F O R T H
D I M E N S I O N S

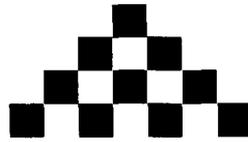SMART RAM

TESTING TOOLKIT

DYNAMIC MEMORY ALLOCATION

DYNAMIC VIRTUAL MEMORY MANAGEMENT

68000 NATIVE-CODE FORTH (II)
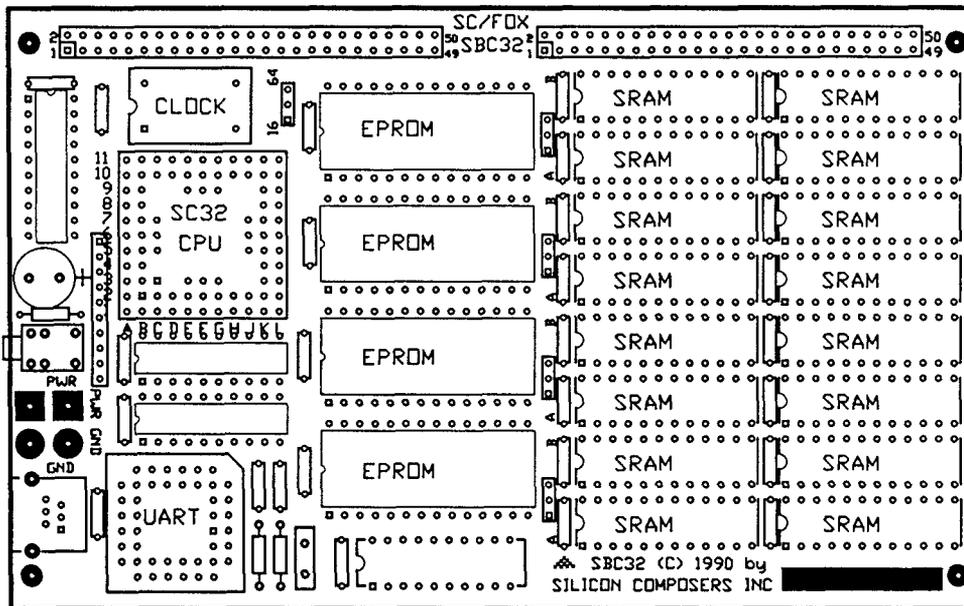
FORTH
D I M E N S I O N S

EDITORIAL

If you haven't paid close attention to the growth of on-line Forth activity, you may be surprised. Forth programs, debates, questions, news, and insights are being shared between several BBSs and larger communication systems—including some international ones—thanks to their respective sysops and to both electronic and manual gateways between systems. There is more reason than ever to tune in to the on-line Forth community. *FD*'s "Reference Section" lists the electronic resources we find and, despite some past problems, we try to keep it both current and complete. (You can help by informing us of changes and additions.)

If you didn't log on in August, you missed meetings scheduled with Bill Ragsdale and Glen Haydon. To further encourage your virtual presence on at least one of these electronic venues, upcoming guest conferences on GEnie's Forth RoundTable include:

Dick Miller, President of Miller Microcomputer Services
"To DOS or Not to DOS"
Thursday, September 20
9:30 p.m. Eastern/6:30 p.m. Pacific

Jef Raskin, originator of the Apple Mac and the Canon Cat
"What Happened to the Cat?"
Wednesday, October 17
9:30 p.m. Eastern/6:30 p.m. Pacific

(Note that the October conference is on Wednesday instead of the usual Thursday.)

*　　　*　　　*

Speaking of Glen Haydon (of MVP-FORTH, WISC, etc.), he has completed a significant revision of his book *All About Forth*. It has long been popular as the working reference volume of definitions, implementation examples, and relevant details about a widely used set of Forth words. But the recent, greatly revised and expanded version makes the book an annotated glossary of practically all Forth words in common usage, in all the primary dialects. Implementation examples are given in high-level Forth or 8086/88 assembly language to help clarify the text of a word's definition. When in doubt, just look it up! This essentially new book is, in my opinion, an important contribution to every Forth programmer's workbench. Look for it on the FIG Mail Order Form.

**If you live in Memphis,
don't blame us...**
Publishing News reported that seventy-five percent of monthly magazines were delivered late in early 1990, an increase over last year. Memphis, Tennessee had the worst record (none delivered on time) and San Mateo, California had the best record (one hundred percent).

—*Marlin Ouverson*
Editor

About 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. Many research hardware and software designs that will advance the general state of the art. 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.

LETTERS

Language of Choice

Dear Sir,

I agree with Philip Koopman Jr.'s call to market Forth for what it is, the premier controller language (*FD* XII/1). I'm not a professional programmer, but I am a fan of Forth.

Koopman hits the mark by spotlighting the perception of Forth. I'm looking at a well-balanced article written by Dara Pearlman and published in the September 1983 issue of *Popular Computing*. It led me to send for the fig-FORTH listing years ago. But the article's title says it all: "Forth Inspires a Fanatic Following—But Why Hasn't It Taken the World by Storm?" Next to it is a color photograph of the original FIG five in their FIG t-shirts and waving copies of *Forth Dimensions*. I fear that, in the public's mind, Forth is a strange language promoted by even stranger fanatics.

The perception of a language is its theme, the one-sentence statement that defines the problem it solves for the computing public (preferably a large segment of it). BASIC, for example, was created to introduce computers to Social Science students at Dartmouth. It fulfilled the need of the novice for an easy way to learn to use a computer. That BASIC can now be found as a complex, structured, compiled language is beside the point.

The result, Koopman points out, is a lack of marketing focus in promoting Forth. This has caused Forth to lose the synergism from a strong, consistent marketing sponsor. Where would BASIC be if not for Dartmouth, Microsoft, Apple, and all the other home computer companies and universities who found in BASIC an ideal vehicle to sell their products and services? Forth has to be marketed for what it is, the ideal way for a business to save time and money programming controllers. That it

can do much more is beside the point.

Koopman's request for programmers to go public with their experiences in Forth projects is good. We may think Forth is great, but it would help a lot if some hard facts about time and money would back up that opinion for businesses to act upon.

For publicity, I believe a Forth version of Steve Ciarcia's "Circuit Cellar, Ink." would be an ideal showcase. The Forth chips are out there. So why not encourage the Forth hackers? The Forth hardware people should take a good look at this.

Koopman calls us to actively promote Forth now. The expansion of microcomputers into everything gives Forth a rare opportunity to be like those caterpillars that march around in a circle until they die. It does need to focus on a single marketing goal. Why not the one for which it was created?

Yours truly,

Walter J. Rottenkolber
P.O. Box 936
Visalia, California 93279

Dear Editor,

I agree with most of Philip Koopman, Jr.'s article. To me, additional definitions are so logical and not too hard to follow, but others seem to consider it witches' brew. Consider the Forth definition in Gespac's *Glossary of Technical Terms* (1989):

> *Forth is a high level language that is based on a very small kernel. It allows user[s] to define instructions which can then be used to define other, higher level instructions, which can be used to define higher level instructions and so on .... Experienced Forth users worship that language. Forth, however, has been banned by most corporations because*

> *the "Forth Dictionary" tends to become a dialect of the programmer, making it almost impossible to read and be maintained by any other programmers...*

It does seem that embedded systems, without a kernel, that can be used to mechanize operations for well-defined situations, could help. With enough systems or parts available that can easily be used with all CPUs, use could not help [but] increase acceptance.

It's not the whole answer...

Phil Chadwick
358 Thompson Mill Road
New Hope, Pennsylvania 18938

Standards Notwithstanding

I have been using Forth continuously (but not exclusively) since 1975. I started with my own creation, which has grown into a very viable system. I have also used the most notable commercial versions. Chuck Moore's statement ("Best of GEnie," *FD* XI/5) inspires the following comments based upon experience:

1. Classic Forth consists of a few elementary definitions (' is tick an address, ! is store a word, etc.), a philosophy of programming tasks, and a definite form in which the program is expressed (numbers precede commands, names follow, and space is the only delimiter). The philosophy seems to be the hard part to grasp for the uninitiated, perhaps because it is always cryptically demonstrated and is seldom simply expressed.

2. Complex operations do not survive the test of time. Simpler ways evolve, or they are returned to the vocabulary.

3. VOCABULARY has a purpose only if

it speeds up compilation and FORGETing. It is great for task management. Other uses are probably abuses.

4. My Forth has been recreated just about every year. Little has gone untried, and useless things have fallen by the wayside. This is an inherent part of the life of Forth.

5. Forth and Standard Forth are contradictions of definitions (terms). Any particular Forth contains only the necessary common definitions, adaptations, and extensions to do a particular job. A Standard Forth would require all definitions to be common, even adaptations and extensions (as do BASIC, Modula II, etc.).

6. Some programmers cannot comprehend the philosophy of Forth; this relegates them to use other languages and lose the benefits of Forth, or to use a Forth compiler that does not require such understanding (if somebody can invent one).

7. The ideal Forth machine has not yet been demonstrated. Much of Forth can be parallel processed (but mostly not in the commonly thought-of way).

8. QWERTY keyboards (or some substitute) are necessary for common editing (data-entry stage). All other needs are managed better by a logic tree and a very few keys. Forth is a natural at executing logic trees.

9. IBM-type PCs are a collection of design compromises incorporated to lock customers into a commercial operation that continuously requires additional expenditure for both software and hardware. Extremely expensive, high-speed processors are required to overcome this burden. The efficiency of Forth permits a very inexpensive, moderate-speed processor to outperform the IBM type.

10. Experiment until you fully understand, and program only what you do fully understand. This demonstrates the definition of "smart." Forth is the most natural execution of this philosophy.

11. Because of its adaptability and extensibility, real Forth is master of what it has to do. Standard Forth would be master-of-none, but it could become a commercial, general-purpose master-of-none like most other standard languages.

12. PUSH and POP are longer words for >R and R>. Long natural words are for users of applications; short, cryptic symbols are for programmers.

13. Multiply and divide are overused, but require less understanding of the situation (in a conniving sense). Methods using logic operations are often much more efficient. SHIFT and 2*, 2/, 4*, 4/, 8*, 8/, etc. are often better choices if you understand what you are doing.

14. Floating-point arithmetic is bad. But it is no joke if its abuses are not accounted for when calculating something like the strength of the floor you are standing on. Errors can far exceed tolerances.

15. Source code is withheld because of mass commercial intent or because it is not presentable (often both). Forth is best used for customized adaptations which have little mass commercial appeal. However, publish the source code of an interesting Forth program—no matter how unpresentable—and the Forth community will correct it, improve it, adapt it, and return it for little or no cost. Beat that, if you can.

Forth is the best programming language for those who can master it and bend it to their needs. I will be using a flexible, adaptable version of Forth for the rest of my life (standards committee notwithstanding).

Fred F. Kloman
3533 DeLeone Road
San Marcos, California 92069

He Wants a New View
In the interest of software quality, wouldn't it be nice if there were a utility which would help in the data-flow analysis of a colon definition? Perhaps a modification to VIEW in F-PC. The idea is to have the colon definition displayed, one word per line, with its stack comment. E.g.,

```
?EMIT    (c -- )
DUP      (c -- c c)
IF       (f -- c)
EMIT     (c -- )
ELSE     ( -- )
DROP     (c -- )
THEN     ( -- )
;        ( -- )
```

The stack comments could be checked to see that all inputs and outputs were properly matched, a big time saver; and it would encourage the use of stack comments.
    Yours truly,

Ken Kupisz
Ontario Hydro
700 University Avenue
Toronto, Ontario
Canada M5G 1X6

[And remember to account for words that leave a varying number of stack items, depending on run-time conditions...—Ed.]

Return to DO-FOREVER
Dear Marlin,
    Oops, DO-FOREVER as printed in *Forth Dimensions* XII/1 won't even load, much less run. Somewhere in all the changes it correctly acquired an IF but failed to acquire a matching ENDIF. The minimal correction would be to insert an ENDIF immediately before the semi-colon. In my letter of January 23, I tried to be more elegant and added an error message in case the user types an undefined word. [See Figure One.]
    Best wishes,

Tom Napier
One Lower State Road
North Wales, Pennsylvania

```
: DO-FOREVER   ( repeat next word until keyboard input )
  -FIND
  IF DROP CFA ( this is fig-FORTH )
    BEGIN DUP EXECUTE ?TERMINAL
    UNTIL DROP
  ELSE CR ." Do what?"
  ENDIF  ;
```

Figure One. DO-FOREVER as it ought to be.

DYNAMIC
VIRTUAL MEMORY
MANAGEMENT

ANTERO TAIVALSAARI - TAMPERE, FINLAND

■

In this article we shall present dynamic virtual memory management extensions to Forth. With the extensions, persistent storage space for data items can be allocated and deallocated dynamically in Forth's virtual memory, and the sizes of the data items can be changed at any time. A simple heap-based memory compaction mechanism is used in order to keep the blocks unfragmented. The extensions are proven functional in Laxen & Perry's F83, but they should be quite easily portable to other Forth models, too.

Introduction

Forth is a multidimensional programming language which has many characteristics that are normally related only to the operating system. One of the special characteristics of Forth is its unique way to handle virtual memory. In Forth, a virtual memory device—typically a disk drive—is accessed in 1K blocks by using a very limited but powerful set of words. Disk blocks can be loaded to main memory buffers by giving a block number and the command BLOCK. The buffer can then be examined and modified; the Forth system will automatically save the buffer into the disk if the user marks the buffer updated.

...can also be used for dynamic databases not tied to traditional models.

