0 DO
      Phi{ I } F@ H{ N 1- I - } F@ F* F+
    LOOP
    FNEGATE FOVER F+       \ err
    Err F!
    y_old F!
;

: preset ( -- )    \ initial filter setup for each step
    N 0 DO
          U{ I } F@ Phi{ I } F@ F*
          F{ I } F!
          D{ I } F@ F{ I } F@ F* V{ I } F!
    LOOP
    V{ 0 } F@ K{ 0 } F!
    V{ 0 } F@ F{ 0 } F@ F*  lambda F@ F+      ( -- , F: alpha )

    D{ 0 } F@ FOVER F/ D{ 0 } F!         \ d{ 0} = d{ 0}/alpha
    Alpha{ 0 } F!
;

: adjust_gain ( -- )  \ apply RLS adaptive scheme to adjust the gain
    preset
    N 1 DO
      \ d{ i} = d{ i}*alpha/lambda
      D{ I } F@ Alpha{ I 1- } F@ F* lambda F@ F/ D{ I } F!

      \ calculate new alpha
      V{ I } F@ F{ I } F@ F* Alpha{ I 1- } F@ F+
        FDUP Alpha{ I } F!

      \ finish update of D,  d{ i} = d{ i} / new alpha
      D{ I } F@ FSWAP F/ D{ I } F!

      \ update U keeping a copy of the old value
      F{ I } F@ Alpha{ I 1- } F@ F/ FNEGATE
      K{ I 1- } F@ F* U{ I } F@
        FSWAP FOVER F+ U{ I } F!        ( -- , F: uold )

      \ update the gain
      V{ I } F@ F* K{ I 1- } F@ F+
        K{ I } F!
    LOOP

    \ compete the Gain update
```

```
    N 0 DO
           K{ I } F@ Alpha{ I } F@ F/  K{ I } F!
       LOOP
;

: adjust_coefficients ( -- )
    Err F@
    N 0 DO
        FDUP K{ I } F@ F*
      H{ I } F+!
    LOOP
    FDROP
    Phi{ }shuffle
    y_old F@ Phi{ 0 } F!
;

: rls_filter ( -- , F: y -- yf )  \ do RLS filter for one data point
    do_filter
    adjust_gain
    adjust_coefficients
    Err F@
;

\ ==================================================================
0.65E0 0.5E0 initialize
\ ==================================================================
```

Listing Two

```
\ lms.fth    An implementation an LMS adaptive filter
\ This is an ANS Forth program requiring:
\         1. The Floating point word set
\         2. The Forth Scientific Library Array words
\
\   There is an environmental dependency in that it is assumed
\   that the float stack is separate from the parameter stack

\ This code is released to the public domain  September 1996   Taygeta Scientific Inc.
\ $Author:   skip  $
\ $Workfile:   lms.fth  $
\ $Revision:   1.0  $
\ $Date:   17 Jul 1997 02:57:08  $
\
\ ==================================================================
\ the Array words from the Forth Scientific Library
S" /usr/local/lib/forth/fsl-util.fth" INCLUDED
\ ==================================================================

8 CONSTANT N       \ the filter order
8 CONSTANT M       \ the running mean block size

FVARIABLE beta   \ weighting coefficient

M N MAX CONSTANT Ny

M Float Array Err{      \ filter error output
Ny Float Array Y{       \ filtered data
N Float Array A{        \ filter coefficients
N Float Array X{        \ input data

: F+! ( addr -- , F: x -- )  \ increment a Float variable
    DUP F@ F+ F!
;
```

```
: initialize ( -- , F: beta -- )  \ one time initialization
    2.0E0 F* beta F!
    M 0 DO
        0.0E0 Err{  I }  F!
    LOOP
    N 0 DO
      0.0E0 A{  I }  F!
        0.0E0 X{  I }  F!
    LOOP
    Ny 0 DO
      0.0E0 Y{  I }  F!
    LOOP
;

: }shuffle ( n x{ -- )   \ slide all the data values down by one
    1 ROT 1- DO
             DUP I 1- }  F@ DUP I }  F!
    -1 +LOOP
    DROP
;

: do_filter ( -- , F: x --  )  \ apply the LMS filter on the current data
    N X{ }shuffle
    FDUP X{  0 }  F!
    0.0E0
    N 0 DO
      A{  N 1- I - }  F@ X{  I }  F@ F* F+
    LOOP

    Ny Y{  }shuffle
    FDUP Y{  0 }  F!
    FNEGATE F+       \ err
    N Err{  }shuffle
    Err{  0 }  F!
;

: get_adjustment ( k -- , F: -- x )    \ calculate adjustement
    NEGATE
    0.0E0
    M 0 DO
        Err{  OVER I + }  F@ X{  OVER I + }  F@ F* F+
    LOOP
    DROP
    Beta F@ F* M S>D D>F F/
;

: lms_adapt ( -- )  \ apply LMS adaptive scheme
    N 0 DO
      I get_adjustment
        A{  I }  F+!
    LOOP
;

: lms_filter ( -- , F: y -- yf )  \ do LMS filter for one data point
    do_filter
    lms_adapt
    Y{  0 }  F@ Err{  0 }  F@ F-
;

\ =================================================================
 0.85E0 initialize
\ =================================================================
```

Yet another Forth objects package

After criticizing the Neon model in the last issue, here I present (and expose to criticism) a model that I find better, and its implementation. Its properties (most are advantages, in my opinion) are:

- It is straightforward to pass objects on the stack. Passing selectors on the stack is a little less convenient, but possible.
- Objects are just data structures in memory, and are referenced by their address. You can create words for objects with normal defining words like `constant`. Likewise, there is no difference between instance variables that contain objects and those that contain other data.
- Late binding is efficient and easy to use.
- It avoids parsing, and thus avoids problems with state-smartness and reduced extensibility; for convenience, there are a few parsing words, but they have non-parsing counterparts. There are also a few defining words that parse. This is hard to avoid, because all standard defining words parse (except `:noname`); however, such words are not as bad as many other parsing words, because they are not state-smart.
- It does not try to incorporate everything. It does a few things and does them well (in my opinion). In particular, I did not intend to support information hiding with this model (although it has features that may help); you can use a separate package for achieving this.
- It is layered; you don't have to learn and use all features to use this model. In particular, the features discussed after the section "Programming Style" are optional and independent of each other.
- An implementation in ANS Forth is available.

I have used the technique on which this model is based to implement Gray [ertl89] [ertl97]; we have also used this technique in Gforth.

This paper assumes (in some places) that you have read the paper on structures. *[Editor's note: due to space constraints, the Structures paper will appear in the subsequent issue.]*

Why Object-Oriented Programming?

Often we have to deal with several data structures *(objects)*, that have to be treated similarly in some respects, but differ in others. Graphical objects are the textbook example: circles, triangles, dinosaurs, icons, and others, and we may want to add more during program development. We want to apply some operations to any graphical object, e.g., `draw` for displaying it on the screen. However, `draw` has to do something different for every kind of object.

We could implement `draw` as a big `CASE` control structure that executes the appropriate code depending on the kind of object to be drawn. This would be not be very elegant, and, moreover, we would have to change `draw` every time we add a new kind of graphical object (say, a spaceship).

What we would rather do is: When defining spaceships, we would tell the system: "Here's how you `draw` a spaceship; you figure out the rest."

This is the problem all systems solve that (rightfully) call themselves object-oriented, and the object-oriented package I present here also solves this problem (and not much else).

Terminology

This section is mainly for reference, so you don't have to understand all of it right away. I (mostly) use the same Smalltalk-inspired terminology as [mckewan97]. In short:

class
A data structure definition with some extras.

object
An instance of the data structure described by the class definition.

instance variables
Fields of the data structure.

selector (or *method selector*)
A word (e.g., `draw`) for performing an operation on a variety of data structures (classes). A selector describes *what* operation to perform. In C++ terminology: a (pure) virtual function.

method
The concrete definition that performs the operation described by the selector for a specific class. A method specifies *how* the operation is performed for a specific class.

selector invocation
A call of a selector. One argument of the call (the top-of-stack) is used for determining which method is used. In Smalltalk terminology: a message (consisting of the selector and the other arguments) is sent to the object.

receiving object
The object used for determining the method executed by a selector invocation. In our model, it is the object that is on the TOS when the selector is invoked. (*Receiving* comes from Smalltalk's *message* terminology.)

child class
A class that has *(inherits)* all properties (instance variables, selectors, methods) from a *parent class*. In Smalltalk termi-

Anton Ertl • Vienna, Austria
anton@mips.complang.tuwien.ac.at

Anton Ertl uses Forth as a subject of research focusing on native code compilation and as a tool for non-Forth research. He is a coauthor of Gforth, which aims to be both fast and portable.

nology: The subclass inherits from the superclass. In C++ terminology: The derived class inherits from the base class.

(If you wonder about the message-sending terminology, it comes from a time when each object had its own task, and objects communicated via message passing; eventually, the Smalltalk developers realized they can do most things through simple—indirect—calls. They kept the terminology.)

Basic Usage

You can define a class for graphical objects like this:

```
object class
\ "object" is the parent class
  selector draw ( x y graphical -- )
end-class graphical
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is a word (say, a constant) that produces a graphical object.

How do we create a graphical object? With the present definitions, we cannot create a useful graphical object. The class `graphical` describes graphical objects in general, but not any concrete graphical object type (C++ users would call it an *abstract class*); e.g., there is no method for the selector `draw` in the class `graphical`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class
\ "graphical" is the parent class
  cell% field circle-radius

:noname ( x y circle -- )
  circle-radius @ draw-circle ;
overrides draw

:noname ( n-radius circle -- )
  circle-radius ! ;
overrides construct

end-class circle
```

Here we define a class `circle` as a child of `graphical`, with a field `circle-radius` (which behaves just like a field in the structure package); it defines new methods for the selectors `draw` and `construct` (`construct` is defined in `object`, the parent class of `graphical`).

Now we can create a circle on the heap (i.e., `allocated` memory) with

```
50 circle heap-new constant my-circle
```

`heap-new` invokes `construct`, thus initializing the field `circle-radius` with 50. We can draw this new circle at (100,100) with

```
100 100 my-circle draw
```

Note: You can invoke a selector only if the object on the TOS (the receiving object) belongs to the class where the selector was defined or to one of its descendents; e.g., you can invoke `draw` only for objects belonging to `graphical` or its descendents (e.g., `circle`). Immediately before `end-class`, the search order has to be the same as immediately after `class`.

The `object` Class

When you define a class, you have to specify a parent class. So how do you start defining classes? There is one class available from the start: `object`. You can use it as the ancestor for all classes. It is the only class that has no parent. It has two selectors: `construct` and `print`.

Creating Objects

You can create and initialize an object of a class on the heap with `heap-new` (... class -- object) and in the dictionary (allocation with `allot`) with `dict-new` (... class -- object). Both words invoke `construct`, which consumes the stack items indicated by "..." above.

If you want to allocate memory for an object yourself, you can get its alignment and size with `class-inst-size 2@` (class -- align size). Once you have memory for an object, you can initialize it with `init-object` (... class object --); `construct` does only a part of the necessary work.

Programming Style

This section is not exhaustive.

In general, it is a good idea to ensure that all methods for the same selector have the same stack effect: when you invoke a selector, you often have no idea which method will be invoked, so, unless all methods have the same stack effect, you will not know the stack effect of the selector invocation.

One exception to this rule is methods for the selector `construct`. We know which method is invoked, because we specify the class to be constructed at the same place. Actually, I defined `construct` as a selector only to give the users a convenient way to specify initialization. The way it is used, a mechanism different from selector invocation would be more natural (but probably would take more code and more space to explain).

Class Binding

Normal selector invocations determine the method at run time, depending on the class of the receiving object (late binding).

Sometimes we want to invoke a different method. E.g., assume you want to use the simple method for `printing objects`, instead of the possibly long-winded `print` method of the receiver class. You can achieve this by replacing the invocation of `print` with

```
[ bind] object print
```

in compiled code, or

```
bind object print
```

in interpreted code. Alternatively, you can define the method with a name (e.g., `print-object`), and then invoke it through the name. Class binding is just a (often more convenient) way to achieve the same effect; it avoids name clutter and allows you to invoke methods directly without naming them first.

A frequent use of class binding is this: When we define a method for a selector, we often want the method to do what the selector does in the parent class, and a little more. There is a special word for this purpose: `[ parent]`. `[ parent]` *`selector`* is equivalent to `[ bind]` *`parent selector`*, where *`parent`* is the parent class of the current class. E.g., a method definition might look like:

```
:noname
  dup [ parent] foo
  \ do parent's foo on the receiving object
  ... \ do some more
; overrides foo
```

[mckewan97] presents class binding as an optimization technique. I recommend not using it for this purpose unless you are in an emergency. Late binding is pretty fast with this model anyway, so the benefit of using class binding is small; the cost of using class binding where it is not appropriate is reduced maintainability.

While we are at programming style questions: You should bind selectors only to ancestor classes of the receiving object. E.g., say, you know the receiving object is of class `foo` or its descendents; then you should bind only to `foo` and its ancestors.

Method Conveniences

In a method, you usually access the receiving object pretty often. If you define the method as a plain colon definition (e.g., with `:noname`), you may have to do a lot of stack gymnastics. To avoid this, you can define the method with `m: ... ;m`. E.g., you could define the method for `draw`ing a `circle` with:

```
m: ( x y circle -- )
  ( x y ) this circle-radius @ draw-circle
;m
```

When this method is executed, the receiver object is removed from the stack; you can access it with `this` (admittedly, in this example the use of `m: ... ;m` offers no advantage). Note that I specify the stack effect for the whole method (i.e., including the receiver object), not just for the code between `m:` and `;m`. You cannot use `exit` in `m: ... ;m`—instead, use `exitm`.[1]

You will frequently use sequences of the form `this` *`field`* (in the example above: `this circle-radius`). If you use the field only in this way, you can define it with `inst-var` and eliminate the `this` before the field name. E.g., the `circle` class above could also be defined with:

```
graphical class
  cell% inst-var radius

m: ( x y circle -- )
  radius @ draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  radius ! ;m
overrides construct

end-class circle
```

`radius` can only be used in `circle` and its descendent classes, and inside `m: ... ;m`.

You can also define fields with `inst-value`, which is to `inst-var` as `value` is to `variable`. You can change the value of such a field with `[ to-inst]`. E.g., we could also define the class `circle` like this:

```
graphical class
  inst-value radius

m: ( x y circle -- )
  radius draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  [ to-inst] radius ;m
overrides construct

end-class circle
```

Names and Scoping

Inheritance is frequent, unlike structure extension. This exacerbates the problem with the field name convention: One always has to remember in which class the field was originally defined; changing a part of the class structure would require changes for renaming in otherwise unaffected code.

To solve this problem, I added a scoping mechanism (which was not in my original charter): A field defined with `inst-var` is visible only in the class where it is defined and in the descendent classes of this class. Using such fields only makes sense in `m:`-defined methods in these classes, anyway.

This scoping mechanism allows us to use the unadorned field name, because name clashes with unrelated words become much less likely.

Once we have this mechanism, we can also use it for controlling the visibility of other words: All words defined after `protected` are visible only in the current class and its descendents. `public` restores the compilation (i.e., `current`) wordlist that was in effect before. If you have several `protected`s without an intervening `public` or `set-current`, `public` will restore the compilation wordlist in effect before the first of these `protected`s.

Interfaces

In this model, you can only call selectors defined in the class of the receiving objects or in one of its ancestors. If you call a selector with a receiving object that is not in one of these classes, the result is undefined; if you are lucky, the program crashes immediately.

Now consider the case when you want to have a selector (or several) available in two classes: You would have to add the selector to a common ancestor class, in the worst case to `object`. You may not want to do this, e.g., because someone else is responsible for this ancestor class.

The solution for this problem is interfaces. An interface is a collection of selectors. If a class implements an interface, the selectors become available to the class and its descendents. A class can implement an unlimited number of interfaces. For the problem discussed above, we would define an

---

[1] Moreover, for any word that calls `catch` and was defined before loading `objects.fs`, you have to redefine it like I redefined `catch`:

```
: catch this >r catch r> to-this ;
```

Figure One

```
( object selector-body )
2dup selector-interface @ ( object selector-body object interface-offset )
swap object-map @ + @ ( object selector-body map )
swap selector-offset @ + @ execute
```

interface for the selector(s), and both classes would implement the interface.

As an example, consider an interface `storage` for writing objects to disk and getting them back, and a class `foo` that implements it. The code for this would look like:

```
interface
  selector write ( file object -- )
  selector read1 ( file object -- )
end-interface storage

bar class
  storage implementation

... overrides write
... overrides read
...
end-class foo
```

(I would add a word `read` (file -- object) that uses `read1` internally, but that's beyond the point illustrated here.)

Note that you cannot use `protected` in an interface; and, of course, you cannot define fields.

In the Neon model, all selectors are available for all classes; therefore, it does not need interfaces. The price you pay in this model is slower late binding and, therefore, added complexity to avoid late binding.

Implementation

An object is a piece of memory, like one of the data structures described with `struct ... end-struct`. It has a field `object-map` that points to the method map for the object's class.

The *method map*[2] is an array that contains the execution tokens (XTs) of the methods for the object's class. Each selector contains an offset into the method maps.

`selector` is a defining word that uses `create` and `does>`. The body of the selector contains the offset; the `does>` action for a class selector is, basically:

```
( object addr )
@ over object-map @ + @ execute
```

Since `object-map` is the first field of the object, it does not generate any code. As you can see, calling a selector has a small, constant cost.

A class is basically a `struct` combined with a method map. During the class definition, the alignment and size of the class are passed on the stack, just as with `struct`s, so `field` can also be used for defining class fields. However, passing more items on the stack would be inconvenient, so `class` builds a data structure in memory, which is accessed through the variable `current-interface`. After its definition is complete, the class is represented on the stack by a pointer (e.g., as parameter for a child class definition).

At the start, a new class has the alignment and size of its parent, and a copy of the parent's method map. Defining new fields extends the size and alignment; likewise, defining new selectors extends the method map. `overrides` just stores a new XT in the method map at the offset given by the selector.

Class binding just gets the XT at the offset given by the selector from the class' method map and `compile,`s it (in the case of `[ bind]`).

I implemented `this` as a `value`. In an `m: ... ;m` method, the old `this` is stored to the return stack at the start and is restored at the end; and the object on the TOS is stored `TO this`. This technique has one disadvantage: If the user does not leave the method via `;m`, but via `throw` or `exit`, `this` is not restored (and `exit` may crash). To deal with the `throw` problem, I have redefined `catch` to save and restore `this`; the same should be done with any word that can catch an exception. As for `exit`, I simply forbid it (as a replacement, there is `exitm`).

`inst-var` is just the same as `field`, with a different `does>` action:

```
@ this +
```

Similar for `inst-value`.

Each class also has a wordlist that contains the words defined with `inst-var` and `inst-value`, and its protected words. It also has a pointer to its parent. `class` pushes the wordlists of the class and all its ancestors on the search order, and `end-class` drops them.

An interface is like a class without fields, parent, and protected words; i.e., it just has a method map. If a class implements an interface, its method map contains a pointer to the method map of the interface. The positive offsets in the map are reserved for class methods; therefore, interface map pointers have negative offsets. Interfaces have offsets that are unique throughout the system, unlike class selectors, whose offsets are only unique for the classes where the selector is available (invocable).

This structure means that interface selectors have to perform one indirection more than class selectors to find their method. Their body contains the interface map pointer offset in the class method map, and the method offset in the interface method map. The `does>` action for an interface selector is, basically [as in Figure One] where `object-map` and `selector-offset` are first fields and generate no code.

As a concrete example, consider the following code:

```
interface
  selector if1sel1
  selector if1sel2
end-interface if1

object class
  if1 implementation
  selector cl1sel1
  cell% inst-var cl1iv1

' m1 overrides construct
```

---

[2] This is Self [chambers&ungar89] terminology; in C++ terminology: virtual function table.

```
' m2 overrides iflsell
' m3 overrides iflsel2
' m4 overrides cllsel2
end-class cl1