With the basic virtual memory commands of Forth, different kinds of database systems are quite easily implemented. However, as typical of most database management systems based on traditional storage models (hierarchical, network or relational), these database systems tend to be rather dependent on physical aspects of the storage system and data. Such things as file sizes, data item sizes, and physical locations of data must occasionally be taken into serious consideration at the programmer's level. Also, traditional storage models restrict the modifiability of the database. For example, the sizes of data items must usually be determined during the creation of the database, which can be very inconvenient if the description (schema) of the database is to be changed

```
Screen 0
*****************************************************************
* Antero Taivalsaari
* Ruovedenkatu 13 D 54
* SF-33720 Tampere
* FINLAND
* Electronic mail: tsaari@tukki.jyu.fi (128.214.7.5)
*****************************************************************
* FILE.....: DISKHEAP.BLK
* PURPOSE..: Dynamic virtual memory management extensions
*          : to Forth-83.
* REQUIRES.: F83.COM
* AUTHOR...: Antero Taivalsaari
* DATE.....: 29.10.1989
*****************************************************************

Screen 1
\ Constants, addresses, variables                   Apt 29.10.89

0    CONSTANT pointerBlk   \ block for persistent pointers
6    CONSTANT /handle      \ size of handle (= 6 bytes)
170  CONSTANT handles/blk  \ max# of handles per block
1020 CONSTANT maxBytes/blk \ max# of databytes per block
0    CONSTANT firstHandle  \ index of first handle

: lastHandle       pointerBlk block ;     \ index of last handle
: firstDataBlk     pointerBlk block 2+ ;  \ blk# of first datablk
: lastDataBlk      pointerBlk block 4 + ; \ blk# of last datablk
: #ofFreeHandles   pointerBlk block 6 + ; \ # of free handles

VARIABLE reUseHandles       \ determines whether handles may be
reUseHandles on             \ reused after they are freed.
-->
```

```
Screen 20
\ DISKHEAP.BLK                               Apt 29.10.89

Dynamic virtual memory management.

Example:

variable var1          \ defines variables for storing indexes
variable var2          \ to handles.
createHeap test.blk    \ creates a new diskheap 'test.blk'.
20 allocate var1 !     \ allocates 20 bytes referred by var1.
var1 @ area /area type \ types contents of var1.
40 allocate var2 !     \ allocates 40 bytes referred by var2.
50 var1 @ resize       \ resizes data-area of var1 to 50 bytes.
var2 @ free            \ disposes data-area of var2.

Screen 21
\ Constants, addresses, variables                   Apt 29.10.89

DiskHeap contains three different memory areas: pointer block,
handle block(s), and data block(s). Block 0 (pointerBlk)
contains persistent pointers to handles and data. Handles,
which are static references to the data-area, begin from block
1. Handle blocks are followed by data blocks which continue
to the end of file. Since one handle block can contain only 170
handles, data blocks must occasionally be moved. When # of
handles in the last handle block exceeds 170, a new block is
requested from DOS and the first data block is moved to this
new block. The original first data block can then be used
for storing handles.
```

```
\ Variables, offsets                              Apt 29.10.89

VARIABLE currentBlk        \ blk# of current datablock
VARIABLE currentHandle     \ index# of current handle
VARIABLE areaSize          \ size of current data area

: 'blk           ; immediate    ( handleAddress -- 'blk )
: 'offset        2+ ;           ( handleAddress -- 'offset )
: 'size          4 + ;          ( handleAddress -- 'size )

: free/blk                      ( -- 'freeSize )
    currentBlk @ block ;

: refs/blk                      ( -- '#ofReferences )
    free/blk 2+ ;
-->
```

```
Screen 3
\ handle>virtual  inHandleRange?  virtual>memory  Apt 29.10.89

Defer doError
' abort is doError

: handle>virtual        ( handleIndex -- offset block )
    handles/blk /mod swap /handle * swap 1+ ;

: inHandleRange?        ( handleIndex --  )
    firstHandle lastHandle @ 1- between not
    IF ." - Not in handle range " doError THEN ;

: virtual>memory        ( offset block -- address )
    block + ;
-->
```

```
Screen 4
\ handle  free?  roomForNewHandle?  tooBig?       Apt 29.10.89

: handle                ( handleIndex -- handleAddress )
    dup inHandleRange? handle>virtual virtual>memory ;

: free?   'blk @ 0= ;   ( handleAddress -- flag )

: roomForNewHandle?     ( -- flag )
    lastHandle @ handle>virtual nip firstDataBlk @ < ;

: tooBig?               ( size --  )
    maxBytes/blk >
    IF ." - Cannot allocate over 1020 byte areas "
        doError
    THEN ;
-->
```

```
Screen 5
\ convertHandles                                  Apt 29.10.89

: convertHandles    ( oldBlock lastBlock --  )
    over currentBlk !
    lastHandle @
    refs/blk @ 0 ?DO
      BEGIN 1- dup
            handle dup currentHandle !
            'blk @ 3 pick =
      UNTIL
      over currentHandle @ 'blk ! update
    LOOP drop 2drop ;
-->
```

```
Screen 22
\ Variables, offsets                              Apt 29.10.89

Handles are referred with a simple 16-bit index, starting
from 0. Handle indexes must be between 'firstHandle-lastHandle'.
Each handle contains three 2-byte fields, which are:
     - 'blk     blk# of data-area,
     - 'offset  offset to the beginning of data in the block,
     - 'size    current size of data-area.

In the beginning of each data block there are following fields:
     - free/blk     # of free data bytes in this block,
     - refs/blk     reference count to this data block (tells
                    how many handles refer to this data block).
```

```
Screen 23
\ handle>virtual  inHandleRange?  virtual>memory  Apt 29.10.89

doError         vectored error handling.


handle>virtual  returns virtual address of a handle.


inHandleRange?  ensures that handle index is between the
                values of variables 'firstHandle' and
                'lastHandle'.

virtual>memory  fetches a virtual block to memory and returns
                the sum of its address and the parameter
                'offset'.
```

```
Screen 24
\ handle  free?  roomForNewHandle?  tooBig?       Apt 29.10.89

handle          fetches a handle from virtual memory and
                returns its address in a disk buffer.

free?           tests whether handle is free
                (that is: field 'blk is zero).
roomForNewHandle?
                tests whether current handle block has room
                for one more handle.
tooBig?         ensures that we shall not try to allocate
                data-areas of over 1020 bytes.
```

```
Screen 25
\ convertHandles                                  Apt 29.10.89

convertHandles      this word is used when a data block is
                    moved to another block. All references
                    to 'oldBlock' are changed to refer to
                    'newBlock'.
```

```
Screen 6
\ firstDataBlk>newBlk  makeNewHandle          Apt 29.10.89

: firstDataBlk>newBlk     ( -- )
    1 more
    1 lastDataBlk +! update
    firstDataBlk @ lastDataBlk @ 2dup copy convertHandles
    1 firstDataBlk +! update
    firstDataBlk @ currentBlk ! ;

: makeNewHandle           ( -- handleIndex )
    roomForNewHandle? not
    IF firstDataBlk>newBlk THEN
    lastHandle @
    1 lastHandle +! update ;
-->

Screen 7
\ findOldHandle  giveHandle               Apt 29.10.89

: findOldHandle       ( -- handleIndex )
    lastHandle @
    BEGIN 1- dup
         handle free?
    UNTIL
    -1 #ofFreeHandles +! update ;

: giveHandle          ( -- handleIndex )
    #ofFreeHandles @
    reUseHandles @ and
    IF   findOldHandle
    ELSE makeNewHandle
    THEN ;
-->

Screen 8
\ roomInCurrentBlk?  makeNewDataBlk  >endOfData   Apt 29.10.89

: roomInCurrentBlk?      ( size -- flag )
    free/blk @ > not ;

: makeNewDataBlk          ( -- )
    1 more
    1 lastDataBlk +! update
    lastDataBlk @ currentBlk !
    maxBytes/blk free/blk !
    0 refs/blk ! update ;

: >endOfData              ( -- address )
    b/buf free/blk @ - ;

->

Screen 9
\ findNextBlk  allocateRoom                Apt 29.10.89

: findNextBlk            ( size -- )
    false lastDataBlk @ currentBlk @ ?DO
      i currentBlk !
      over roomInCurrentBlk?
      IF drop true leave THEN
    LOOP nip
    not IF makeNewDataBlk THEN ;

: allocateRoom       ( size -- offset blk )
    >endOfData >r r@ currentBlk @ virtual>memory
    over erase negate free/blk +!
    1 refs/blk +! update
    r> currentBlk @ ;
-->
```

```
Screen 26
\ firstDataBlk>newBlk  makeNewHandle          Apt 29.10.89

firstDataBlk>newBlk  requests a new block from DOS and copies
                     the first data block to this new block.
                     This word is used when old data blocks are
                     changed to handle blocks.

makeNewHandle        creates a new handle. If there is no room
                     for a new handle in the current handle
                     block, then 'firstDataBlk>newBlk' is
                     executed.

Screen 27
\ findOldHandle  giveHandle               Apt 29.10.89

findOldHandle        finds a free handle from existing
                     handles. This word is used if 'reUseHandles'
                     is ON, and #ofFreeHandles is > 0.

 giveHandle          gives a free handle. If 'reUseHandles' is
                     OFF or #ofFreeHandles = 0, then a new
                     handle must be created.

Screen 28
\ roomInCurrentBlk?  makeNewDataBlk  >endOfData   Apt 29.10.89

roomInCurrentBlk?    tests whether current data block has room
                     for 'size' more bytes.

makeNewDataBlk       creates a new empty data block and sets
                     the relevant pointer values. Initially
                     there are no handles referring to this
                     block ('refs/blk' = 0) and the free space
                     in the block is 'maxBytes/blk'.

>endOfData           returns the address of the first free byte
                     in the current data block.

Screen 29
\ findNextBlk  allocateRoom                Apt 29.10.89

findNextBlk          finds the next data block having room
                     for 'size' bytes. This block is then made
                     current. If existing blocks do not have
                     enough room, a new block is created.

allocateRoom         allocates 'size' bytes from current data
                     block and returns the virtual address
                     of this new data-area. 'free/blk' and
                     'refs/blk' are updated. The allocated
                     area is erased to all zeros.
```

```
: findRoom           ( size -- offset blk )
    dup roomInCurrentBlk? not
    IF dup findNextBlk THEN
    allocateRoom ;

: Allocate           ( size -- handleIndex )
    dup tooBig?
    giveHandle >r
    dup findRoom
    r@ handle >r r@ 'blk 2!
    r> 'size ! update
    ( save-buffers ) r> ;

-->
```

```
Screen 11
\ handleUpdate                              Apt 29.10.89

: handleUpdate       ( +/-n offset -- )
    lastHandle @
    refs/blk @ 0 ?DO
      BEGIN 1- dup
            handle dup currentHandle !
            'blk @ currentBlk @ =
      UNTIL
      over currentHandle @ 'offset @ > not
      IF 2 pick currentHandle @ 'offset +! update THEN
    LOOP drop 2drop ;


-->
```

```
Screen 12
\ moveData                                  Apt 29.10.89

: moveData           ( +/-n handleAddress -- )
    dup 'offset @ swap 'size @ +
    2dup handleUpdate
    >endOfData over - >r
    currentBlk @ virtual>memory
    tuck + r> move update ;

-->
```

```
Screen 13
\ Free                                      Apt 29.10.89

: Free     ( handleIndex -- )
    dup handle >r r@ free?
    IF r> 2drop ." - Handle already free " doError THEN
    r@ 'blk @ currentBlk !
    r@ 'size @ negate r> moveData
    handle dup 'blk off update
    'size @ free/blk +!
    -1 refs/blk +! update
    1 #ofFreeHandles +! update
    ( save-buffers ) ;

-->
```

```
Screen 30
\ findRoom  Allocate                        Apt 29.10.89
```

findRoom   locates the next data block that has enough room
                  for 'size' bytes and return the virtual address
                  of this new data-area.

****
Allocate   allocates 'size' bytes from diskHeap and returns
**************   the handle index of this new data-area. The
                  new data-area is erased to all zeros.

                  This is one of the basic commands in dynamic
                  virtual memory management.

```
Screen 31
\ handleUpdate                              Apt 29.10.89
```

handleUpdate   increments or decrements the 'offset' fields of
                  handles pointing to the current data block
                  according to the parameter 'n' in case
                  the 'offset' of a handle is >= than the parameter
                  'offset'. This word is used when the size of
                  an existing data-area changes.
                  Note that thanks to reference counter 'refs/blk'
                  we don't necessarily have to loop through all
                  the handles.

```
Screen 32
\ moveData                                  Apt 29.10.89
```

moveData   moves the data-areas above the data-area referred
                  by 'handleIndex' in the current data block
                  upwards or downwards in the block according to
                  the parameter 'n' (positive n = upwards, negative
                  = downwards). The 'offset' fields in handles are
                  also updated (handleUpdate). This word is used
                  when the size of an existing data-area changes.

```
Screen 33
\ Free                                      Apt 29.10.89
**************
```
Free   frees existing data-areas. The data block is
**************   compressed by moving the rest of data in the
                  block downwards; the counter 'free/blk' is
                  incremented with the size of freed data-area,
                  and the reference counter 'refs/blk' is
                  decremented by one. The handle to the
                  deallocated data-area is marked free by storing
                  '0' into the 'blk' field of the handle. Then the
                  number of free handles is incremented by one.

                  This is one of the basic commands in dynamic
                  virtual memory management.

```
Screen 14
\ area  /area                               Apt 29.10.89


: area                ( handleIndex -- address )
    handle dup free?
    IF drop ." - Handle not in use " doError THEN
    dup 'size @ areaSize !
    2@ virtual>memory ;

: /area
    areaSize @ ;     ( -- areaSize )



-->




Screen 15
\ expandCurrentBlk                          Apt 29.10.89


: expandCurrentBlk        ( sizeMore handleIndex -- )
    dup handle >r r@ 'size @
    IF   over r> moveData
    ELSE >endOfData currentBlk @ r> 'blk 2!
    THEN
    over negate free/blk +!
    dup area /area + 2 pick erase update
    handle 'size +! update ;



-->




Screen 16
\ moveToOtherBlk                            Apt 29.10.89


: moveToOtherBlk     ( sizeMore handleIndex -- )
    2dup handle 'size @ + dup
    findNextBlk allocateRoom          \ allocate new room
    2dup >r >r virtual>memory         \ fetch to memory
    over area swap /area cmove update \ move to new room
    dup handle >r r@ 'blk @ currentBlk ! \
    r@ 'size @ negate r> moveData     \ free existing space
    handle >r r@ 'size @ free/blk +!  \ in the original
    -1 refs/blk +! update             \ block
    r@ 'size +!
    r> r> r> rot 'blk 2! update ;     \ update handle



-->  ( huh, but it works! )




Screen 17
\ expandArea                                Apt 29.10.89


: expandArea   ( sizeMore handleIndex -- )
    swap 0 max over
    handle >r r@ 'size @ over + tooBig?
    r> 'blk @ currentBlk !
    tuck roomInCurrentBlk?
    IF   expandCurrentBlk
    ELSE moveToOtherBlk
    THEN ( save-buffers ) ;



-->
```

```
Screen 34
\ area  /area                               Apt 29.10.89
*********
area      this word is used to fetch a data-area from virtual
********* memory into a disk buffer. The word returns the
          address of the first byte of data into the parameter
          stack. A pointer to the size of the data-area is
          stored to the variable 'areaSize' for the word '/area'
*********
/area     returns the size of the current data-area.
*********
          These are basic commands in virtual dynamic memory
          management.




Screen 35
\ expandCurrentBlk                          Apt 29.10.89
expandCurrentBlk       expands the data-area referred by the
                       'handleIndex' 'sizeMore' bytes.
                       The expansion is accomplished by moving
                       all the data-areas that are located higher
                       in the block 'sizeMore' bytes upwards in
                       the block. The reserved new data space is
                       erased to all zeros.




Screen 36
\ moveToOtherBlk                            Apt 29.10.89
moveToOtherBlk         moves an existing data-area to another
                       block and at the same time expands the
                       data-area 'sizeMore' bytes.
                       This command is used when the free space
                       in the original block is insufficient
                       for 'sizeMore' byte expansion. The data
                       space in the original block is freed.




Screen 37
\ expandArea                                Apt 29.10.89
expandArea             expands the data-area referred by the
                       handle 'handleIndex' 'sizeMore' bytes. If
                       there is not enough free space in the
                       original block, the data-area is moved
                       to another block. When needed, a completely
                       new block is requested from DOS.
                       Remember that the size of a data-area cannot
                       exceed 1020 bytes.
```

```
Screen 18
\ shrinkArea  Resize                          Apt 29.10.89


: shrinkArea   ( sizeLess handleIndex -- )
    swap 0 max over
    handle >r r@ 'size @ min negate swap
    r@ 'blk @ currentBlk !
    over r> moveData
    over negate free/blk +!
    handle 'size +! update ( save-buffers ) ;

: Resize       ( newSize handleIndex -- )
    >r r@ handle 'size @ 2dup <>
    IF 2dup > IF - r> expandArea
              ELSE swap - r> shrinkArea THEN
    ELSE 2drop r> drop THEN ;
-->



Screen 19
\ createHeap  useHeap                         Apt 29.10.89


: createHeap   ( -- )        \ usage: createHeap filename
    3 create-file
    0 lastHandle !
    2 firstDataBlk !
    2 lastDataBlk !
    firstDataBlk @ currentBlk !
    0 #ofFreeHandles ! update
    maxBytes/blk free/blk !
    0 refs/blk ! update save-buffers ;

: useHeap       ( -- )       \ usage: useHeap filename
    open
    firstDataBlk @ currentBlk ! ;
```

```
Screen 38
\ shrinkArea  Resize                          Apt 29.10.89

  shrinkArea          shrinks the data-area referred by the
                      'handleIndex' 'sizeLess' bytes. This is
                      accomplished by moving all the data-areas
                      higher in the block downwards 'sizeLess'
                      bytes in the block.
  *******************
  Resize              resizes the data-area referred by the handle
  ******************* 'handleIndex' to 'newSize' bytes.

                      This is one of the basic commands in
                      dynamic virtual memory management.



Screen 39
\ createHeap  useHeap                         Apt 29.10.89
  **************
  createHeap     creates a new diskHeap. The size of a new heap
  ************** is three blocks (0 = pointer block, 1 = handle
                 block, 2 = data block).
  **************
  useHeap        takes an existing diskHeap in use.
  **************
```

frequently later.

Our original intention in the development of these dynamic virtual memory management extensions has been to create persistent object storage for the author's object-oriented, Forth-based language Kevo [Tai89, Tai90] (the language was previously called Cool, but was renamed because other languages under that name appeared to exist already). In the development of such a storage system the following design goals were listed:

- Allocated data items (objects) should persist between different sessions and programs.
- The objects should be referred to by using only a simple identifiers (values) that can be easily passed as parameters or stored to other data structures.
- Physical aspects such as file sizes, locations of objects, and sizes of objects should be unimportant to the user.
- Data objects could be allocated and deallocated dynamically "on the fly."
- The sizes of objects must be able to change dynamically.
- A simple but effective garbage collection mechanism must be provided to keep the data blocks unfragmented and the size of the database moderate.

In the literature, many different approaches to object-oriented database systems have been presented (see, for example, MSO86, Bee87, LiS88, and Low88). By analyzing several alternatives, we concluded that a simplified version of a heap-based storage system [ACC81, Har88 p.26] would be quite sufficient for our purposes. For main memory management, heap extensions to Forth have already been presented by Dress [Dre86] and Pountain [Pou87]. Therefore, our main task has actually been to modify the existing heap mechanisms to work with virtual buffers instead of main memory.

These extensions have proven to be functional as persistent object storage. However, they should be useful in any application where the sizes of virtual data items are to change frequently. Such systems include, for example, textual data bases or hypertext applications. The extensions can also be used as a kernel for new kinds of dynamic databases not bound to any of the traditional database models. Other interesting application areas might be operating system extensions to Forth; older overlay systems, for example, could be rather easily and more elegantly reimplemented with dynamic virtual memory extensions. One of our future visions has also been to build a dynamic persistent Forth, completely based on dynamic virtual memory.

Main Ideas

We shall not explain the function of the extensions very carefully. The basic principles of heap memory management are quite well documented already [Dre86]. Furthermore, our code is rather well commented and thereby should be easy to understand. A brief description of the main technical details is given, though.

Storage areas

In a heap memory management system, the storage space is usually divided into two different regions. The first one contains handles: statically located references to the data items. The actual data items are located in another memory region called the data area. The locations of items in the data area may vary, since access to the items takes place indirectly via handles.

In a virtual heap system ("diskheap"), both the handle and data areas must be located in virtual memory. In addition to these memory areas, some persistent pointers and indexes to the handle and data blocks are also needed. Therefore, in our system the disk file (heap) is divided into three different regions: pointer block, handle blocks, and data blocks. The pointer block is located at block zero and is followed by the handle blocks and data blocks, which continue to the end of file (see Figure One).

Handles

Handles are references to the data area. The location of a handle is always static, so the handles can be safely referenced despite the fact that the actual locations of data may vary. In our extensions, the handles are referenced via simple two-byte indexes. When a new data area is allocated, an index to that new data area is returned. This index must thereafter be remembered somehow, since it will be the only way to access the data area. Because handle indexes are simple values, they can be easily stored as variables or passed as parameters. In order to create complex persistent data structures, the handle indexes can, of course, be stored to other data areas in the heap. Values of handle indexes range from zero (the constant `firstHandle`) to the current maximum handle index value, which is kept in the persistent variable `last-Handle`.

Handles are six bytes long. Each handle consists of a four-byte virtual address field (block, offset) telling the location of the data area, and of a two-byte field holding the current size of the data area (see Figure Two). If the data area referred to by a handle is deallocated, a zero value is stored to the block number field (first two bytes) to mark the handle free. The special variable `reUseHandles` can be used to specify whether free handles may be reused. When `reUseHandles` is turned off, the system may work a bit faster but, at the same time, will need more space for the handles. The default is `reUseHandles on`.

Since one Forth disk block is 1024 bytes, it can contain a maximum of 170 handles. In case the number of handles in a handle block exceeds 170, a new disk block is automatically requested from DOS and the first data block is moved to the new block; the original data block then becomes a new handle block.

Data areas

Data blocks are located after the handle blocks. In the beginning of each data block, four bytes are reserved for system usage. The first two bytes (`free/blk`) tell the number of free bytes currently in the block. The next two bytes (`refs/blk`) hold a reference count telling how many handles are currently referring to this data block. This simple reference count mechanism is used to rev up the resizing of data areas. The rest of the block (1020 bytes) is used for storing data.

To avoid the need for complex garbage collection mechanisms, the data blocks are kept compacted all the time. Each time the size of a data area changes or a data area is deallocated, the rest of the data in the block is moved upwards or downwards in the buffer, so the free space in a data block is

always located in the end of the block. Since the size of a data block is only 1K and the compaction is done in a main memory buffer, the compaction can be done very quickly. Of course, when moving data, the virtual data addresses (blocks, offsets) of the handles referring to the affected blocks must also be changed, and that is more time-consuming.

When a data area is expanded, it may happen that the free space in the data block becomes inadequate. In that case, free space will be searched for in the blocks following the current data block (`cur-rentBlk`). If a data block that has room for the expanded data area is found, the data area is moved and expanded into that data block. The storage space in the original data block is then freed and compacted. In case no data blocks having enough room exist, a new block is requested from DOS. Note that we have not implemented any special mechanisms for finding free space in the existing data blocks. A heap operation (allocation, deallocation, or resizing) always leaves the latest affected block current, and free space is then searched upwards beginning from this current block. For our purposes, this mechanism has proven to be fair and fast enough.

The extensions do have some restrictions. Since virtual memory handling is done in 1K buffers, the size of a data area may not exceed 1020 (`maxBytes/Blk`) bytes. This restriction can, though, be avoided by defining higher-level words that will automatically allocate two or more handles when the size of a data area exceeds 1020 bytes. One data area may then be physically composed of several
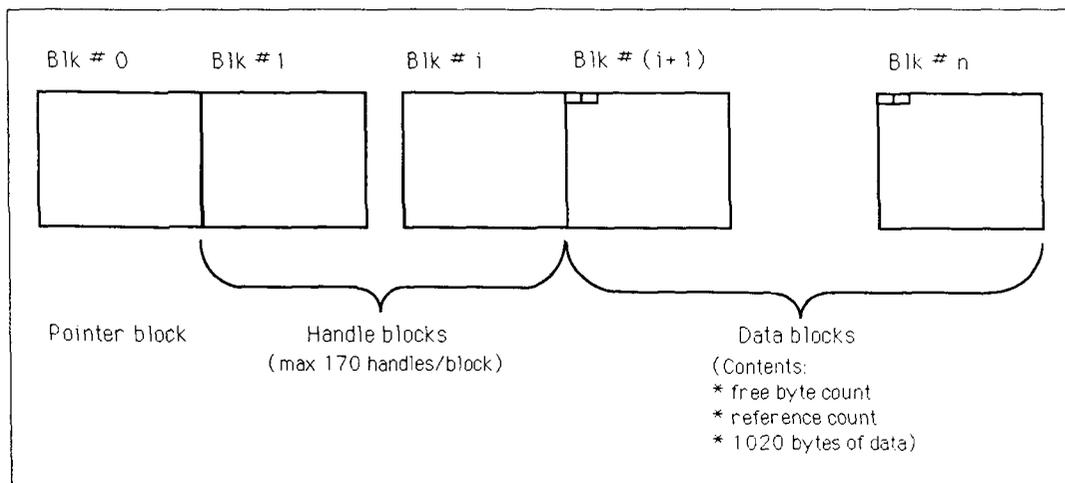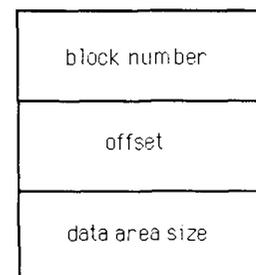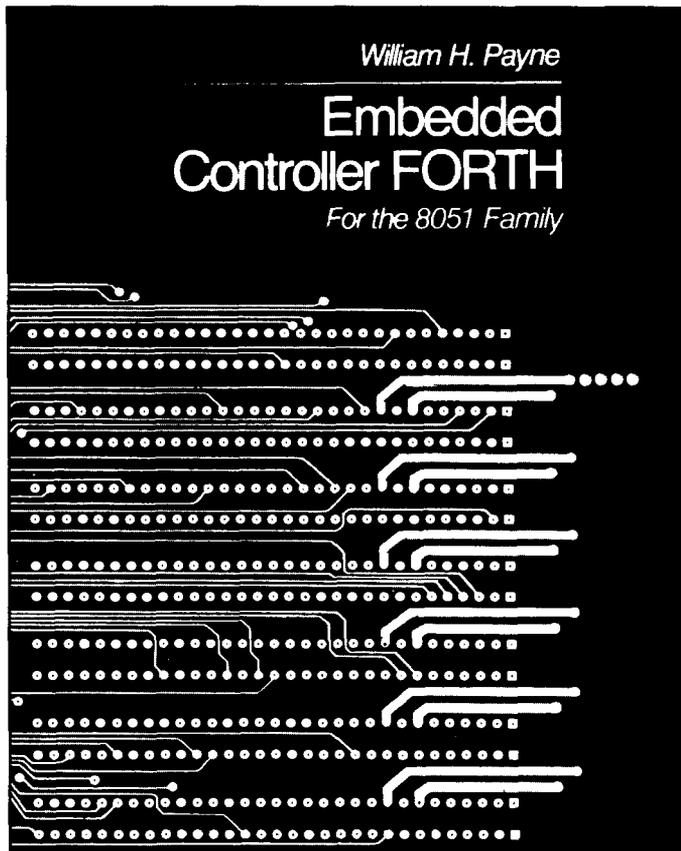

Figure One. Structure of a diskheap

Figure Two. Structure of a handle

separate data areas. This kind of mechanism has been implemented in the author's Kevo system, but since the mechanism is more application dependent, we will not discuss it here. When implementing such extensions, we must of course keep in mind that in Forth the virtual memory blocks will not necessarily be loaded to a contiguous memory area.

Example
   A new diskheap is created simply by saying

```
createheap test.hea
```

This creates a new disk heap called test.hea which is initially three disk blocks long. Data areas can now be allocated by using the command allocate. In this simple example, we shall at first create two Forth variables and allocate data areas of 20 and 40 bytes. The handle indexes of the data areas are stored to the variables.

```
variable data1
20 allocate data1 !

variable data2
40 allocate data2 !
```

The allocated data areas are initially erased to all zeros. To fill a data area with text we can use, for example, the standard Forth input word EXPECT which takes two parameters: the address where the string is to be stored and the maximum number of input characters. These parameters are provided by the extension words area and /area.

```
data1 @
area /area expect
update

data2 @
area /area expect
update flush
```

   To see the contents of the data areas we can use, for example, the Forth word TYPE:

```
data1 @
area /area type

data2 @
area /area type
```

The data areas were originally 20 and 40 bytes long. If we now want to change the first data area to be 50 bytes long and the second data area to be 30 bytes, we can simply use the word RESIZE:

```
50 data1 @ resize
30 data2 @ resize
```

Existing data areas can be deallocated by using the word FREE:

```
data1 @ free
data2 @ free
flush
```

Note that we must remember to use the word UPDATE to mark the buffer updated whenever the contents of an existing data area are changed; the words FLUSH or SAVE-BUFFERS must also be occasionally used. Before ending the use of a diskheap, FLUSH should always be executed.

An existing diskheap can be reopened with the command USEHEAP, e.g.:

```
useheap test.hea
```

About Coding

Our code should be standard Forth-83. Only a few F83-specific words are used. Such words are MORE which is used to request more disk blocks from DOS, and COPY which is used to copy a disk block to another. Other non-standard words are CREATE-FILE to create a new DOS file and OPEN to open existing files. Of course, the underlying Forth system must support block-oriented I/O.

In some definitions, the return stack is used heavily; the code could perhaps be made more readable if more variables were used. However, we have tried to minimize the number of potentially shared variables in order to keep the definitions re-entrant. This is to allow the extensions to be used in future multiprogramming environments. In the present form, however, multiprogramming is not supported.

Efficiency

For our purposes, the extensions have proven to be efficient enough. Accessing a data area requires at most two physical disk reads: reading the handle and the data block. Data area allocation, deallocation, or resizing, in turn, may take more time—

especially deallocation and resizing, if the disk heap is large. This is mainly because heap compaction may require going through all the handles in order to find the ones whose respective data areas are to be moved during the compaction; more intelligent algorithms for handle management might be worth investigating. However, the efficiency can be much improved by adding more resident disk buffers to the Forth system, thus decreasing the need for physical disk operations. A minimum of four buffers is needed, but when using very dynamically changing disk heaps the recommended number of buffers is at least eight. Since the extensions use the disk heavily, the speed of the disk device is also crucial. Although the extensions work correctly on floppy disk files, a hard disk—preferably a fast one—is recommended.

The main reason for implementing these virtual memory management extensions has been to create a persistent object storage kernel for the Kevo system; I have had neither time nor the interest to optimize this low-level code yet. I would, however, be grateful to receive any suggestions to improve the code.

Acknowledgements

Many thanks to Mike Elola and Michael Perry for reading and commenting on the previous version of this article at FORML'89. Without their encouragement I may not have sent this article for publication.

References

[ACC81] Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. "PS-Algol: An Algol with a persistent heap." *ACM SIGPLAN Notices* Vol. 17 no. 7, 1981, pp.24–31.

[Bee87] Beech, D. "Groundwork for an object database model." In Shriver, B., Wegner, P. (eds): *Research directions in object-oriented programming*, MIT Press, 1987, pp.317–354.

[Dre86] Dress, W.B. "A Forth implementation of a heap data structure." *Journal of Forth Application and Research* Vol 3. no. 3, 1986, pp.39–50.

[Har88] Harper, R. "Modules and persistence in standard ML." In Atkinson,

M.P., Buneman, P., Morrison, R. (eds): *Data Types and Persistence*, Springer-Verlag, 1988, pp.21–30.

[LiS88] Lindsjïrn, Y., Sjïberg, D. "Database concepts discussed in an object-oriented perspective." In Gjessing, S., Nygaard, K. (eds): *Proceedings of ECOOP'88 European Conference on Object-Oriented Programming* (Oslo, Norway), 1988, pp.300–318

[Low88] Low, C. "A shared, persistent object store." In Gjessing, S., Nygaard, K. (eds): *Proceedings of ECOOP'88 European Conference on Object-Oriented Programming* (Oslo, Norway), 1988, pp.390–408.

[MSO86] Meier, D., Stein, J., Otis, A., Purdy, A. "Development of an object-oriented DBMS." In Meyrowitz, N. (ed): *OOPSLA'86 Conference Proceedings* (Portland, Oregon), 1986, pp.472–482.

[Pou87] Pountain, D. *Object-oriented Forth: implementation of data structures*. Academic Press, 1987.

[Tai89] Taivalsaari, A. "Cool — unifying class and prototype inheritance." To appear in *Proceedings of FORML'89 Conference* on Forth and object-oriented programming (Pacific Grove, California), 1989.

[Tai89] Taivalsaari, A. "Implementing class inheritance without explicit classes." Submitted paper, 1990.

Antero Taivalsaari is a Ph.D. student of computer science at the University of Jyväskylä, Finland. He has been an avid Forth programmer for seven years, the last two of which have found him interested in object-oriented programming, the theme of his doctoral dissertation.

DYNAMIC MEMORY
ALLOCATION

DREAS NIELSEN - BELLEVIEW, WASHINGTON

Many programs require the ability to manipulate data elements of indeterminate size or number. Text strings are an example of one such type of data: each string may be a different length, and it is usually neither feasible nor economical to statically allocate (at compile or assembly time) a buffer capable of holding the largest possible string. Programs that manipulate arrays of numbers often need to establish their memory requirements dynamically— that is, at run time, without the use of a statically allocated buffer. Creation of linked lists, trees, and other more complex data structures also typically cannot be carried out with statically allocated memory. The solution to this problem is to provide the program with a means to dynamically allocate memory at run time. Dynamically allocated memory is drawn from the pool of main memory remaining after a program has been loaded and the stack (and other language-specific data structures) has been established. Dynamic memory allocation requires more complex run-time support than static buffers (which require none), but provides greater flexibility. If the available memory is to be re-used repeatedly for different purposes, the special-purpose code needed to manipulate a statically allocated block may be equivalent in size and complexity to that for general-purpose dynamic memory allocation.

Dynamic memory allocation is used by many common languages. In some of these, it is solely under the control of the language itself (e.g., strings in BASIC and dBase, all objects in Smalltalk and Lisp); in others, partial or complete control is given to the programmer (e.g., Pascal, C, and Modula-2). Explicit control of dynamic memory allocation is a powerful tool, and fundamental to effective implementation of many useful algorithms. Data structures such as linked lists, queues, and trees, for

example, are difficult to implement without it.

This article provides an introduction to the implementation of dynamic memory allocation, covering a few of the principles and providing examples for illustration. This is a topic that does not seem to be well covered in the literature; indeed, Knuth (1973) and Aho, Hopcroft, and Ullman (1983) seem to be the only commonly available references that treat it in depth. On the other hand, why would you want to know anything more about the implementation of such an arcane feature, particularly if you are not writing a compiler or operating system? First of all, you may need (or want) to use a language which does not intrinsically provide this feature but does have the systems-programming capability to implement it. Secondly, the memory allocation functions provided by a language or standard library may not exactly suit your needs. Curiosity, of course, may be sufficient reason in and of itself.

The heart of any memory allocation routine is a data structure.

Despite the relative paucity of information, dynamic memory allocation is not as complex as it may seem. Furthermore, an understanding of the techniques and costs involved can help you decide when a general-purpose routine is suitable, when a specialized routine may be better, and when to do without dynamic memory allocation. Source code for both a simple general-purpose routine and a more efficient specialized routine is provided for illustration. The implementation of dynamic strings is used as an example application of the general-purpose routine.

General Principles

A prerequisite for dynamic memory allocation is a pool of contiguous available memory from which smaller blocks can be allocated. This free memory space is generally called the heap; dynamically allocated subsets of it commonly are referred to as blocks. The physical location of the heap and the way in which it is isolated from other features of the run-time environment are dependent on the language in use and, often, its implementation. These details will not be considered here. (Dynamic allocation of space for local variables, which typically uses the stack rather than a heap, will also not be considered.) Other issues related to dynamic memory allocation, such as the identification of free blocks within the heap, the application interface, and efficiency considerations, are more general. The following discussion will address these topics. Note that although the principles described may apply, for instance, to BASIC's dynamic string handling, they will not necessarily allow you to add new dynamic memory functions to BASIC or some other languages.

The Free List

The heart of any memory allocation routine is a data structure that identifies the location of all free blocks of memory; this is conventionally called the "free list." Typically it takes the form of a singly linked list in which each node identifies the location of a block of available memory, the size of the block, and the position of the next node in the list. At first glance, allocation of storage space for the free list itself would seem to be a problem. Initially, all free memory would be in one block, requiring only one node—but after a series of allocations and de-allocations, the list may contain any number of nodes.

Where and how, then, is the free list stored?

One answer is to store the pointers to free memory in the free memory itself. This sounds a bit like a snake swallowing its own tail, but is actually quite simple and straightforward to implement. A small portion of each free block is used to store the block size and pointer to the next node, as shown in Figure One (page 23). The pointers to the free blocks are therefore implicit—the address of each node is itself the address of a free block. One consequence of this method of storage is that free blocks cannot be smaller than a node of the free list. In Figure One the nodes are shown sorted from low to high addresses. This arrangement makes deallocation easier, as shown below, but it is not the only scheme that can be used. Nodes may be sorted by block size, for example, to make allocation simpler.

Other methods may also be used to store the free list. The second example shown below uses a bit map, an approach made possible by the fact that blocks are of a fixed size and the total number of blocks is known.

Types of Dynamic Memory Allocation

There are several important distinctions among memory allocation systems. As mentioned previously, one of these is the issue of implicit versus explicit control—whether language features alone can make use of this resource or whether the programmer can use it too. Although implicit and explicit control of memory allocation are not generally found together, they are not mutually exclusive. The dynamic string package presented here, for example, automatically allocates and de-allocates memory to carry out its functions, but can coexist with user programs that make explicit use of the same functions.

When a language has sole access to memory allocation functions it can control all pointers to allocated space, and so need not replace each deallocated block back into the free list as soon as the application program releases it. This technique of delayed reclamation of de-allocated space ("garbage collection") allows programs to run faster, as long as there is sufficient free memory in the heap, at the expense of a relatively lengthy delay for garbage collection whenever the heap becomes exhausted. Lisp is an example of a language that manages memory using garbage col-

Listing One

```
Screen 1
0.   ( Dynamic Memory Allocation -- Screen 1 )
1.   ( Each block of free space begins with a 4-byte control block.
2.   The first word contains the address of the next free block
3.   [or 0 if none] and the second contains the number of bytes in
4.   the current block [including the control block]. )
5.
6.   ( Create pointer to beginning of free space, w/ size=0. )
7.   2VARIABLE FREELIST     0 0 FREELIST 2!
8.
9.   ( Initialize memory pool. )
10.  : DYNAMIC-MEM    ( start_addr  length --   )
11.         OVER DUP FREELIST !   ( Save starting addr. )
12.         0  SWAP !  ( Set null pointer. )
13.         SWAP 2+ !  ( Save length in 1st control block. )
14.       ;
15.


Screen  2
0.   ( Dynamic Memory Allocation, Screen 2: MALLOC )
1.   ( Returns pointer to n free bytes, or 0 if there is no space.
2.   Word before returned address holds size of block.  No free
3.   blocks of less than 4 bytes are allowed.               )
4.   : MALLOC  ( n -- n )
5.   2+ FREELIST  DUP
6.   BEGIN
7.   WHILE  DUP @  2+ @ ( Size )   2 PICK  U<
8.   IF @ @ DUP ( get new link )
9.   ELSE   DUP @ 2+ @ ( size )  2 PICK -  4 MAX  DUP 4 =
10.  IF     DROP  DUP @ DUP @ ROT !
11.          ELSE   2DUP  SWAP @ 2+ !    SWAP @ +
12.          THEN    2DUP ! 2+  0   ( store size, bump pointer, )
13.          THEN                 ( and set exit flag )
14.     REPEAT    SWAP DROP ( dump #bytes )   ;
15.


Screen  3
0.   ( Dynamic Memory screen 3:  FREE  )
1.   ( Deallocates memory.  Pointer passed must be from MALLOC )
2.   : FREE    ( ptr --  )
3.   2- DUP @  SWAP    2DUP 2+ !    FREELIST DUP
4.   BEGIN    DUP   3 PICK U<    AND
5.   WHILE    @ DUP @
6.   REPEAT               ( at exit: ( size block ptr1 )
7.   DUP @  DUP 3 PICK !  ?DUP  ( sz blk ptr1 0 -or- ptr2 ptr2 )
8.   IF  DUP 3 PICK 5 PICK +  =       ( size blk ptr1 ptr2 t/f )
9.    IF  DUP  2+ @  4 PICK +  3 PICK 2+ !  @ 2 PICK !
10.  ELSE DROP THEN                      ( sz blk ptr1 )
11.  THEN                               ( sz blk ptr1 )
12.     DUP 2+ @ OVER +  2 PICK =       ( sz blk ptr1 t/f )
13.     IF  OVER 2+ @  OVER 2+ DUP @  ROT +  SWAP !  SWAP @ SWAP !
14.     ELSE  !
15.     THEN  DROP  ;
```

Listing Two

```
Screen  1
0.   ( ASCIIZ string manipulation routines  )
1.   : TEXT ( c -- ) ( Parse text to matching char, put in PAD )
2.      >IN @ TIB @ + C@ OVER = IF DROP 0 PAD C! 1 >IN +! PAD
3.         ELSE WORD THEN COUNT DUP PAD + 0 SWAP C!  PAD SWAP CMOVE ;
4.
5.   : SCAN0 ( s -- z ) ( Returns address of terminating null. )
```

```
6.        BEGIN DUP C@ WHILE 1+ REPEAT   ;
7.
8.    : STRLEN  ( s -- n )  ( Return length of string in bytes )
9.        DUP SCAN0  SWAP  --   ;
10.
11.   : CHARS   ( n -- )   ( Define a string buffer of n chars. )
12.       CREATE 0 C, ALLOT DOES>  ;
13.
14.   : STRCPY  ( s1 s2 -- )  ( Copies from s1 to s2 )
15.       OVER STRLEN 2DUP + 0 SWAP C! CMOVE ;
Screen   2
0.    ( ASCIIZ string extensions   )
1.
2.    ( Return the address of a string literal compiled into
3.      the dictionary.  )
4.
5.    : (")     ( -- s )
6.       R> DUP  BEGIN DUP C@ WHILE 1+ REPEAT  1+  >R ;
7.
8.    : "       ( -- s )   ( Example: " This string." State-smart. )
9.       34 TEXT  PAD  STATE @ IF   COMPILE (")
10.                             DUP STRLEN 1+ HERE SWAP ALLOT STRCPY
11.                        THEN  ;    IMMEDIATE
12.
13.   : PRINT ( s -- )  ( Print the ASCIIZ string at the addr. )
14.       BEGIN DUP C@ DUP WHILE EMIT 1+ REPEAT 2DROP ;

Screen   3
0.    ( More char and ASCIIZ string extensions )
1.    : UCASE  ( c -- c )   ( Uppercases character. )
2.        DUP 96 >  OVER 123 <  AND  IF 223 AND THEN  ;
3.
4.    : CFROM ( a1 a2 -- a1 a2 c )  ( Gets char from pointer under.)
5.        OVER C@ ;
6.    : CFROM+ ( Like CFROM, but increments pointer )
7.        CFROM  ROT 1+ -ROT  ;
8.    : CTO    ( a1 a2 c -- a1 a2 )  ( Puts char at top pointer.)
9.        OVER C!  ;
10.   : CTO+   ( Like CTO, but increments pointer. )
11.       CTO 1+ ;
12.   : CTRANS+  ( a1 a2 -- a1+1 a2+1 )  ( Transfers a char. )
13.       CFROM+  CTO+ ;
14.
15.   : EOS?    ( a1 -- f )   C@  NOT  ;

Screen   4
0.    ( More character and ASCIIZ string extensions. )
1.    : C@C=   ( c addr -- f )    C@ =  ;
2.
3.    : STRPOS  ( c zstr -- n )    ( Returns position of c in zstr, )
4.        0 >R  BEGIN  2DUP C@C= NOT  ( 0-based, or -1 if not found. )
5.        OVER EOS? NOT AND  WHILE 1+ R> 1+ >R  REPEAT
6.        C@C= IF  R>  ELSE  R> DROP -1  THEN ;
7.
8.    : INSTR  ( c zstr -- f )   ( T if c in zstr, F otherwise )
9.        STRPOS -1 = NOT ;
10.
11.   : STRCAT  ( zstr1 zstr2 -- )   ( appends zstr1 to zstr2 )
12.       SCAN0  STRCPY  ;
13.
14.   : TOUPPER  ( zstr - )    BEGIN DUP EOS? NOT WHILE
15.       DUP C@ UCASE OVER C!  1+ REPEAT DROP  ;
```

lection. The technique is particularly appropriate for programs that require dynamically allocated memory but are expected to ordinarily require less than the total amount of memory available. This article describes only immediate reclamation of de-allocated memory; Knuth and Aho et al. should be consulted regarding strategies for garbage collection.