create obj1 object dict-new drop
create obj2 cl1    dict-new drop
```

The data structure created by this code (including the data structure for `object`) is shown in the figure [opposite], assuming a cell size of four.

Related Work

For a comparison with the Neon model, you just have to compare the properties of the Neon model presented in the last issue with the properties presented here.

Another well-known publication is [pountain87]. However, it is not really about object-oriented programming, because it hardly deals with late binding. Instead, it focuses on features like information hiding and overloading that are characteristic of modular languages like Ada (83).

There are also many other papers on object-oriented Forth extensions; E.g., [rodriguez&poehlman96] lists 17 and [mckewan97] lists six. In the rest of this section, I will discuss two systems that have the implementation using method maps in common with the package discussed here.

The model of [zsótér96] makes heavy use of an active object (like `this` in my model): The active object is not only used for accessing all fields, but also specifies the receiving object of every selector invocation; you have to change the active object explicitly with `{ ... }`, whereas in my model it changes more or less implicitly at `m: ... ;m`. Such a change at the method entry point is unnecessary with the [zsótér96] model, because the receiving object is the active object already; on the other hand, the explicit change is absolutely necessary in that model because, otherwise, no one could ever change the active object.

The model of [paysan94] combines information hiding and overloading resolution (by keeping names in various wordlists) with object-oriented programming. It sets the active object implicitly on method entry, but also allows explicit changing (with `>o ... o>` or with `with ... endwith`). It uses parsing and state-smart objects and classes for resolving overloading and for early binding: the object or class parses the selector and determines the method from this. If the selector is not parsed by an object or class, it performs a call to the selector for the active object (late binding), like [zsóter96]. Fields are always accessed through the active object. The big disadvantage of this model is the parsing and the state-smartness, which reduces extensibility and increases the opportunities for subtle bugs; essentially, you are only safe if you never tick or `postpone` an object or class.

Acknowledgments

Marcel Hendrix provided helpful comments on the paper. András Zsótér and Bernd Paysan helped me with the related works section.

References

[chambers&ungar89] Craig Chambers and David Ungar. "Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[ertl89] M. Anton Ertl. http://www.complang.tuwien.ac.at/papers/ertl89.ps.Z "GRAY - ein Generator für rekursiv absteigende Ybersetzer." Praktikum, Institut für Computersprachen, Technische Universität Wien, 1989. In German.

[ertl97] M. Anton Ertl. http://www.complang.tuwien.ac.at/papers/ertl97.ps.gz "GRAY - ein Generator für rekursiv absteigende Ybersetzer." In *Forth-Tagung*, Ludwigshafen, 1997. In German.

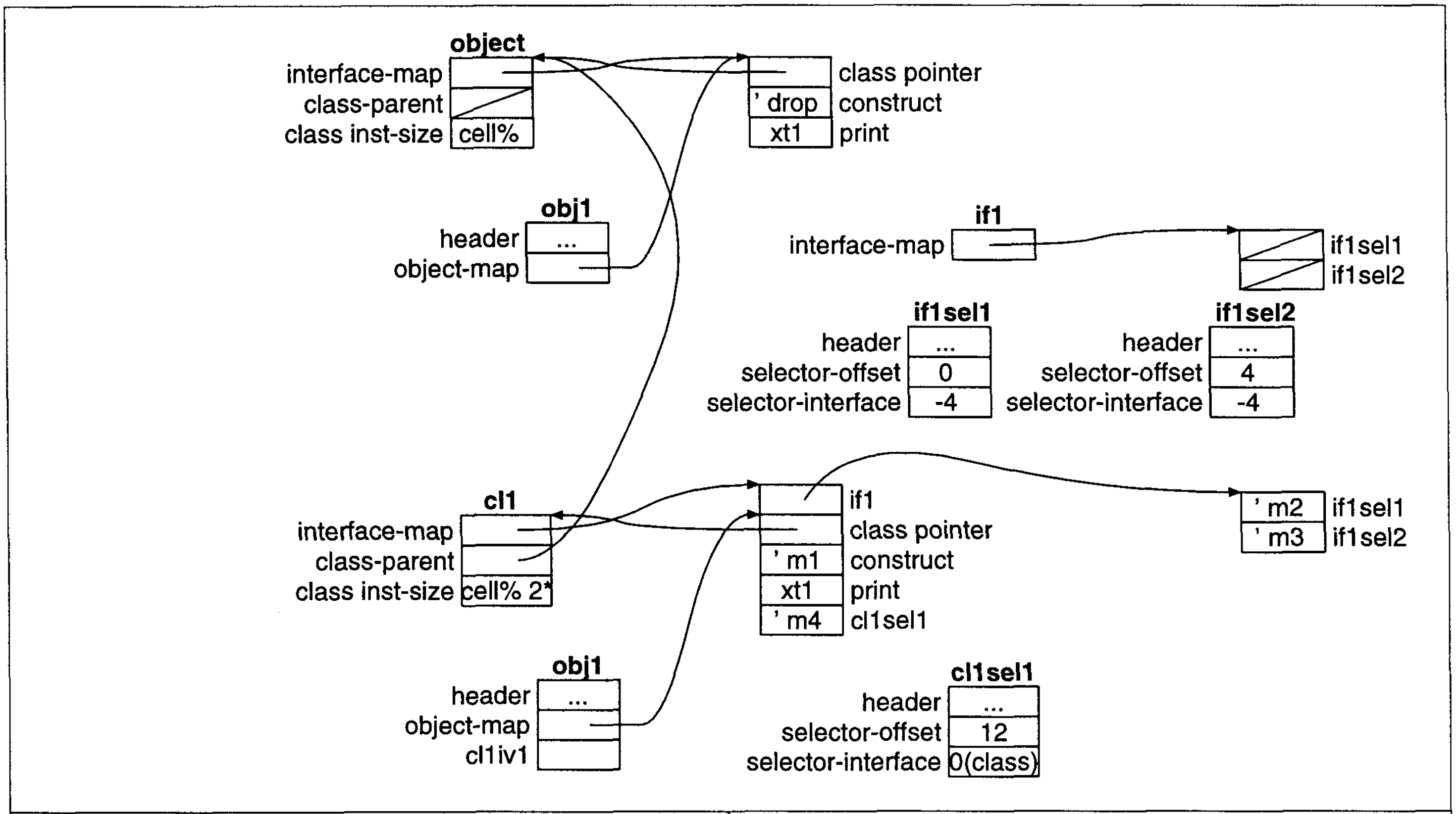
[mckewan97] Andrew McKewan. "Object-oriented programming in ANS Forth." *Forth Dimensions*, March 1997.

[paysan94] Bernd Paysan. "Object oriented bigFORTH." *Vierte Dimension*, **10** no. 2, 1994. An implementation in ANS Forth is available at http://www.informatik.tu-muenchen.de/paysan/oof.zip.

[pountain87] Dick Pountain. *Object-Oriented Forth*. Academic Press, London, 1987.