Another important distinction between memory allocation schemes is related to the need for fixed- or variable-size memory blocks. An application that creates and destroys only a single type of uniformly sized structure may use a different strategy than one which manipulates structures of many different sizes. Implementations satisfying these different needs may vary greatly in complexity and efficiency. Some of these differences are illustrated by the examples described below.

A third important factor is the sequence of allocation and de-allocation requests that will be generated by an application. If de-allocation proceeds in the inverse order of allocation (i.e., like a stack), specialized routines tailored for the purpose may be made much more efficient than general-purpose memory allocation functions. Other patterns of allocation and de-allocation requests can lead to varying fragmentation of the free list; memory allocation routines can also be optimized to cope with a high or low degree of fragmentation.

Application Interface

A simple example of dynamic memory use is a program which sorts or counts values in an input file by constructing a binary tree in memory as the file is read. A new node of the tree would be allocated every time a new item is found in the file. The sorted output can then be written to another file during an in-order traversal of the tree. A simple application such as this needs only to be able to allocate additional memory as needed. Any application much more complicated than this, however, will generally need to de-allocate memory as well. If the program described above is extended to read several files in succession, the tree should be de-allocated before the next file is read, to reduce the risk of running out of memory. This application is still simple enough, however, that performance can be improved by reclaiming the entire heap at once

rather than de-allocating the tree node-by-node.

Application programs generally make use of dynamic memory allocation, therefore, via two routines: one to allocate memory and one to release it. These routines are known to C programmers by the names "malloc" and "free" and to Pascal programmers as "new" and "release." Initialization of the dynamic memory buffer and routines is performed by the standard run-time code for these languages. If you write your own memory allocation routines, you will have to take care of this detail yourself, providing a third (initialization) interface to application software. The initialization routine is responsible for marking the entire contents of the heap as available; it may carry out other tasks also, depending upon the needs of the allocation and de-allocation routines.

Efficiency

The efficiency of dynamic memory allocation is principally a function of the time required to grab and release a chunk of memory. The amount of overhead space (i.e., the number of extra bits required for each allocated block) is also an efficiency consideration, but one that is likely to be less important than that of time. Factors that can affect the time required to allocate or free a block of memory are:

- Amount of free space available.
- Pattern of previous allocation and de-allocation requests; that is, the degree of fragmentation of the free space.
- Size of the block(s) to be allocated.
- Algorithms used.

Clearly, these all interact in ways that may differ from one application to another and even from one data set to another. If you are concerned about efficiency, your best approach is to evaluate the first three factors as best you can and use them to select appropriate algorithms. Generally applicable analyses of these interactions are probably not possible, although the individual factors may be examined (see Knuth, for example, for a discussion of the effect of memory fragmentation).

Choice of an appropriate algorithm can greatly affect the efficiency of an application. The two techniques presented here provide an illustrative contrast. The general-purpose routine requires two bytes of overhead per block, and the time required to allocate or de-allocate a block depends

Listing Three

```
Screen  1
0.  ( Dynamic strings, screen 1.  DYNAMEM package must be loaded.)
1.  : STRVAR   ( Create pointer to dynamic string. )
2.      CREATE  0 , ;      ( a VARIABLE by another name )
3.
4.  STRVAR __SYSSTR        ( Save ptr to created/modified strings. )
5.
6.  : LEN    ( dstr -- )    @ STRLEN ;
7.
8.  : RELEASE  ( dstr -- )  DUP @  ?DUP IF FREE THEN  0 SWAP ! ;
9.
10.  : STRSAVE  ( zstr dstr -- )      ( Assigns zstr to dstr )
11.      SWAP DUP STRLEN 1+ MALLOC  ( dstr zstr mem )
12.      SWAP OVER STRCPY   SWAP DUP RELEASE    !  ;
13.
14.  : S!  ( dstr1 dstr2 -- )  ( Stores 1 in 2, making a copy )
15.      SWAP @ SWAP STRSAVE  ;

Screen  2
0.  ( Dynamic strings, screen 2          )
1.
2.  : LEFT  ( dstr1 n -- dstr2 )   ( Returns left n chars of dstr1)
3.      OVER LEN OVER MIN 1+ MALLOC   DUP >R   ROT @ SWAP ROT
4.      ( zstr mem n -- )  2DUP + 0 SWAP !   CMOVE
5.      __SYSSTR RELEASE   R> __SYSSTR !   __SYSSTR   ;
6.
7.  : RIGHT  ( dstr1 n -- dstr2 ) ( Returns right n chars of dstr1)
8.      OVER LEN SWAP -- 0 MAX  SWAP @ + __SYSSTR STRSAVE __SYSSTR ;
9.
10.  : SUBSTR  ( dstr1 n1 n2 -- dstr2 )
11.       ( Substring of dstr1 starting at char n1, of length n2 )
12.      ROT @  ROT 1- OVER STRLEN MIN + __SYSSTR STRSAVE
13.      __SYSSTR SWAP LEFT   ;
Screen  3
0.  ( Dynamic strings, screen 3.   S+  SAY  UPPER  )
1.
2.  : S+  ( dstr1 dstr2 -- dstr3 )   ( Appends 2 to 1 )
3.      OVER LEN  OVER LEN + 1+ MALLOC  DUP >R   ROT @ OVER
4.      STRCPY   SWAP @ SWAP STRCAT __SYSSTR RELEASE R> __SYSSTR !
5.      __SYSSTR   ;
6.
7.  : SAY   ( dstr -- )
8.      @ PRINT ;
9.
10.  : UPPER    ( dstr1 -- dstr2 )  ( Makes an uppercased copy )
11.        __SYSSTR S!  __SYSSTR @ TOUPPER __SYSSTR   ;
Screen  4
0.  ( Dynamic strings, screen 4.       S"  )
1.
2.  STRVAR __SYSSTR2
3.
4.  : (S")              ( For pre-incrementing NEXTs )
5.      R> DUP  BEGIN DUP C@ WHILE 1+ REPEAT  1+ >R __SYSSTR2
6.      STRSAVE  __SYSSTR2   ;
7.
8.  : S"  ( -- dstr )     ( Accepts text from input stream )
9.                       ( into anonymous dynamic string. )
10.      34 TEXT  PAD     STATE @ IF   COMPILE (S")
11.          DUP STRLEN 1+ HERE SWAP ALLOT STRCPY
12.      ELSE
13.          __SYSSTR2 STRSAVE  __SYSSTR2
14.      THEN   ;       IMMEDIATE
15.
```

Listing Four

```
Screen  1
0.    ( Dynamic mem. alloc. for fixed node size, screen 1.        )
1.
2.    VARIABLE NODESIZE      ( Size of each node )
3.    VARIABLE NODEMAP       ( Pointer to bit map of nodes )
5.    VARIABLE NODEBUF       ( Pointer to memory buffer )
6.    VARIABLE SRCHPTR       ( Node # at which to start search for free)
7.
8.    : >MASK    ( -1<n<8 --  mask  )
9.        1+  DUP 2 > IF 1 SWAP 1- 0 DO 2* LOOP THEN ;
10.
11.   : NODE     ( n -- m a )   ( n=node #, m=mask, a=address )
12.       8 /MOD  NODEMAP @ +  SWAP >MASK  SWAP  ;
13.
14.
15.


Screen  2
0.    ( Dynamic mem. alloc. for fixed size nodes, screen 2.    )
1.
2.    : >BYTES  ( n -- n2 )   ( Converts bits to bytes. )
3.        8 /MOD SWAP 0= NOT ABS +  ;
4.                                              HEX
5.    : CLEARNODES    ( -- )
6.        #NODES @  >BYTES  0 DO  FF  NODEMAP @ I + C! LOOP
7.        0 SRCHPTR ! ;                        DECIMAL
8.
9.    : NODEBUFSIZ  ( n1 n2 n3 -- )    ( n1 = address of buffer )
10.       DUP NODESIZE !              ( n2 = size of buffer, b )
11.       1+ /  DUP #NODES !          ( n3 = size of node, b   )
12.       >BYTES    OVER + NODEBUF !
13.       NODEMAP !
14.       CLEARNODES    ;
15.


Screen  3
0.    ( Dynamic mem. alloc. for fixed size nodes, screen 3.  )
1.                                              HEX
2.    : GETNODE    ( -- a )    ( a = 0 if no space available )
3.        0 ( accumulator )  #NODES @  0 DO  I SRCHPTR @ +
4.        #NODES @ MOD  DUP    NODE C@ SWAP AND ( free? )
5.        IF    DUP 1+ #NODES @ MOD SRCHPTR !
6.              DUP  NODE  DUP C@  ROT  FF XOR  AND SWAP C!
7.              SWAP DROP  NODESIZE @ *  NODEBUF @ +   LEAVE
8.        ELSE  DROP
9.        THEN  LOOP  ;
10.
11.   : RELEASENODE    ( a -- )   ( a  as returned by GETNODE )
12.       NODEBUF @ -   NODESIZE @ /
13.       DUP SRCHPTR !
14.       NODE DUP C@  ROT OR  SWAP C!  ;
15.                                              DECIMAL
```

upon the pattern of previous requests. The specialized routine for fixed-size blocks requires only one bit of overhead per block (approximately), in many cases requires near-constant (and minimum) time to allocate a block, and constant time to de-allocate a block.

General-Purpose Memory Allocation

The most important feature of a general-purpose memory allocation scheme is the flexibility to satisfy an indeterminate number of requests for blocks of varying sizes. The most appropriate structure for maintaining the free list under these conditions is a linked list. Each node of the list identifies the position of a free block, its size, and the location of the next block in the list. Generally, this linked list is stored within the free space itself, as shown in Figure One. The address of each node therefore identifies the position of the associated free block, and this information need not be explicitly stored.

For the sake of efficiency during de-allocation, the free list is generally kept sorted in order of increasing addresses. By using a doubly linked list, it is possible to make de-allocation slightly more efficient yet (the typical de-allocation strategy is discussed below).

Because each allocated block may be of a different size, and because de-allocation routines are typically passed only the address of an allocated block, the size of each block must be stored when it is allocated. (Modula-2, however, requires the size of the block to be passed to the standard deallocation routine.) It seems that the extra space needed to store the size could be eliminated if the de-allocation routine were passed the size as well as the address but, as discussed below, in some cases more space is actually allocated than is requested, unknown to the calling routine. For this reason, it is important to store the amount of space actually allocated rather than that requested.

Fitting Strategies

When searching for a free block to satisfy an allocation request, the memory allocation routine can select either:

- the first free block that is large enough (first fit) or
- the block that is closest in size to that needed (best fit).

The first-fit strategy is generally regarded as superior, as the number of small blocks tends to proliferate when using the best-fit method. In addition, because it usually must examine more (often all) of the free list for each allocation request, the best-fit method is slower.

If allocation requests fall into a known pattern, however, you may find that the best-fit method, or some variant of it, is more memory efficient. For example, suppose that your application most often requests blocks of 30, 50, or 70 bytes. After some period of use, most of the free blocks

are likely to also be of these sizes. In such a case, your best strategy may be to choose the first free block of appropriate size, reducing the number of useless 20-byte (approximately) free blocks created.

Eliminating Small Blocks

Wasted space is created whenever a free block is created that is smaller than the application is likely to request. The existence of too-small free blocks slows down the memory allocation routines, as their nodes must be examined each time the free list is traversed. Although it is not always possible to prevent this waste of space, it is possible to eliminate its effect on performance. This is done by including the "extra" space with the allocated block that would otherwise have left the bytes behind. The actual size of the allocated block, including the extra bytes, must be recorded in its reference cell, and the troublesome node can then be eliminated.

An Example of General-Purpose Memory Allocation

An implementation of a general-purpose memory allocation scheme is shown in Listing One. Forth encourages the construction of application-specific languages of arbitrarily high level, yet is unsurpassed for the direct memory manipulation needed to implement system routines. In keeping with the Forth philosophy of providing simple tools to build custom applications, there are no standard Forth words for dynamic memory allocation. The examples in these listings are presented in the same spirit: although they are fully functional, they should be regarded as examples only. You should modify, improve, or replace them as appropriate to the needs of your own applications. Heed the dictum about not reinventing the wheel, but be advised to trade in your standard steel-belted radials for racing slicks when the competition gets hot.

The two principal interface words, MALLOC and FREE, are shown in screens two and three of Listing One. These routines have the same calling conventions, as well as the same names, as their C counterparts, so even if you know nothing but C, you should be able to make some sense of the Forth code. (Some of the more avid proponents of other languages would say that if you know nothing but C, you know nothing at all; that's a rather harsh judgment, but I would agree that users of languages of the

PL-1 family—C, Pascal, Modula-2, and Ada—could profitably broaden their horizons by learning something different: Forth, Lisp, Prolog, APL, and Smalltalk all embody unusual approaches to computing.) This code is written for a 16-bit Forth-83 Standard system.

The free list in this implementation is a singly linked list in which each node occupies four bytes. Each node contains a link to the next, followed by the size of the block in bytes. No free blocks smaller than four bytes are allowed. If satisfying a request from an available block would leave fewer than four bytes, the extra bytes are included in the block being allocated. Except for this limitation, there is no minimum size imposed on either the allocated or free blocks. Free blocks are selected by the first-fit strategy.

The word DYNAMIC-MEM, in screen one, is used to initialize the heap. It should be passed the starting address and size of the heap in bytes. The heap itself may either be compiled directly into the Forth dictionary or placed in free memory above the dictionary. (If you choose the latter course, take care to avoid conflicts with PAD, TIB, block buffers, and the parameter and return stacks.)

DYNAMIC-MEM creates a single node or control block at the beginning of the heap space, setting its size to that of the entire heap. The address of this first node is stored in the double variable FREELIST, which has the same format as a node but, having a fixed address, serves as the root, always pointing to the first real node in the free list. The size cell of FREELIST is always zero; it exists so that FREE does not have to treat the root node as a special case.

Each block of allocated memory is preceded by a cell containing the block's size. This information is needed to de-allocate the block. Each allocated block is therefore actually two bytes larger than its nominal size. This overhead cost should be considered if you wish to use the smallest possible heap, based upon your knowledge of the number and size of blocks needed.

The word MALLOC is used to reserve a block; it is passed the number of bytes desired and returns the address of an appropriately sized block, or zero if the request cannot be satisfied. The first thing this word does is increase the requested size by two bytes to allow for the size cell. A sequential search of the free list is then performed, which is terminated when a block of suffi-

Figure One. Linked list in the heap.

Free block
Allocated block
Block to be de-allocated
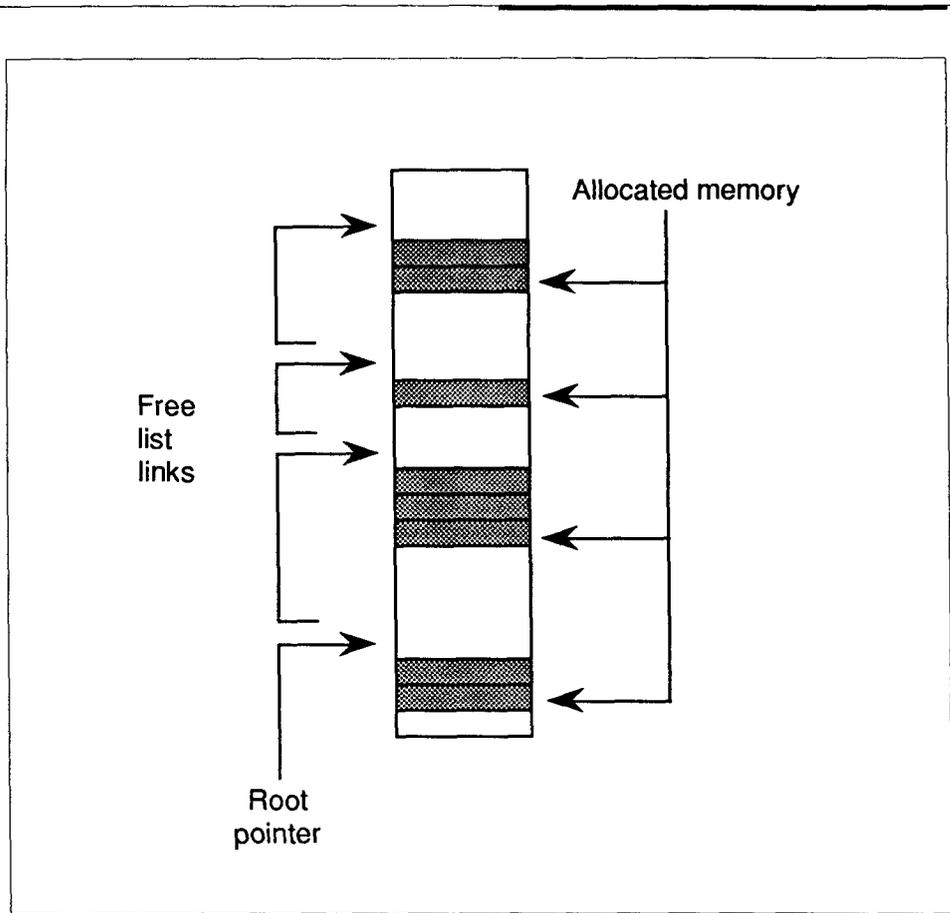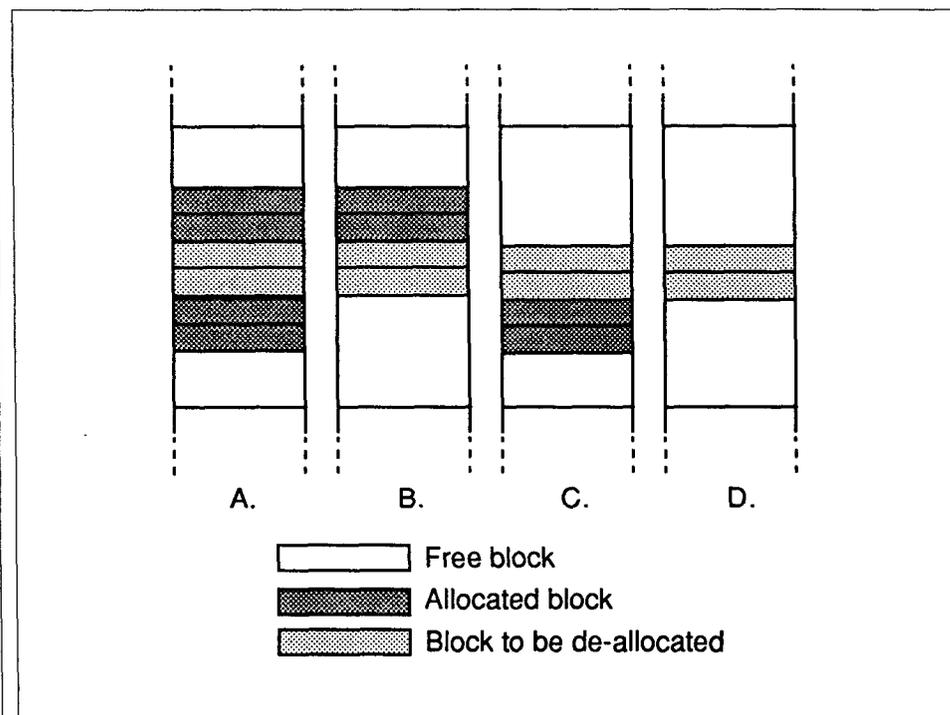
Figure Two. De-allocating a block; surrounding memory.

cient size is found or the end of the free list is reached. Either of these conditions is signaled by a zero on the stack; the test for this value occurs at the beginning of line seven. During this search, two values are kept on the stack: the number of bytes needed and the address of the node that contains the address of the node currently being examined. The address of the node "one back" must be maintained so that its node's link address can be adjusted in case the current node is entirely allocated and must be dropped from the free list.