[rodriguez&poehlman96] Bradford J. Rodriguez and W. F. S. Poehlman. "A survey of object-oriented Forths." *SIGPLAN Notices*, pages 39–42, April 1996.

[zsótér96] András Zsótér. "Does late binding have to be slow?" *Forth Dimensions*, **18** no. 1 pp. 31–35, 1 1996. An implementation in ANS Forth is available at http://www.forth.org/fig/oopf.html.



Glossary

`bind ( ... "class" "selector" -- ... )`
Execute the method for `selector` in `class`.

`<bind> ( class selector-xt -- xt )`
`xt` is the method for the selector `selector-xt` in `class`.

`bind' ( "class" "selector" -- xt )`
`xt` is the method for `selector` in `class`.

`[bind] ( compile-time: "class" "selector" -- )`
`( run-time: ... -- ... )`
Compile the method for `selector` in `class`.

`class ( parent-class -- align offset )`
Start a new class definition as a child of `parent-class`. `align offset` are for use by `field`, etc.

`class->map ( class -- map )`
`map` is the pointer to `class`'s method map; it points to the place in the map to which the selector offsets refer (i.e., where `object-maps` point).

`class-inst-size ( class -- addr )`
Used as `class-inst-size 2@ ( class -- align size )`, gives the size specification for an instance (i.e., an object) of `class`.

`class-override! ( xt sel-xt class-map -- )`
`xt` is the new method for the selector `sel-xt` in `class-map`.

`construct ( ... object -- )`
Initializes the data fields of `object`. The method for the class `object` just does nothing `( object -- )`.

`current' ( "selector" -- xt )`
`xt` is the method for `selector` in the current class.

`[current] ( compile-time: "selector" -- )`
`( run-time: ... -- ... )`
Compile the method for `selector` in the current class.

`current-interface ( -- addr )`
This variable contains the class or interface currently being defined.

`dict-new ( ... class -- object )`
`allot` and initialize an object of class `class` in the dictionary.

`drop-order ( class -- )`
Drops `class`'s wordlists from the search order. No check is made whether `class`'s wordlists are actually on the search order.

`end-class ( align offset "name" -- )`
`name` execution: `-- class`
Ends a class definition. The resulting class is `class`.

`end-class-noname ( align offset -- class )`
Ends a class definition. The resulting class is `class`.

`end-interface ( "name" -- )`
`name` execution: `-- interface`
Ends an interface definition. The resulting interface is `interface`.

`end-interface-noname ( -- interface )`
Ends an interface definition. The resulting interface is `interface`.

`exitm ( -- )`
`exit` from a method; restore old `this`.

`heap-new ( ... class -- object )`
`allocate` and initialize an object of class `class`.

`implementation ( interface -- )`
The current class implements `interface`. I.e., you can use all selectors of the interface in the current class and its descendents.

`init-object ( ... class object -- )`
Initializes a chunk of memory (`object`) to an object of class `class`; then performs `construct`.

`inst-value`
`( align1 offset1 "name" -- align2 offset2 )`
`name` execution: `-- w`
`w` is the value of the field `name` in `this` object.

`inst-var`
`( align1 offset1 align size "name" -- align2 offset2 )`
`name` execution: `-- addr`
`addr` is the address of the field `name` in `this` object.

`interface ( -- )`
Starts an interface definition.

`;m ( colon-sys -- ) ( run-time: -- )`
End a method definition; restore old `this`.

`m: ( -- xt colon-sys ) ( run-time: object -- )`
Start a method definition; `object` becomes new `this`.

`method ( xt "name" -- )`
`name` execution: `... object -- ...`