Line seven of screen two fetches the size of the current block and tests it against the request. Line eight performs two fetches to get the link to the next block if the size is insufficient. Line nine evaluates whether the entire block should be allocated; if so, the pointers are adjusted in line ten, otherwise the size of the current block is reduced in line 11. In either case, the address of the block is left on the stack. Line 12 stores the size for later use, increments the pointer past the size cell, and sets a zero flag on the stack to terminate the loop.

Release of an allocated block may or may not result in the addition of another node to the free list. Blocks above and below the one to be de-allocated may themselves be either free or reserved. The four possibilities are shown in Figure Two. Only when the memory configuration is as shown in Figure Two-a will a new node be added to the free list. The situation shown in Figure Two-b will result in the creation of a new node within the newly de-allocated block, and the removal of the node above, for no net change. The link address previously pointing to the node to be removed must also be modified. When the situation is as shown in Figure Two-c, only the size of an existing node need be changed. If free memory bounds the de-allocated block on both sides, as in Figure Two-d, then the size of the lower node must be changed and the upper one eliminated.

The need to examine the blocks on both sides of the one to be de-allocated is why the free list is kept sorted by address. To find the address of the preceding free node, a sequential search is performed for a node which has an address lower than that of the one to be de-allocated, but a link address that is higher. If the size and address of the lower node sum to the address of the one to be de-allocated, then the situation in either

Figure Two-c or Two-d applies. To find the address of the following block (which will have a free-list node if empty), it is only necessary to sum the size and address of the block to be de-allocated; if the resulting address appears in the free list, then the situation in either Figure Two-b or Two-d applies.

Evaluation of the memory configuration and removal of the indicated node are performed by the word FREE in Listing One, screen three. This word begins by fetching the size of the node and storing it in the second cell, creating the size cell of a valid node header. A sequential search of the free list is then performed (lines 3–6), ending with the address of the free node below the one to be de-allocated. Note that this may be the root (FREELIST) which, because of the extra cell allocated to it, may be treated exactly like any other node header.

In line nine, the link address held by the next-lower free node is stored in the block to be de-allocated, completing the valid node header for this block. Nothing yet points to this header, and it may eventually be abandoned. Construction of the header at this step is more efficient, however, if the node is not to be abandoned. Lines eight through ten evaluate whether the node to be de-allocated is immediately followed by a free node; if so, the size cell of the newly created node header is increased by the size of the following free block and the link address is set to that contained in the following header. Lines 12–14 evaluate whether the block to be de-allocated is preceded by a free block; if so, the link and size cells of the preceding header are modified appropriately, and if not, the link address of the preceding header is set to that of the de-allocated block.

An Example Application

The use of these words is illustrated by a set of routines for manipulation of dynamic strings. Listing Two contains a set of static string-handling words, and Listing Three ties these together with the dynamic memory words in Listing One.

Strings are generally stored in memory in one of two ways: with the string length in the first byte or word, or with the end of the string marked with a sentinel character, usually an ASCII zero. For simplicity, I will refer to these alternatives as counted strings and zstrings. Dynamic strings will be referred to henceforth as dstrings. Forth

contains several standard words for manipulating counted strings (using a single byte for a count), but is not limited to this form of storage. I prefer to use zstrings, as they allow you to scan a string more easily; the remainder of the string can always be represented by a single stack element rather than by an address-count pair, as is necessary with counted strings. The words in Listing Two are therefore designed to create and manipulate zstrings rather than the more usual (for Forth) counted strings.

Because this is an illustration and not central to the point of this article, the words in Listing Two will not be described in detail. A few points are worth noting, however. In particular, the words TEXT and (") may be found in existing Forth systems with slightly different actions. Typically these create and return counted strings, whereas the versions shown here are designed for zstrings. If possible, you should rewrite SCAN0 in your native assembly language, as it may amount to only a single instruction. The words in Listing Two do not form a complete set of tools for handling static strings, but they include all those used to illustrate dynamic string handling in Listing Three as well as a few others.

The words in Listing Three integrate those in Listings One and Two. They allow

strings of any length (within the limits of the heap space) to be stored or modified without any concern on your part about overrunning a statically allocated string buffer. These words mimic some of the string-manipulation functions of dBase in name and application.

Dynamic strings are represented by a pointer to a zstring; the zstring itself is stored in the heap, rather than in the Forth dictionary. A dstring can be converted to a zstring simply by a fetch (@) operation. With that in mind, and an explanation of the role of __SYSSTR, the words in Listing Three should be easy to interpret.

Several of the dynamic string manipulation routines create new unnamed dstrings—that is, ones which do not directly replace one of the dstrings passed as a parameter. The words LEFT, RIGHT, and S+ are examples. This new, unnamed dstring is left on the stack, where you may save it (with S!), display it (with SAY), or otherwise dispose of it. The pointer to the heap space allocated for this string must not be lost, however, or the space will be unrecoverable. __SYSSTR is used to store this pointer. Note that the pointer is stored only until the next operation which creates a new unnamed dstring; at that point, the space is de-allocated and the pointer reassigned. In some situations, this limits the operations

that can be successively carried out on an unnamed dstring. Consider the following sequence of commands:

```
STRVAR COMPOST
" Gardeners rarely grow cabbage."
COMPOST STRSAVE
COMPOST 3 LEFT
COMPOST 5 RIGHT
S+
```

The result of this would be garbage, but not the "Garbage." that you might expect. Both of the phrases COMPOST 3 LEFT and COMPOST 5 RIGHT leave a pointer to an anonymous zstring, but only one anonymous pointer (__SYSSTR) is allowed. Thus, the two arguments passed to S+ will both be __SYSSTR, and the result will always be to concatenate the rightmost five-character substring of COMPOST with itself. The solution to this problem is to use another dstring defined with STRVAR for intermediate storage of the leftmost substring.

Any number of successive operations on a single, unnamed dstring may be carried out, however. For example:

```
COMPOST 6 LEFT UPPER SAY
```

These routines are written so that __SYSSTR may be one of their arguments, and space for the resulting string will be allocated before __SYSSTR is de-allocated.

Another way of reducing conflicts between uses of __SYSSTR is to use a different system string for each routine. This approach is taken with the word S" (the dstring counterpart to "), simply to allow the convenience of entering a string while an unnamed dstring resides on the stack. The drawback is that heap space may remain allocated long after the unnamed dstring is no longer needed by the application.

The technique of implementing dynamic strings shown in Listing Three is only an example. Counted strings could be used instead of zstrings. The count could also be kept in the dstring header, whether counted strings or zstrings are used. This last approach may be most suitable when you want to use zstrings for most purposes but your application frequently needs to evaluate the length of strings; the extra space devoted to storage of the string size,

although unnecessary, may save computation time. Tailor the tools to the task.

Special-Purpose Memory Allocation

If there is anything systematic about the size of blocks that will be needed, the number of allocation requests, or the pattern of allocation and de-allocation, you may be able to improve performance and save memory by using a special-purpose memory allocation routine. Whereas most general-purpose memory allocation routines will probably be based on a model somewhat like that presented above, you are pretty much on your own when it comes to designing a special-purpose routine. Knuth and Aho, Hopcroft, and Ullman describe a technique known as the "buddy system," which is a sort of general-purpose special-purpose system, suitable when only a limited number of sizes of blocks will be needed. Its advantage is that it can be customized for different combinations of sizes of blocks.

Considerations of fitting strategies and the problems of small blocks do not pertain when all blocks are the same size. It is, in fact, easier to design an appropriate solution for a single special-purpose application than it is to design a good general-purpose memory allocation routine.

The technique described here is one that is suitable only when blocks of a single size will be needed. But for this limitation, it has a number of advantages over the general-purpose routine described above:

- The time required to allocate a node is likely to be much less.
- The time required to de-allocate a node is constant.
- The overhead is only one bit per block, rather than two bytes.

These advantages are conferred by the representation of the free list as a bit map rather than as an actual linked list. The bit map consists of a series of bytes long enough so that their total number of bits is at least as great as the number of nodes that can be accommodated by the heap. The state of each bit (set or reset) indicates the availability of a corresponding node. The code for this implementation is shown in Listing Four. The words NODEBUFSIZ, GETNODE, and RELEASENODE are analogous to DYNAMIC-MEM, MALLOC, and FREE in Listing One.

A free block is indicated by a set (1) bit

in the map. To allocate a block, it is necessary to scan the map for such a bit and calculate the address to which it corresponds. To increase efficiency when a sequence of successive allocation requests may be performed, each search of the map begins where the previous one left off. To increase efficiency when an alternating sequence of allocation and de-allocation requests is performed, a pointer is set whenever a block is de-allocated so that the next search will begin with that block and so will be satisfied immediately. In some cases, only one of these fine-tuning mechanisms may be appropriate; both are shown here for illustration.

The housekeeping information is kept in the five variables shown in Listing Four, screen one. The first three words (>MASK, NODE, and >BYTES) manage the conversion between the bit map and actual addresses. The word >MASK ("to-mask") takes a bit number and converts it into a mask that can be used to test or set the bit with AND or OR. This is a good candidate for coding in assembly language.

Initialization of the bit map and housekeeping information is performed by the word NODEBUFSIZ. The beginning of the memory buffer is set aside for the bit map; this routine calculates the number of nodes that will fit and the size of the map needed. The map always occupies an integral number of bytes. Depending upon the buffer and node sizes, up to seven bits of the last byte of the map may be unused. The overhead per block may therefore be slightly more than one bit.

The word CLEARNODES has been factored out of NODEBUFSIZ so that it may be used to re-initialize the buffer without the need to use RELEASENODE to de-allocate each block. This word must be used with great care, and subsequent reference to dangling pointers should be avoided.

GETNODE (Listing Four, screen three) allocates space by looping over the total number of nodes; the phrase
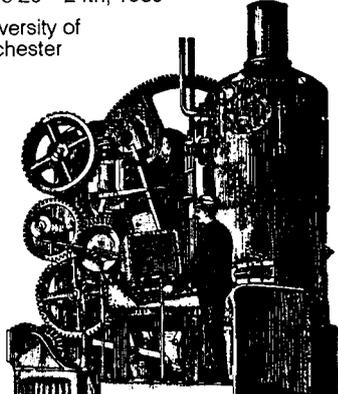
```
I SRCHPTR @ +
#NODES @ MOD
```

translates a relative node number into an actual node number based upon a non-zero starting position. If a free block is found, the starting point for the next search is set (line five), that entry in the map is marked as allocated (line six), and the actual ad-

PROCEED!

dress of the block calculated (line seven).

The word RELEASENODE de-allocates space by calculating the node number (screen three, line 12) and setting the appropriate bit (line 14). In addition, it sets the starting point of the next search to the node just de-allocated (line 13).

Because of the uniform block size, this approach lends itself to compressed displays of the allocation map more easily than does the first. A simple word to display this map may be defined as follows:

```
: SHOWMAP
  #NODES @ 0 DO
  I NODE C@ AND IF
    ." 1"
    ELSE
    ." -"
    THEN
  LOOP ;
```

Summary

The examples shown in this article, although useful in their own right, are intended principally to illustrate a point. That is: you can improve the performance of your application programs by tailoring memory allocation routines to their specific needs.

Several changes could be made in the general-purpose routine which might improve its suitability for certain applications. For example, each search could be started wherever the previous one terminated, as is done with the specialized routine. Also, backward links in each node header would eliminate the need for a sequential search when a block is to be de-allocated.

If the size of blocks is known at compile time (which is very often the case), the special-purpose routine could be improved by making NODESIZE a constant rather than a variable. Depending upon the actual block size (e.g., for powers of two), other changes may also increase performance. See the references.

Other special cases of memory allocation, such as a series of LIFO requests, may be handled by techniques very different from either of the examples shown here.

Although the standard libraries of most conventional languages provide routines only for general-purpose memory allocation, you can still take advantage of opportunities to create special-purpose routines as needed. If you cannot supplant the standard routines, they can at least be used to

permanently allocate a heap large enough for your own routines. For some applications, you may even wish to have two or more different memory allocation techniques in use simultaneously, each with its own heap. Consider the needs of your application carefully, use the techniques shown here and in the references as guides, and you can design memory allocation routines that provide optimum performance.
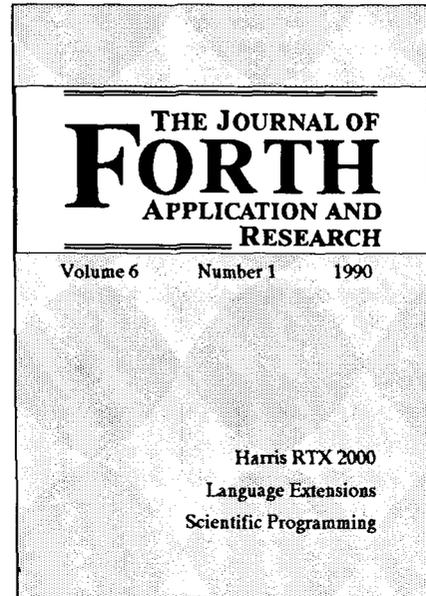
References

Aho, A.V., J.E. Hopcroft, and J.D. Ullman. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, Mass. 427 pp.

Knuth, Donald E. 1973. *The Art of Computer Programming*. Vol. 1, Fundamental Algorithms. Second Edition. Addison-Wesley, Reading, Mass. 634 pp.

SMART
RAM

ROB CHAPMAN - EDMONTON, ALBERTA, CANADA

I first heard of smart RAM from Bob La Quey. It was one of those amusing things to throw out for discussion when you sat down to chew the fat. We tossed the idea around a lot but never had an application for it. Until now.

Breeding Forths

After spending six months creating and tuning a real-time kernel for the RTX-2000, I wanted to port it to other processors. Since we had a lot of 68000s kicking around at work, I decided that the first port would be to a familiar processor: an 8 MHz 68000. The traditional, well-known method of building a 68000 system of ROM, RAM, SIO, and a processor with the burn-EPROM-plug-it-in-doesn't-work-edit-swear-try-again method didn't really appeal to me, so I figured there must be a better way. A bolt of lightning struck and I thought, maybe this is an opportunity to use smart RAM. That would allow me to interactively and incrementally test the Forth, monitor the performance of each word, and tune it for the 68000.

68000 and Smart RAM

Being a Forth-bred minimalist, I didn't want to build a lot of hardware (or software) to achieve a smart RAM system. The resulting hardware configuration consists of a 68000, interface logic, a bot, and a laptop.

The laptop connected to the bot provides file storage, an editor, and a terminal.

The bot (bundle of technology) is a concept that, in itself, is worth another paper. For the scope of this paper, though, a bot is a minimal RTX-2000 system, which consists of an RTX-2000, RAM, SIO, 96-pin expansion connector, and a Forth. This, bundled with some software, emulates smart RAM.

The interface logic maps the 68000 address bus, data bus, and the control signals onto the RTX-2000 gio bus, where a data exchange protocol takes care of moving data between the 68000 and RAM.

Developing a smart RAM language required a lot of fiddling (as is usual, and very much a part of Forth), but it eventually broke down into primitives to deal with the 68000 signals; words to transfer data between RAM and the 68000; and a debugging language which would allow for single-stepping, peeping and poking into the 68000 registers, and monitoring the 68000 transactions.

Smart RAM concepts can be applied in many areas...

Data Exchange Protocol

All data exchanges are initiated by the 68000 asserting address strobe low. This qualifies all the signals. All data exchanges are then terminated by the smart RAM asserting dtack low.

There are three types of exchanges: instruction-read, data-read, and data-write. If r/-w is high and p/-d is high, the 68000 is doing an instruction read. If p/-d is low, it is doing a data read. If r/-w and p/-d are low, it is doing a data write.

During a data write, a check is performed to make sure it falls within the area reserved for the 68000 Forth. During an instruction read, the instruction doesn't have to come from RAM, it can come from the stack. Feeding the 68000 instructions from the stack allows instruction sequences to be inserted, and if followed by a jump back to the original address these sequences are essentially transparent. This technique is used for unobtrusive register peeps and pokes.

As well, data reads and writes may be ignored by just asserting dtack. Instruction reads may be skipped by writing a no-op to the data bus and asserting dtack.

Debugging Tools

The debugging tools consist of a transaction monitor with single-stepping and a register content editor.

The transaction monitor displays each transaction between the 68000 and smart RAM. The status is displayed before each dtack. Filters allow selective viewing of instruction reads, data reads/writes, or both. There are also filters for disassembling and decompiling. If the disassembler is enabled, each 68000 instruction is displayed.

If the decompiler is enabled, the high-level Forth words being executed are displayed. The monitor can be adjusted to show any depth of nesting. After a word is tested, a summary of the transaction and the status of the Forth registers is displayed. This summary and status may also be turned off. The monitor output may be stopped by touching any key. An escape will then abort the execution of the current word, while any other key will resume status display.

When single-stepping is enabled, either by an out-of-bounds read/write or by user selection (SSTEP), execution is stopped just before each dtack. Execution may be aborted by pressing the escape key, or continued by pressing any other key.

The register content editor serves two purposes. The seven registers used by the virtual Forth machine may be viewed, and any of the registers may be altered. This basic peeking and poking is accomplished by feeding instruction sequences to the 68000 followed by a jump back to the address it started from.

68000 Forth Model

I chose a Forth model for the 68000 which is similar to the RTX-2000. Technically, it is a 16-bit, subroutine-threaded, stack-cached Forth.

The top two parameter stack items are cached on chip in registers, just like the RTX-2000. The top of stack is an address register to allow for very quick fetches and stores. The next stack item is kept in a data register for quick ALU operations.

Since subroutine threading is used, it allows the mixing of assembler and Forth within the same definition (the mini-assembler was written to support this by making the assembler words immediate). Subroutine threading also allows for in-line optimization of assembler instructions.

One interesting dilemma resulted from using subroutine threading and 16-bit stacks: every subroutine call pushes a 32-bit address onto the return stack and, likewise, every exit pulls a 32-bit address from the return stack. Since Forth allows for modifying control flow via direct access to the return stack (R>, R, and R>), there was an immediate compatibility problem. This is solved by allowing the return stack to be 32 bits, while keeping the parameter stack as 16 bits. R>, R, and R> take care of the translation.

The model uses five other registers, as well as the two top parameter stack items. Two address registers are used for the stack pointers. A data register is used for intermediate results. An address register is used for some pointer operations. The FOR ... NEXT loop uses a data register to hold the index. The previous contents of the index register is pushed onto the return stack by FOR and is restored by NEXT. This model, coupled with a good peephole optimizer to minimize off-chip stack flow, should run fast. All the stack primitives are one or two opcodes (except ROT), which means that it is cheaper to in-line them than to do a subroutine call.

Interactive Development

I started with the Forth I wrote for the RTX-2000, or 2K-Forth, and defined everything in high level except about 20 primitives. These include stack and math/logic operators, memory access, and a test operation:

```
SWAP    DROP
DUP     NIP
>R      R>
```
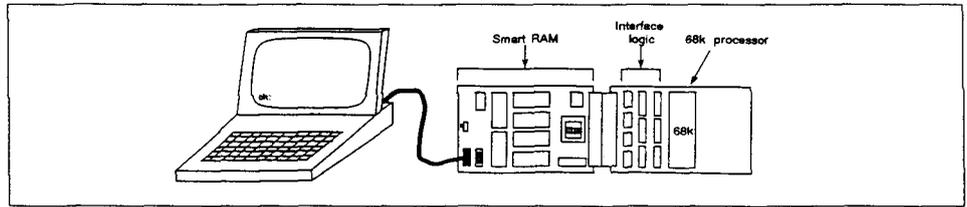

Figure One. The smart RAM system consists of a laptop, an RTX-2000 bot, and a prototype board. The laptop serves as a terminal, an editor, and provides file storage. The RTX-2000 bot emulates smart RAM and runs Forth. The prototype board contains the 68000 and the interface logic which multiplexes the 68000 signals onto the RTX-2000 16-bit gio bus.
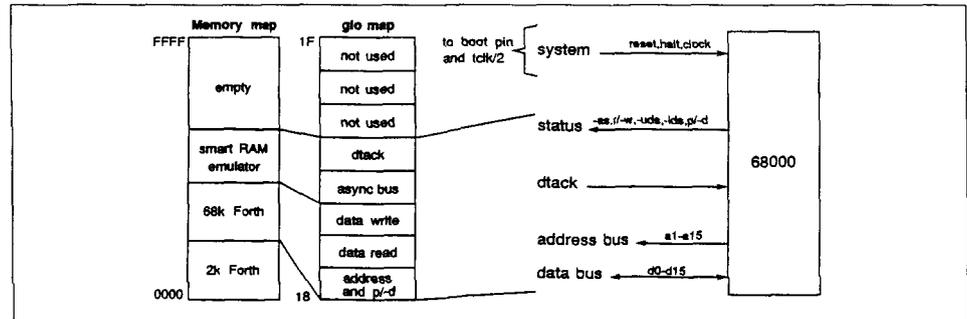

Figure Two. All the 68000 signals are mapped onto the RTX-2000 gio data bus. Only the 15 least significant address lines are used. This, with -uds and -lds, gives an effective address range of 64K bytes. dtack is the output of a flip-flop which is set by the address strobe going low, and is reset through the gio bus. Address, data-read, data-write, control signals, and dtack map to five out of the possible eight addresses on the gio bus. The signal p/-d (actually FC1) allows differentiation between opcode and data fetches. The boot pin is used to control reset and halt. The clock input is fed tclk divided by two, so that the 68000 receives a 5 MHz clock.
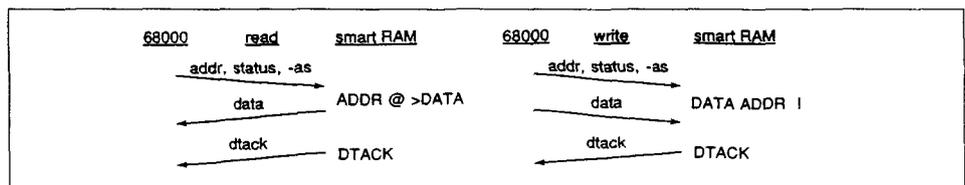

Figure Three. Exchanges between the 68000 and smart RAM are initiated by the 68000 asserting address strobe and terminated by smart RAM asserting dtack.
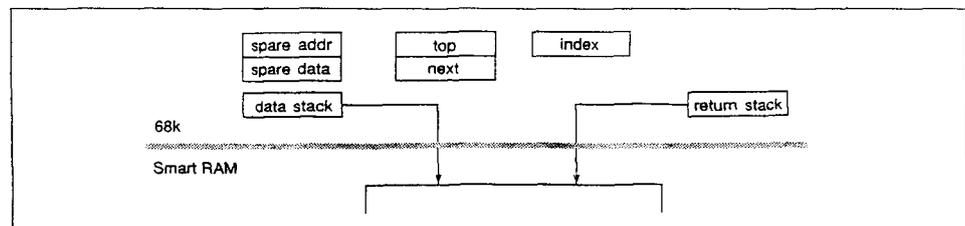
Figure Four. The 68000 Forth uses seven out of 16 registers. The top two data stack items (top and next) are kept in registers to quicken most Forth operations. Two pointers are needed to keep track of the stacks in memory. The index register is only used for FOR ... NEXT loops. When it is used, the previous contents are pushed onto the return stack. Two extra registers (spare data and spare addr) are needed for some intermediate calculations.

R   +
-      AND
OR   XOR
NOT   2*
2/   @
C@   !
C!   0=

The modified 2K-Forth was tested by meta-compiling it on the bot.

Once I had pared the Forth down to a few primitives, it was very simple to redefine them for the 68000. Adjustments had to be made to the metacompiler to compile the 68000 subroutine calls and returns. Headers, byte ordering, and word alignment stayed the same. The metacompiler kept track of which definitions consisted of fewer than four words, so that they could be in-lined (this mega-simplified the return stack definitions). Since there were very few primitives, a mini-assembler was built (they could have been hand coded, but the assembler provided a creative outlet).

Development proceeded rapidly by executing each primitive on the 68000, monitoring bus activity, and checking the Forth registers when done. Operations like SWAP took one opcode and no memory access.

Once I had the twenty primitives working, I started testing the other definitions. Since I was monitoring bus activity, I could see just how long it took to execute the words. This motivated me to code some of these words in assembler. When I needed more 68000 instructions, I added them to my assembler and disassembler as well.

The development actually was done in parallel. When a problem occurred in the 68000 Forth, I created a new tool to solve it on the smart RAM. Sometimes this made matters worse, since one bug can sometimes hide another. But perseverance paid off, since I now have most of the tools I need.

Finally came the steak dinner test: does it work? To test the whole kernel, I connected the input and output queues from the 68000 Forth to the I/O port of the bot. The 2K-Forth serviced the I/O port and took care of RAM transactions. The euphoria came when it all worked.

Prospects

The next few processors to be connected to the smart RAM are the 68020, a microcontroller from National Semiconductor and, possibly, a popcorn part like the

Figure Five. This is a listing of four trace monitor outputs of ROT with different filters selected. All input has been highlighted [and follows the ok: prompts]. Initially, the status of the Forth registers is shown by the command S. ROT is then run with all output filtered. The summary that comes back displays how deeply nested the calls got (in this case zero, since ROT is just four instructions); how many subroutine calls were made; how many opcodes were fetched; how many stores to RAM; how many data fetches from RAM; and, finally, how many dtacks (one for each memory cycle). The third run has disassembling and decompiling enabled, for a maximum trace depth of one. The code is displayed first and the Forth code is displayed after it. The decompiler considers the four opcode instructions to be ROT. In the fourth run, RAM accesses are traced as well, and we can see the fetch from stack and the store back to stack (the fetch and store don't happen right away, because of the internal architecture of the 68000). The display on the right is the actual signals that exist on the 68000 busses before a dtack. Non-zero values represent a logical one, while zero represents a logical zero.

Figure Six. RESET-INPUT is decompiled with TSEE and then the first five Forth words are traced. The output has been temporarily halted to view the results. In this trace, we are only interested in the highest-level calls.

6811 from Motorola.

The concept of smart RAM can be applied in many other areas as well. It could be used as a way to speed up slow RAM in a system, with the addition of some fast latches (one for the data stack, one for the return stack, and one for instructions). The smart RAM would prefetch the next needed value from RAM before the processor actually requests it, and store it in the latch.

When the processor requests the data, it is already in the latch. The stacks are easy to predict, but programs would be harder. But by knowing what the branch or jump instructions are ahead of time, the smart RAM could make an intelligent choice for what to put into the latch.

To take things to an extreme, the smart RAM could intercept slow instructions or

TESTING
TOOLKIT

PHIL KOOPMAN, JR. - WEXFORD, PENNSYLVANIA

O ne of Forth's strong points is its support of interactive development and testing. Sometimes, however, interactive testing is not enough. During the development of low-level software for the RTX family, we wanted a method to create a permanent record of test cases for Forth words. This record serves as documentation for users and maintainers. In addition, a full suite of test cases for a program provides a way to be sure that a change in one part of the program does not disturb other parts of the program.

How to Use It

Each test case consists of code that places elements on the data and return stacks, creates and executes a test definition, then verifies that the correct results were placed on both stacks. For example, a test case for the word DUP would be:

The test case can be any sequence of Forth words.

```
DS( 1111 --
RS( --
TEST: DUP ;DONE
-- )RS
-- 1111 1111 )DS
```

The first line of the test case specifies that the data stack input to the test is the number 1111. The second line specifies that no elements are to be placed onto the return stack. The third line creates and executes a temporary Forth word with a body of DUP, carefully handling the data and return stack contents before and after the test. The fourth line specifies that no values should

```
\ Forth testing support
\ By Philip Koopman Jr., for Harris Semiconductor
\    Derived from test code used for the RTX chip family
\    Developed on F-TZ (an F-PC and F-83 derivative) version 3.X11

VARIABLE #STACK -1 #STACK !    \ Saves number of stack elements for testing
CREATE R-SAVE 8 ALLOT          \ Note: F-TZ uses 32-bit return addresses!

: GET-DEPTH   ( ..stack.stuff.. - ..stack.stuff.. )
  DEPTH #STACK @ --   #STACK ! ;

: DS(     ( -- $BAD1 $BAD2 )
  \ Init RS to -1 so that '-' will know it is a DS input
  \ Uses hex 0BAD1 and hex 0BAD2 as sentinel values for DS
  -1 #STACK !       $BAD1 $BAD2 ;

: RS(     ( -- $BAD3 $BAD4 )
  \ Uses hex 0BAD3 and hex 0BAD4 as sentinel values for RS
  DEPTH #STACK !   $BAD3 $BAD4 ;

: --      ( n1 n2 n3 .. n.n - n1 n2 n3 .. n.n sentinel )
  #STACK @  0<  NOT    IF ( if RS( )  GET-DEPTH   THEN ;

: ?DATA     ( n1 n2 -- )
  = NOT   ABORT" DATA STACK ERROR" ;

: ?RETURN   ( n1 n2 -- )
  = NOT   ABORT" RETURN STACK ERROR" ;

: --        ( --)
  DEPTH  #STACK ! ;

: PERCOLATE   ( r1 n.n .. n1 --  n.n .. n1 r1 )
  #STACK @  ROLL  -1 #STACK +! ;

: )RS        ( r.n .. r3 r2 r.1  n1 n2 n3 .. n.n -- )
  GET-DEPTH   #STACK @
  IF  BEGIN  PERCOLATE  ?RETURN   #STACK @ 0= UNTIL   THEN
  $BAD4 ?RETURN   $BAD3 ?RETURN   -1 #STACK ! ;

: )DS        ( r.n .. r3 r2 r.1  n1 n2 n3 .. n.n -- )
  GET-DEPTH   #STACK @
  IF  BEGIN  PERCOLATE  ?DATA     #STACK @ 0= UNTIL   THEN
  $BAD2 ?DATA     $BAD1 ?DATA     -1 #STACK ! ;

: REVERSE   ( n.1 n.2 .. n.n  n  -- n.n .. n.2 n.1 )
  DUP 0> IF  0 DO   I ROLL   LOOP  ELSE DROP THEN ;

: INIT-TEST   ( ..DS.stuff..  ..RS.stuff..  -- ..DS.stuff.. )
              ( RS:    -- ..RS.stuff.. )
```

be left on the return stack, and generates an error message if this is not the case. The fifth line specifies that two values of the number 1111 should be returned from the test, again generating an error message if this is not the case. It is very important that the test cases be written in exactly this order, with no missing items, for proper operation.

The body of the test case between TEST: and ;DONE can be any sequence of Forth words, including primitives that manipulate the return stack. The words INIT-TEST and FINISH-TEST are automatically compiled with the test case to handle the data and return stacks for proper execution.

In order to be sure that a word is working properly, it is not enough to simply place the required number of parameters on the stack and then see if the correct results are returned. The problem is that a word may cause unexpected side effects (such as corruption of elements on the data and return stacks) that are not detected immediately. In order to handle this case, the test words place two "sentinel values" onto both the data stack and the return stack, then check to ensure that no corruption has occurred. While side effects are usually not a problem in high-level code, they can easily create problems when dealing with assembly language or microcode word implementations.

Ideas for Further Refinements

The test capability presented here is rather simple, in order to keep the code (somewhat) understandable. Features that could be added to improve its usability include: allowing RS ( ) RS to be optional, so tests that deal only with data stack operations could automatically generate and test return stack sentinel values; more sophisticated error messages that show exactly what is wrong with a stack when an error does occur; methods to ensure that only desired memory locations are modified for words that perform fetches and stores; and methods to ensure that only desired on-chip registers are modified for assembly language definitions.

The code is written for F-TZ, a version of F-PC, developed by Tom Zimmer. F-PC is a descendent of F-83, but allows using a dictionary space of greater than 64K bytes. The code presented should be relatively

```
  CR ." TEST-"
  #STACK @ 0< ABORT" You must specify both DS( and RS(."
  R> R-SAVE ! R> R-SAVE 2+ !          \ Save return address
  #STACK @ REVERSE
  BEGIN #STACK @ 0> WHILE >R -1 #STACK +! REPEAT
  R-SAVE 2+ @ >R R-SAVE @ >R  ;    \ Restore return address

: FINISH-TEST ( ..DS.stuff.. -- ..DS.stuff.. ..reversed.RS.stuff.. )
              ( RS: ..RS.stuff.. -- )
  R> R-SAVE ! R> R-SAVE 2+ !          \ Save return address
  \ Transfer return stack contents onto data stack for later compare
  0 >R
  BEGIN R> R> SWAP 1+ >R  DUP $BAD3 = UNTIL
  R> REVERSE
  R-SAVE 2+ @ >R R-SAVE @ >R          \ Restore return address

  ." -DONE" -1 #STACK ! ;

\ TEST and DONE use F-TZ specific words to compile a short
\   definition containing the word to be tested, execute that
\   definition, then FORGET it from the dictionary.
\ This borrows a compilation idea from Rick van Norman's RTX test code
CREATE MARKER  4 ALLOT
: TESTER ;
: TEST:    ( --)
   XHERE 2DUP MARKER 2! PARAGRAPH + DUP XDPSEG ! 0 XDP !
   XSEG @ -- ['] TESTER >BODY !
   COMPILE INIT-TEST ] ;


: ;DONE
   COMPILE FINISH-TEST COMPILE EXIT
   STATE OFF TESTER MARKER 2@ XDP ! XDPSEG ! ;
IMMEDIATE

\ Test ROT for proper operation
  DS( 1111 2222 3333 --
  RS( --
  TEST: ROT ;DONE
  -- )RS
  -- 2222 3333 1111 )DS


\ Test >R for proper operation
  DS( 5555 --
  RS( --
  TEST: >R ;DONE
  -- 5555 )RS
  -- )DS


\ Any combination may go between TEST: and ;DONE
  DS( 1111 2222 3333 --
  RS( 7777 2222 9999 --
  TEST: SWAP R> ROT >R ;DONE
  -- 7777 2222 3333 )RS
  -- 1111 2222 9999 )DS


\ Null test to be sure it works
  DS( --
  RS( --
  TEST: ;DONE
  -- )RS
  -- )DS
```

Twelfth Annual

FORML CONFERENCE

The original technical conference
for professional Forth programmers, managers, vendors, and users.

Following Thanksgiving, November 23–25, 1990

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

Conference Theme: Forth in Industry

Papers are invited that address relevant issues in the development and use of Forth in industry. Papers about other Forth topics are also welcome.

Mail abstract(s) of approximately 100 words by October 1, 1990 to FORML, P.O. Box 8231, San Jose, CA 95155.

Completed papers are due November 1, 1990.

Conference Registration

Registration fee for conference attendees includes conference registration, coffee breaks, and note-book of papers submitted, and for everyone rooms Friday and Saturday, all meals including lunch Friday through lunch Sunday, wine and cheese parties Friday and Saturday nights, and use of Asilomar facilities.

Conference attendee in double room—$285 • Non-conference guest in same room—$160 • Children under 17 in same room—$120 • Infants under 2 years old in same room—free • Conference attendee in single room—$360

Register by calling the Forth Interest Group business office at (408) 277-0668 or writing to: **FORML Conference, Forth Interest Group, P.O. Box 8231, San Jose, CA 95155.**

About Asilomar

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations. It is situated on the tip of the Monterey Peninsula overlooking the Pacific Ocean. Asilomar is part of the California State Park system; it occupies 105 secluded acres of forest and dune. If you like, you may jog on the beach before breakfast, join an informal discussion under a cypress tree after lunch, and exchange stories in front of a fireplace at the nightly wine and cheese parties. Guests of conference attendees may enjoy sightseeing along the beautiful Big Sur coast, visiting the new Monterey Aquarium, or shopping in nearby Carmel.

FORST: A 68000
NATIVE-CODE FORTH

JOHN REDMOND - SYDNEY, AUSTRALIA.

∎

I have very mixed feelings about using C. The syntax is dense and helps maximize typing errors, but the statements are powerful. As normally used, it falls far behind (say) Pascal as an algorithmic language, but it is used much more often. It seems to be the most popular applications language, despite the flab which most systems insist on adding to the code. Finally, the code is usually tolerably fast and, for Forth programmers, it is the language to beat!

In their enthusiasm to push the enormous advantages of their own language, Forth programmers opt for a system which is at once powerful, malleable, comfortable and primitive. And they condemn themselves to forging, for the most part, their own tools while the Unix/C environment has all those powerful utilities with the ridiculous names. And, above all, C has access to enormously powerful and flexible I/O functions. As ForST took shape, I decided that it just had to include the I/O functions, its best feature, from C.