Creates selector `name` and makes `xt` its method in the current class.

`object ( -- class )`
The ancestor of all classes.

`overrides ( xt "selector" -- )`
Replace default method for `selector` in the current class with `xt`. `overrides` must not be used during an interface definition.

`[parent] ( compile-time: "selector" -- )`
`( run-time: ... object -- ... )`
Compile the method for `selector` in the parent of the current class.

`print ( object -- )`
Prints the object. The method for the class `object` prints the address of the object and the address of its class.

`protected ( -- )`
Set the compilation wordlist to the current class's wordlist

`public ( -- )`
Restore the compilation wordlist that was in effect before the last `protected` that actually changed the compilation wordlist.

`push-order ( class -- )`
Add `class`'s wordlists to the search-order (in front).

`selector ( "name" -- )`
`name` execution: `... object -- ...`
Creates selector `name` for the current class and its descendents; you can set a method for the selector in the current class with `overrides`.

`this ( -- object )`
The receiving object of the current method (a.k.a. *active object*).

`<to-inst> ( w xt -- )`
Store `w` into the field `xt` in `this` object.

`[to-inst] ( compile-time: "name" -- )`
`( run-time: w -- )`
Store `w` into field `name` in `this` object.

`to-this ( object -- )`
Sets `this` (used internally, but useful when debugging).

`xt-new ( ... class xt -- object )`
Makes a new object, using `xt ( align size -- addr )` to get memory.

EuroForth '97

EMBEDDED COMMUNICATIONS

Oxford, England
September 26–28 1997

Conference proceedings will be published, and will be available immediately after the conference from the Conference Organiser. They will also be published by the Forth Interest Group. There is a refereed section of the proceedings, for those who desire peer review of their work.

Delegates from all parts of Europe and North America are expected. EuroForth is a friendly conference at which time is made available for meeting people, for informal discussions, and for contacts. Enjoy the casual pleasure of punting on the River Cherwell on Friday evening and the formal elegant dinner at St. Edmund Hall on Saturday.

EuroForth '97 is being held in the lovely setting of St. Anne's College, Oxford. St. Anne's was founded in 1879, and was the first Oxford institution to offer University education to women. In 1979, it opened its doors to men, as well. This college is within easy walking distance to the city centre, yet is free from the crowds of central Oxford.

EuroForth, the annual European Forth Conference will focus this year on the increasing use of communications within applications, ranging from embedded controllers connected to the outside world through modems, to Internet payment engines delivering secure cash transfers from domestic PCs, and including the process-control environment which connects peripherals as diverse as door-access controllers, mass spectrometers, commercial laundry controllers, and walking robots to management systems.

These and many more are the domain of modern Forth systems, and papers are sought about the following topics:

- Embedded networking—
  including Fieldbusses and embedded TCP/IP
- Portable software tools
- New development environments
- Formal methods
- Virtual machines
- Any Forth-related topic

The Conference Organiser, EuroForth '97
c\o Microprocessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
England
Tel: +44 1703 631441
Fax: +44 1703 339691
net: mpe@mpeltd.demon.co.uk