Much has been written, many times, about Forth blocks and screens. Disk I/O is based on (usually) 1024-byte chunks of disk space. No operating system directory is used and there is often a problem of reconciling Forth's use of the disk with that of the resident system. The F83 use of blocks within DOS files, for instance, is rather contrived.

ForST's approach is to use buffered TOS files, which permits fully redirectable I/O based on (wait for it!) GETC and PUTC. Once these functions are incorporated, it is extremely simple to carry out file copy and filtering functions.

System File Usage

As normally configured, ForST has eight file structures available for its own use. When source code is compiled from the disk (using LOAD <filename>), the source code can include nested LOAD instructions to a nesting level of seven. This is a powerful and convenient approach, analogous to that of standard Forth, but it uses a special system stack to keep track of the nesting. If a compilation error occurs, all open files are automatically closed.

File Structure

FILE is a very simple defining word:

```
: FILE
  CREATE 24 ALLOT DOES> ;
```

The six 32-bit fields within the structure it creates are used in the following way:

offset 0: #chars in buffer
         4: character pointer
         8: buffer pointer
      12: system file handle
      16: file mode
            (0=input,1=output)
      20: #chars read or written from/
            to buffer so far

(Using 32 bits for each field is wasteful, but simple to follow.)

Opening a File

When FOPEN is used, it expects on the stack the address of a file structure and the operating mode. If the mode is non-zero, a file will be created using FMAKE; otherwise, the low-level word OPEN is used to open an existing file for reading. If successful, it will return a system file handle, which is then kept in the file structure. Then MALLOC is used to allocate a 1024-byte buffer in heap memory. The buffer address is returned and stored in the two pointer fields of the file structure. Finally, the #chars fields are cleared ready for a read or write.

When a file is accessed with GETC for the first time, it attempts to fetch a character from the allocated buffer. When it fails, a 1024-byte disk READ into the buffer is attempted. READ returns the actual number of characters read successfully and the #chars fields are set appropriately. Then a character is fetched from the buffer, the character pointer advanced and the #chars decremented. This will be continued until #chars is zero, when another READ will be necessary. In the event that the buffer is empty and READ returns 0 characters, GETC returns -1 instead of a character.

PUTC carries out its operations in a very similar way. Whenever the buffer is full, it is flushed to the disk before inserting a new character. When FCLOSE is used, it will flush the buffer contents to the disk before using the lower-level MFREE to deallocate the memory buffer and CLOSE to close the file and free the system file handle for further use. Using this approach, forty (!) disk files can be open at any one time.

TOS File Handles

The available handles number 0–45, of which 6–45 may be allocated to disk files. These are the non-standard handles. Standard handles 0–5 are allocated to hardware devices by ForST, in accordance with the TOS designations:

handle 0: console input
          1: console output
          2: serial port (AUX)
          3: parallel port (PRN)
    4 & 5: dummies

It is easy to confuse the different labels associated with files. The file 'handle' is the file descriptor of C and is stored in the array (structure) at the address returned by a named file, e.g.,

```
FILE FILE1 (define the file)
FILE1 0 FOPEN (open it for input)
```

From this point, the address given by `FILE1 + 12` will hold the descriptor, for later use; but most programs will not need to use it. At the user level, `FILE1` is the only file label used.

Block Operations

Although byte-buffered operations are usually the most convenient, block operations are also available with `READ` and `WRITE`. To keep their use intuitive for Forth programmers, their parameters are as for `CMOVE`, e.g.,

```
FILE1 LINEBUFFER LINELENGTH
READ
LINEBUFFER FILE2 LINELENGTH
WRITE
```

These functions are much faster than byte-buffer I/O, but `GETC` and `PUTC` still operate at upwards of 30 Kbytes per second with a hard disk.

System Redirection

`BLK` holds the source descriptor in a standard Forth system. ForST replaces this with `SRC` and adds `DEST` to redirect normal output. When the system looks for input and `BLK` contains zero, it will go to the keyboard buffer; but if `SRC` contains two, it will fetch a character from the serial port. Similarly, if `DEST` contains a non-zero value less than six, it will direct output to a hardware device. If the value in `SRC` or `DEST` is six or higher, it will be interpreted as a file structure address (eg, `FILE1`) and disk I/O will be carried out.

`LOAD` is an important user word which uses system file stuctures:

```
LOAD <filename>
```

The system responds by pushing the value in `SRC` onto the input stack and replacing it with the address of the file structure it has allocated to the source file. If the file, in turn, contains `LOAD` commands, the process is repeated.

ForST allows access to other TOS file utilities. `LSEEK` allows a buffer to be set to any point in the input file:

```
FILE1 0 0 LSEEK
```
is equivalent to rewind,

```
FILE2 -10 1 LSEEK
```
winds the access position back 10 bytes,
```
FILE1 -45 2 LSEEK
```
sets the access position input to 45 bytes back from the end.

`FTELL` returns the present access position of the input or output file. `FDUP` duplicates a standard handle with a disk file handle, and a non-standard handle and `FORCE` forcibly redirects I/O.
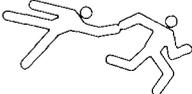
Finally, a point about string compatibility: `TOS` expects a pointer to an uncounted, null-terminated string for any of its arguments. A terminal space—such as is added by `WORD`—has been acceptable for all cases so far tried, but ForST users should be aware (see `NAMEARG` in the code below).

A few simple file utilities are given to illustrate use of the primitives. `COPY` is a good model for more complex filter functions, such as case conversion and character substitution. The conversion between a hybrid F83 file and a normal text file, for example, is trivial.

Next time, the final chapter of this series: the use of named, automatic stack

variables and an analysis of their advantages in writing a floating-point package.

John Redmond is an Associate Professor of Organic Chemistry at Sydney's Macquarie University. He is a "...sometimes-evenings-when-I-have-time programmer" whose chief disappointment of 1988 consisted of attending a plant pathology conference in Acapulco while Forth's own Charles Moore was visiting Sydney. Mr. Redmond welcomes letters from FD readers: 23 Mirool Street, West Ryde, NSW 2114, Australia.

data moves, and be doing them during the time that the processor is not using memory.

As a poor man's emulator, it is a proven concept. Since not all of us have access to a $40,000 analyzer or emulator, the smart RAM is a very powerful way to debug a system. One thing that would be interesting would be to unplug the RAM from a normal system and plug in the smart RAM (along with the appropriate handshake signals). This is the inverse of what most emulators do, since they unplug the processor and emulate it. I think RAM is easier to emulate.

Rob Chapman is a software engineer at IDACOM. He has used a 32-bit Forth in several large projects over the last two years, but now he is much happier with a simpler Forth running on a Forth engine.

```
0 CONSTANT RD
1 CONSTANT WR
FILE FILE1
FILE FILE2

: NAMEARG   32 WORD  0 OVER COUNT + C!  1+ ;

( File dump utility which uses unbuffered file i/o)

: .HEX  <# # # #> TYPE SPACE ;
: .ADDR   CR <# [ASCII] : HOLD
    # # # # # #  #> TYPE 2 SPACES ;
: .BYTES   PAD SWAP 0
    DO COUNT .HEX   I 7 =
    IF SPACE THEN   LOOP   DROP ;
: .CHAR  DUP 32 < IF DROP [ASCII] . THEN EMIT ;
: .CHARS  PAD SWAP 0   DO COUNT .CHAR LOOP   DROP ;
:  DLINE  DUP .BYTES SPACE SPACE .CHARS ;

( Dump a file of any type, opened by low-level OPEN)
:  DUMP  NAMEARG RD OPEN   CLS   HEX  0 ( offset)
    BEGIN   DUP ( offset) .ADDR 16 + ( bump offset)
    OVER ( file handle) PAD 16 READ
    DUP ( bytes read) DLINE
    16 = NOT ( last block) KEY 3 = OR  UNTIL
    DROP ( offset) CLOSE ;
( example dump a:\forth\forst.tos)

( Utilities which use buffered file i/o)

( list a text file)
: LIST  FILE1 RD NAMEARG FOPEN   CR
    BEGIN FILE1 GETC
    DUP 0< ( EOF) NOT   WHILE EMIT REPEAT
    DROP ( EOF char)  FILE1 FCLOSE ;
( example: LIST <fname>)

( copy any file)
: COPY  FILE1 RD NAMEARG FOPEN
    FILE2 WR NAMEARG FOPEN
    BEGIN FILE1 GETC   DUP 0< ( EOF) NOT
    WHILE FILE2 PUTC REPEAT   DROP
    FILE1 FCLOSE   FILE2 FCLOSE ;
( example: COPY <sourcefile> <destfile>)

: CD  NAMEARG CHDIR  0< IF ." cannot set up path " THEN ;
( example: cd a:\forth\examples )

: BLKNAME  GETDTA 30 +  12 32 FILL ;
: FIRST  NAMEARG 47 SFIRST ;
: .FNAME   GETDTA 12 TYPE ;

: DIR  BLKNAME FIRST 0< ( error) NOT
    IF CR .FNAME
        BEGIN BLKNAME SNEXT 0< ( error) NOT
        WHILE .FNAME REPEAT
    THEN ;
( example: dir a:\*.tos)
```

BEST OF
GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

◼

*N*ews *from the GEnie Forth RoundTable*—As the working BASIS being modified by the X3/J14 ANS Forth Technical Committee comes closer to a final document, some of the debate surrounding Forth's future standard seems to be heating up. Recently, one of the *hot potatoes* has been dynamic memory allocation. There are those who think the current tools used in Forth are more than sufficient, and there are those who would 'borrow' concepts incorporated in C.

Read along in the two topic areas devoted to this exchange, and once you have drawn your own conclusions, make them known. You have only yourself to blame if a course is followed you do not agree with.

Category 10
Forth Standards

Topic 36
Memory spaces and position-independent code

Message 11 (Ported from xCFBs)
From: DAVID BREEDING
Subject: Dynamic allocation

Although I consider dynamic allocation to be one of the highest priorities in the new standard, I have yet to read *anything* about it. I keep waiting for someone to bring it up, but somehow no one ever does, so I've finally gotten down to writing myself. All of the so-called "modern" languages support some form of dynamic allocation within the language itself. This memory can be called from fast RAM, cache, or even disk. All of this is transparent to the user.

One of the reasons, I heard, why colleges don't use Forth for teaching is that it leaves out this very thing. I'm not saying that adding DA will make colleges and universities start teaching Forth, but I bet they'll sit up and notice.

Now, how to implement it...

First, let's keep it simple. Two words, DALLOT and UNALLOT. All this does is return an address of n items and then returns this address to a pool. Most of the work in DA is easily accomplished using RAM past HERE, and then feeding the memory back (keeping a linked list of deallocated memory chunks) to HERE after UNALLOT.

Any specialized memory management could be handled by the individual system, only the standard words need to be there for the programmers.

Colleges may not teach Forth because it leaves out dynamic memory allocation.

Message 12
From: B.RODRIGUEZ2

Huh. I guess C and Pascal aren't "modern" languages. Certainly there is no dynamic memory allocation "within the language itself" in C; it's part of the function library. A distinction with a difference! And, while it's been years since I dusted off my Jensen & Wirth, I seem to recall static allocation in Pascal, too.

Local variables are dynamically allocated in both languages, but this topic is called "local variables" in X3J14 circles.

Not to say that dynamic allocation is unnecessary. Nick Solntseff and I recently implemented such a system for our Forth work, using the same words with different names: ALLOC and RELEASE. Soon to be published in *JFAR*, we're told.

I agree, these seem to be the essential primitives. (Damfino why C has so many

forms of alloc. Enlightenment please?) Let me make one further suggestion: in real-time environments it is beneficial to force compaction/garbage collection at a convenient (idle) time, rather than at the usually critical moment when an allocation runs out of room. (Assuming here that your system needs compaction or garbage collection, and you're desperate enough to use such in a real-time problem.) Perhaps a third word, COLLECT, should be defined. Easy to make it a no-op when it's not needed.

But... this should *not* be bound up with the notions of "what is the Forth language" (like Lisp). This should be a standardized *library function*. More power to the X3J14 TC for moving in this direction!
—Brad

Message 13 (Ported from xCFBs)
To: DAVID BREEDING
From: DARRYL BIECH
Subject: Dynamic allocation

Maybe I'm getting a little technical here, but I'm wondering if you were implying that the application will take care of shrinking its claim on system memory (say .COM files, for example) and moving the stack out of the way, etc., prior to deallocations?
—d.b.
NET/Mail: British Columbia Forth Board
Burnaby, B.C., Canada
604-434-5886

Message 14 (Ported from xCFBs)
To: DARRYL BIECH
From: DAVID BREEDING
Subject: Dynamic allocation

That could all be handled on a system level and does not need to be addressed at the "standards" level. There are a lot of ways to implement DA, but a lot of it depends on the hardware setup of the sys-

tem. All I am proposing is a *standard* way of doing DA. All of the particulars (like caching and extended memory use) could be handled by the programmers of the compiler. The only thing that makes sense to me is the inclusion of the two words `DALLOT` and `UNDALLOT`. Which returns an address for use, or returns it to the "heap."

It is a truly exciting subject...when I was in college, we used DA extensively (this was about five years ago). Using DA and recursion, you can do some truly great things that deal with *huge* amounts of data. I have always regretted not having DA as a part of the Forth language.

Message 15 (Ported from *xCFBs*)
To: B.RODRIGUEZ2
From: RAY DUNCAN
Subject: Memory and PIC

Dynamic memory allocation is very useful. All of the LMI Forth systems have had this for several years. But I agree that it should be viewed as an extension to the language rather than part of the core language (similar to its implementation as part of the RTL in C) —it doesn't make any sense to require that this be supported in a ROMmed ANSI Standard Forth kernel, for example.
NET/Mail : LMI Forth Board
Los Angeles, California
213-306-3530

Message 16 (Ported from *xCFBs*)
To: JEFF CYNX
From: RAY DUNCAN
Subj: Comment

My injunction against this is both worldly and spiritual. If you want your program to run properly under multitasking environments such as DesqView, Win 3, OS/2's DOS box, etc., you should be well-behaved in your use of memory. Not using `MALLOC` means that you will not be able to use new capabilities such as `SHELL"` and that your program will not be easily portable to the higher-performance UR/FORTH systems for DOS, OS/2, or 32-bit 80386 protected mode.
NET/Mail : LMI Forth Board
Los Angeles, California
213-306-3530

Message 17 (Ported from *xCFBs*)
To: DAVID BREEDING
From: DARRYL BIECH
Subject: Dynamic allocation

Why not have dynamic allocation as an extension or "standard option," which would be palatable to both small and large implementations of the language?
—d.b.
NET/Mail: British Columbia Forth Board
Burnaby, B.C., Canada
604-434-5886

Category 18
'comp.lang.forth

Topic 86
Subject: Dynamic memory allocation

Message 1 (Ported from UseNet)
Path: willett!dwp
From: dwp@willett.UUCP (Doug Philips)
Newsgroups: comp.lang.forth
Subject: Re: global storage of setjmp()/
   longjmp()

Mitch Bradley writes:
   "(...personally I find a "malloc"ed array of jmp_buf's treated as a stack of recovery points more useful than the potential uses of "foo," but that's just me.)"

Which is exactly the point. ANS Forth `CATCH` and `THROW` implicitly perform this stacking action for you, for free. You don't have to synthesize your own stack. The nested handlers go on the return stack in a very natural and easy-to-implement fashion, and they are automatically removed without any special care on the part of the programmer.
   For free?
   Catch frames must interfere with uses of the return stack in the same way that DO LOOP indices do.
   There must either be:
   1) A pointer to the top-most catch frame.
   2) A unique tag to mark catch frames on the return stack.

   Since no special care on the part of the programmer is required, the first option would require support in `NEXT` and the second option would restrict the kinds of temporaries shoved on the return stack.
   I don't see how it is free. Still, I suppose that having a separate Catch/Throw stack would be exceeding the charter of X3J14. In fact, the scheme you describe must have had some "common practice" to be adopted, or is this not true?
   I would like to hear more details about how it's supposed to work. (Maybe I should

wait until BASIS12 is put online.)
—Doug
P.S. This reminds me somewhat of the alloca controversy for C. (Allocate memory on the stack so that procedure exit/longjmp will automatically reclaim it.)

Message 2 (Ported from UseNet)
From: gary@softway.oz (Gary Corby)
Newsgroups: comp.lang.forth
Subject: Re: C memory allocation

Mitch Bradley writes:
   "(Damfino why C has so many forms of alloc. Enlightenment please?)"

Some reasons:
1) History.
2) C library functions are not arbitrarily constrained to be the "most primitive possible" functions.
3) Different alignment requirements for different data types.

   Another reason: The actual system calls used to change data segment space allocation are brk(2) and sbrk(2). The first sets an absolute boundary and the second alters the boundary relative to the current one. Malloc(), calloc(), talloc(), free(), and friends all come down to brk() and sbrk() in the end. There are "most primitive possible" functions. So primitive, in fact, that nobody in their right mind wants to use them if malloc() or something like it is available.
—Gary
Gary Corby (Friend of Elvenkind)
Softway Pty Ltd
ACSnet: gary@softway.oz
UUCP: ...!uunet!softway.oz!gary

Message 3 (Ported from UseNet)
From: wmb@MITCH.ENG.SUN.COM
   (Mitch Bradley)
Newsgroups: comp.lang.forth
Subject: C memory allocation
Sender:
   daemon@ucbvax.BERKELEY.EDU

Gary Corby writes:
Another reason: The actual system calls used to change data segment space allocation are brk(2) and sbrk(2). The first sets an absolute boundary and the second alters the boundary relative to

REFERENCE SECTION

Forth Interest Group

The Forth Interest Group serves both expert and novice members with its network of chapters, *Forth Dimensions*, and conferences that regularly attract participants from around the world. For membership information, or to reserve advertising space, contact the administrative offices:

Forth Interest Group
P.O. Box 8231
San Jose, California 95155
408-277-0668

Board of Directors

Robert Reiling, President *(ret. director)*
Dennis Ruffer, Vice-President
John D. Hall, Treasurer
Wil Baden
Jack Brown
Mike Elola
Robert L. Smith

Founding Directors
William Ragsdale
Kim Harris
Dave Boulton
Dave Kilbridge
John James

In Recognition

Recognition is offered annually to a person who has made an outstanding contribution in support of Forth and the Forth Interest Group. The individual is nominated and selected by previous recipients of the "FIGGY." Each receives an engraved award, and is named on a plaque in the administrative offices.

1979 William Ragsdale
1980 Kim Harris
1981 Dave Kilbridge
1982 Roy Martens
1983 John D. Hall
1984 Robert Reiling
1985 Thea Martin
1986 C.H. Ting

1987 Marlin Ouverson
1988 Dennis Ruffer
1989 Jan Shepherd

ANS Forth

The following members of the ANS X3J14 Forth Standard Committee are available to personally carry your proposals and concerns to the committee. Please feel free to call or write to them directly:

Gary Betts
Unisyn
301 Main, penthouse #2
Longmont, CO 80501
303-924-9193

Mike Nemeth
CSC
10025 Locust St.
Glenndale, MD 20769
301-286-8313

Andrew Kobziar
NCR Medical Systems Group
950 Danby Rd.
Ithaca, NY 14850
607-273-5310

Elizabeth D. Rather
FORTH, Inc.
111 N. Sepulveda Blvd., suite 300
Manhattan Beach, CA 90266
213-372-8493

Charles Keane
Performance Packages, Inc.
515 Fourth Avenue
Watervleit, NY 12189-3703
518-274-4774

George Shaw
Shaw Laboratories
P.O. Box 3471
Hayward, CA 94540-3471
415-276-5953

David C. Petty
Digitel
125 Cambridge Park Dr.
Cambridge, MA 02140-2311

Forth Instruction

Los Angeles—Introductory and intermediate three-day intensive courses in Forth programming are offered monthly by Laboratory Microsystems. These hands-on courses are designed for engineers and programmers who need to become proficient in Forth in the least amount of time. Telephone 213-306-7412.

On-Line Resources

To communicate with these systems, set your modem and communication software to 300/1200/2400 baud with eight bits, no parity, and one stop bit, unless noted otherwise. GEnie requires local echo.

GEnie
For information, call 800-638-9636
• Forth RoundTable
  (ForthNet link*)
  Call GEnie local node, then type M710 or FORTH
  SysOps: Dennis Ruffer (D.RUFFER), Scott Squires (S.W.SQUIRES), Leonard Morgenstern (NMORGENSTERN), Gary Smith (GARY-S)
• MACH2 RoundTable
  Type M450 or MACH2
  Palo Alto Shipping Company
  SysOp: Waymen Askey (D.MILEY)

BIX (ByteNet)
For information, call 800-227-2983
• Forth Conference
  Access BIX via TymeNet, then type j forth
  Type FORTH at the : prompt
  SysOp: Phil Wasson (PWASSON)
• LMI Conference
  Type LMI at the : prompt
  Laboratory MicroSystems products

Host: Ray Duncan (RDUNCAN)

CompuServe
For information, call 800-848-8990
- Creative Solutions Conference
  Type !Go FORTH
  SysOps: Don Colburn, Zach Zachariah, Ward McFarland, Jon Bryan, Greg Guerin, John Baxter, John Jeppson
- Computer Language Magazine Conference
  Type !Go CLM
  SysOps: Jim Kyle, Jeff Brenton, Chip Rabinowitz, Regina Starr Ridley

Unix BBS's with forth.conf (ForthNet links* and reachable via StarLink node 9533 on TymNet and PC-Pursuit node casfa on TeleNet.)
- WELL Forth conference
  Access WELL via CompuserveNet or 415-332-6106
  Fairwitness: Jack Woehr (jax)
- Wetware Forth conference
  415-753-5265
  Fairwitness: Gary Smith (gars)

PC Board BBS's devoted to Forth (ForthNet links*)
- East Coast Forth Board
  703-442-8695
  StarLink node 2262 on TymNet
  PC-Pursuit node dcwas on TeleNet
  SysOp: Jerry Schifrin
- British Columbia Forth Board
  604-434-5886
  SysOp: Jack Brown
- Real-Time Control Forth Board
  303-278-0364
  StarLink node 2584 on TymNet
  PC-Pursuit node coden on TeleNet
  SysOp: Jack Woehr

Other Forth-specific BBS's
- Laboratory Microsystems, Inc.
  213-306-3530
  StarLink node 9184 on TymNet
  PC-Pursuit node calan on TeleNet
  SysOp: Ray Duncan
- Knowledge-Based Systems
  Supports Fifth
  409-696-7055
- Druma Forth Board

512-323-2402
StarLink node 1306 on TymNet
SysOps: S. Suresh, James Martin, Anne Moore
- Harris Semiconductor Board
  407-729-4949
  StarLink node 9902 on TymNet (toll from Post. St. Lucie)

Non-Forth-specific BBS's with extensive Forth Libraries
- Twit's End (PC Board)
  501-771-0114
  1200-9600 baud
  StarLink node 9858 on TymNet
  SysOp: Tommy Apple
- College Corner (PC Board)
  206-643-0804
  300-2400 baud
  SysOp: Jerry Houston
- Psymatic BBS
  Sunnyvale, California
  408-992-0372
  300 – 2400 baud
  This is a programmer's board with a large Forth area.

This list was accurate as of August 1990. If you know another on-line Forth resource, please let me know so it can be included in this list. I can be reached in the following ways:

Gary Smith
P. O. Drawer 7680
Little Rock, Arkansas 72217
Telephone: 501-227-7817
GEnie (co-SysOp, Forth RT and Unix
RT): GARY-S
Usenet domain.: uunet!wugate!
wuarchive!texbell!
ark!lrark!gars

---

*ForthNet is a virtual Forth network that links designated message bases in an attempt to provide greater information distribution to the Forth users served. It is provided courtesy of the SysOps of its various

---

portable to other 83-Standard Forths, as long as the return-address-save sequences in INIT-TEST and FINISH-TEST are changed to save and restore only a single return stack element for most other Forths. Also, TEST: and ;DONE should be redefined for use with other dictionary structures.

Interactive testing is important and useful (and, in fact, there is no reason why these tools cannot be used as an interactive testing format). However, once initial testing is done, it is often useful to have a permanent test suite in a consistent and readable format. Portions of many programs are so crucial to system operation that they merit a full validation suite to prove correct operation. At Harris, validation suites are being used on the instruction sets of some of the RTX processors. The tools presented here provide a starting point for creating a validation suite for a variety of applications.

Philip Koopman Jr. is a senior scientist at Harris Semiconductor and an adjunct professor at Carnegie Mellon University. The opinions in this article are his, and do not necessarily reflect the views of Harris Semiconductor.

---

the current one. Malloc(), calloc(), talloc(), free() and friends all come down to brk() and sbrk() in the end. So there are "most primitive possible" functions. So primitive in fact that nobody in their right mind wants to use them if malloc() or something like it is available.

Note that, while this is true in Unix, it is not necessarily true in other operating systems. Consequently, while sbrk() is certainly the primitive memory allocation operation for Unix, it does not necessarily even *exist* on all C implementations. In particular, I would expect that it would be difficult to properly implement sbrk() on the Amiga (probably the Amiga C library simulates it with some restrictions). sbrk() assumes that each process has its own address space, which is not generally true. Use of sbrk() is not necessarily portable.

By the way, since brk() can be implemented in terms of sbrk(), sbrk() is the true primitive on Unix systems. In many Unix implementations, sbrk() is the true system call, and brk() is implemented as a library routine, a thin veneer around sbrk().
—Mitch Bradley

To suggest an interesting on-line guest, leave e-mail posted to GARY-S on GEnie (gars on Wetware and the Well), or mail me a note. I encourage anyone with a message to share to contact me via the above or through the offices of the Forth Interest Group.

ADVERTISERS INDEX

FIG
CHAPTERS

The FIG Chapters listed below are currently registered as active with regular meetings. If your chapter listing is missing or incorrect, please contact Kent Safford at the FIG office's Chapter Desk. This listing will be updated in each issue of *Forth Dimensions*. If you would like to begin a FIG Chapter in your area, write for a "Chapter Kit and Application." Forth Interest Group, **P.O. Box 8231, San Jose, California 95155**

U.S.A.
- **ALABAMA**
  Huntsville Chapter
  Tom Konantz
  (205) 881-6483

- **ALASKA**
  Kodiak Area Chapter
  Ric Shepard
  Box 1344
  Kodiak, Alaska 99615

- **ARIZONA**
  Phoenix Chapter
  4th Thurs., 7:30 p.m.
  Arizona State Univ.
  Memorial Union, 2nd floor
  Dennis L. Wilson
  (602) 381-1146

- **ARKANSAS**
  Central Arkansas Chapter
  Little Rock
  2nd Sat., 2 p.m. &
  4th Wed., 7 p.m.
  Jungkind Photo, 12th & Main
  Gary Smith (501) 227-7817

- **CALIFORNIA**
  Los Angeles Chapter
  4th Sat., 10 a.m.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson
  (213) 649-1428

North Bay Chapter
2nd Sat., 10 a.m. Forth, AI
12 Noon Tutorial, 1 p.m. Forth
South Berkeley Public Library
George Shaw (415) 276-5953

Orange County Chapter
4th Wed., 7 p.m.
Fullerton Savings
Huntington Beach
Noshir Jesung (714) 842-3032

Sacramento Chapter
4th Wed., 7 p.m.
1708-59th St., Room A
Bob Nash
(916) 487-2044

San Diego Chapter
Thursdays, 12 Noon
Guy Kelly (619) 454-1307

Silicon Valley Chapter
4th Sat., 10 a.m.
H-P Cupertino
Bob Barr (408) 435-1616

Stockton Chapter
Doug Dillon (209) 931-2448

- **COLORADO**
  Denver Chapter
  1st Mon., 7 p.m.
  Clifford King (303) 693-3413

- **CONNECTICUT**
  Central Connecticut Chapter
  Charles Krajewski
  (203) 344-9996

- **FLORIDA**
  Orlando Chapter
  Every other Wed., 8 p.m.
  Herman B. Gibson
  (305) 855-4790

  Southeast Florida Chapter
  Coconut Grove Area
  John Forsberg (305) 252-0108

Tampa Bay Chapter
1st Wed., 7:30 p.m.
Terry McNay (813) 725-1245

- **GEORGIA**
  Atlanta Chapter
  3rd Tues., 7 p.m.
  Emprise Corp., Marietta
  Don Schrader (404) 428-0811

- **ILLINOIS**
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (708) 713-5365

  Central Illinois Chapter
  Champaign
  Robert Illyes (217) 359-6039

- **INDIANA**
  Fort Wayne Chapter
  2nd Tues., 7 p.m.
  I/P Univ. Campus
  B71 Neff Hall
  Blair MacDermid
  (219) 749-2042

- **IOWA**
  Central Iowa FIG Chapter
  1st Tues., 7:30 p.m.
  Iowa State Univ.
  214 Comp. Sci.
  Rodrick Eldridge
  (515) 294-5659

  Fairfield FIG Chapter
  4th Day, 8:15 p.m.
  Gurdy Leete (515) 472-7077

- **MARYLAND**
  MDFIG
  Michael Nemeth
  (301) 262-8140

- **MASSACHUSETTS**
  Boston Chapter
  3rd Wed., 7 p.m.
  Honeywell
  300 Concord, Billerica
  Gary Chanson (617) 527-7206

- **MICHIGAN**
  Detroit/Ann Arbor Area
  Bill Walters
  (313) 731-9660
  (313) 861-6465 (eves.)

- **MINNESOTA**
  MNFIG Chapter
  Minneapolis
  Fred Olson
  (612) 588-9532

- **MISSOURI**
  Kansas City Chapter
  4th Tues., 7 p.m.
  Midwest Research Institute
  MAG Conference Center
  Linus Orth (913) 236-9189

  St. Louis Chapter
  1st Tues., 7 p.m.
  Thornhill Branch Library
  Robert Washam
  91 Weis Drive
  Ellisville, MO 63011

- **NEW JERSEY**
  New Jersey Chapter
  Rutgers Univ., Piscataway
  Nicholas Lordi
  (201) 338-9363

- **NEW MEXICO**
  Albuquerque Chapter
  1st Thurs., 7:30 p.m.
  Physics & Astronomy Bldg.
  Univ. of New Mexico
  Jon Bryan (505) 298-3292

- **NEW YORK**
  Long Island Chapter
  3rd Thurs., 7:30 p.m.
  Brookhaven National
  Laboratory
  AGS dept., bldg. 911, lab rm.
  A-202
  Irving Montanez
  (516) 282-2540

  Rochester Chapter
  Odd month, 4th Sat., 1 p.m.
  Monroe Comm. College
  Bldg. 7, Rm. 102
  Frank Lanzafame
  (716) 482-3398

- **OHIO**
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom
  (216) 247-2492

- **Columbus FIG Chapter**
  4th Tues.
  Kal-Kan Foods, Inc.
  5115 Fisher Road
  Terry Webb
  (614) 878-7241

  Dayton Chapter
  2nd Tues. & 4th Wed., 6:30
  p.m.
  CFC. 11 W. Monument Ave.
  #612
  Gary Ganger (513) 849-1483

- **OREGON**
  Willamette Valley Chapter
  4th Tues., 7 p.m.
  Linn-Benton Comm. College
  Pann McCuaig (503) 752-5113

- **PENNSYLVANIA**
  Villanova Univ. Chapter
  1st Mon., 7:30 p.m.
  Villanova University
  Dennis Clark
  (215) 860-0700

- **TENNESSEE**
  East Tennessee Chapter
  Oak Ridge
  3rd Wed., 7 p.m.
  Sci. Appl. Int'l. Corp., 8th Fl.
  800 Oak Ridge Turnpike
  Richard Secrist
  (615) 483-7242

- **TEXAS**
  Austin Chapter
  Matt Lawrence
  PO Box 180409
  Austin, TX 78718

Dallas Chapter
4th Thurs., 7:30 p.m.
Texas Instruments
13500 N. Central Expwy.
Semiconductor Cafeteria
Conference Room A
Clif Penn (214) 995-2361

Houston Chapter
3rd Mon., 7:30 p.m.
Houston Area League of PC
Users
1200 Post Oak Rd.
(Galleria area)
Russell Harris
(713) 461-1618

- **VERMONT**
  Vermont Chapter
  Vergennes
  3rd Mon., 7:30 p.m.
  Vergennes Union High School
  RM 210, Monkton Rd.
  Hal Clark (802) 453-4442

- **VIRGINIA**
  **First Forth of Hampton
  Roads**
  William Edmonds
  (804) 898-4099

  Potomac FIG
  D.C. & Northern Virginia
  1st Tues.
  Lee Recreation Center
  5722 Lee Hwy., Arlington
  Joseph Brown
  (703) 471-4409
  E. Coast Forth Board
  (703) 442-8695

  Richmond Forth Group
  2nd Wed., 7 p.m.
  154 Business School
  Univ. of Richmond
  Donald A. Full
  (804) 739-3623

- **WISCONSIN**
  Lake Superior Chapter
  2nd Fri., 7:30 p.m.
  1219 N. 21st St., Superior
  Allen Anway (715) 394-4061

INTERNATIONAL
- **AUSTRALIA**
  Melbourne Chapter
  1st Fri., 8 p.m.
  Lance Collins
  65 Martin Road
  Glen Iris, Victoria 3146
  03/889-2600
  BBS: 61 3 809 1787

Sydney Chapter
2nd Fri., 7 p.m.
John Goodsell Bldg., RM
LG19
Univ. of New South Wales
Peter Tregeagle
10 Binda Rd.
Yowie Bay 2228
02/524-7490
Usenet
tedr@usage.csd.unsw.oz

- **BELGIUM**
  Belgium Chapter
  4th Wed., 8 p.m.
  Luk Van Loock
  Lariksdreff 20
  2120 Schoten
  03/658-6343

  Southern Belgium Chapter
  Jean-Marc Bertinchamps
  Rue N. Monnom, 2
  B-6290 Nalinnes
  071/213858

- **CANADA**
  BC FIG
  1st Thurs., 7:30 p.m.
  BCIT, 3700 Willingdon Ave.
  BBY, Rm. 1A-324
  Jack W. Brown
  (604) 596-9764
  BBS (604) 434-5886

  Northern Alberta Chapter
  4th Sat., 10a.m.-noon
  N. Alta. Inst. of Tech.
  Tony Van Muyden
  (403) 486-6666 (days)
  (403) 962-2203 (eves.)

  Southern Ontario Chapter
  Quarterly, 1st Sat., Mar., Jun.,
  Sep., Dec., 2 p.m.
  Genl. Sci. Bldg., RM 212
  McMaster University
  Dr. N. Solntseff
  (416) 525-9140 x3443

- **ENGLAND**
  Forth Interest Group-UK
  London
  1st Thurs., 7 p.m.
  Polytechnic of South Bank
  RM 408
  Borough Rd.
  D.J. Neale
  58 Woodland Way
  Morden, Surry SM4 4DS

- **FINLAND**
  FinFIG
  Janne Kotiranta
  Arkkitehdinkatu 38 c 39
  33720 Tampere
  +358-31-184246

- **HOLLAND**
  Holland Chapter
  Vic Van de Zande
  Finmark 7
  3831 JE Leusden

- **ITALY**
  FIG Italia
  Marco Tausel
  Via Gerolamo Forni 48
  20161 Milano
  02/435249

- **JAPAN**
  Japan Chapter
  Toshi Inoue
  Dept. of Mineral Dev. Eng.
  University of Tokyo
  7-3-1 Hongo, Bunkyo 113
  812-2111 x7073

- **NORWAY**
  Bergen Chapter
  Kjell Birger Faeraas,
  47-518-7784

- **REPUBLIC OF CHINA**
  R.O.C. Chapter
  Chin-Fu Liu
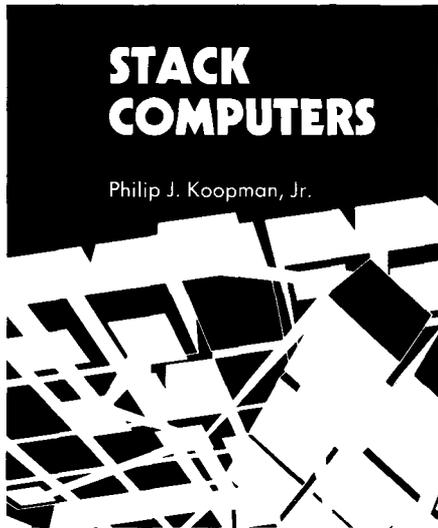 5F, #10, Alley 5, Lane 107
  Fu-Hsin S. Rd. Sec. 1
  TaiPei, Taiwan 10639

- **SWEDEN**
  SweFIG
  Per Alm
  46/8-929631

- **SWITZERLAND**
  Swiss Chapter
  Max Hugelshofer
  Industrieberatung
  Ziberstrasse 6
  8152 Opfikon
  01 810 9289

- **WEST GERMANY**
  German FIG Chapter
  Heinz Schnitter
  Forth-Gesellschaft C.V.
  Postfach 1110
  D-8044 Unterschleissheim
  (49) (89) 317 3784
  Munich Forth Box:
  (49) (89) 725 9625 (telcom)

SPECIAL GROUPS
- **NC4000 Users Group**
  John Carpenter
  1698 Villa St.
  Mountain View, CA 94041
  (415) 960-1256 (eves.)
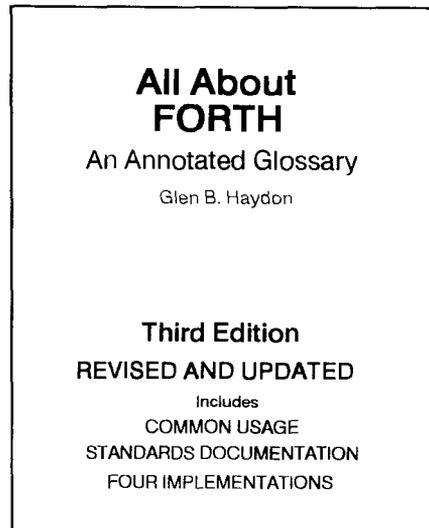
NEW FROM THE FORTH INTEREST GROUP

Forth Interest Group
P.O.Box 8231
San Jose, CA 95155