Mach2 Sys7/32 Documentation

This document details the additional words defined for Mach2. All of these word were used in generating and preparing the System 7 and 32-bit compatible version. These words are all contained in Code Segment 18.

Apple Event Support

High-Level Events

Mach2 now supports High level events, including Apple Events. The `EVENT-TABLE` has been extended to include an entry for the High-Level Event ID, `ID=23`. The Mach2 IOTask code has been modified to include a handler for High Level events. The logic for handling these events is shown below:

```
high-level event
IF
   HLE.Handler 0 =
   IF
      CALL AEProcessAppleEvent
   ELSE
      0 HLE.handler @ EXECUTE    ( result -- result )
      result 0 =
      IF
         CALL AeProcessAppleEvent
      THEN
   THEN
   PAUSE
THEN
```

`HLE.handler` is a new Mach2 system variable that contains a pointer to a routine that handles high level events. This variable is initially set to zero at Mach2 System startup. If a task wants to look at each high level event and perhaps process it, it should hook this variable. The high level event handler should have the following stack interface:

```
my.HLE.handler ( result -- result )
```

This handler executes with the IOTask's stacks and for best results should not `PAUSE`. This handler is initially called with a zero `result` on the stack. The `result` returned by the handler determines whether the IOTask handles this event as an Apple Event. If the handler returns a zero result, then the Apple Event routine `AEProcessAppleEvent` will be called to handle the event. If the result is non-zero, the IOTask will assume that the handler has completely handled the event and proceed to handle the next event.

If in your application, you want multiple tasks to hook this variable, you should establish a chaining mechanism, so that every handler has a chance to examine the HLE.

A `PAUSE` has been included at the bottom of this routine for the following reason. When the IOTask is processing events, it usually does not `PAUSE` until all events have been handled and it receives a null event. If two high-level events came in back-to-back the actions these events would be indicating would both be handled outside of the normal round-robin execution. In order to pass information from an event handler to a task, it is sometimes necessary to use the task variables `UserVector` and `UserData` to pass information and request actions to be performed when the task is given time. If the handler for two sequential HLE's put data in the `UserData` for both HLE's, the second HLE would overwrite the first, effectively losing the first HLE. Therefore a PAUSE was included. This allows your task main loop (if you are organized that way) to separately handle each HLE.

Apple Events
Mach2 now provides a mechanism for installing task-oriented handlers for specific Apple Events. When a HLE comes in that the HLE handler(s) do not claim, the IOTask passes this event to the `AEProcessAppleEvent` function. This function then determines if a handler for this type and ID of event is installed, and if so it calls this handler and passes the AppleEvent information to it. The problem is that because Apple Event handlers execute inside an environment provided by `AEProcessAppleEvent`, they can only depend on the register `A5` being valid. Thus, a handler would be unable to access task user variables in the normal way.

The convention selected for Mach2 is that when a handler is installed into the Apple Event dispatch tables by the application, using the Apple Event function `AEInstallEventHandler`, it should provide the task table address as the reference constant. This `refcon` is passed to the handler when it is called by `AEProcessAppleEvent`. I have created glue code that uses this `refcon` to perform a task switch into the desired task. This sets up the tasks `A3` return stack, the `A4` register to point to the selected task, register `A6` to be the task parameter stack, and register `A7` to be the subroutine stack. Now the handler has access to all of the Mach2 environment.

The effect of this is that when a task `PAUSE's`, the IOTask will eventually get its turn in the round-robin. If an Apple Event comes in and is dispatched to `AEProcessAppleEvent`, which then calls a handler owned by the `PAUSE`'d task, the handler will begin executing with the task state the same as when the task `PAUSE`'d. When the handler exits, the task returns to a `PAUSE`'d state, and control returns through `AEProcessAppleEvent` to the IOTask. The IOTask then `PAUSEs`, and the `PAUSE`'d task regains control and begins its normal execution flow.

There are two ways to build an Apple Event Handler. The first provides more control over the handler execution flow:

```
: my.AEHandler       ( theAppleEvent reply -- OSErr )
   AEHandler.entry   ( glue code is MACHro'd in here )
     .
   ( execution code goes here )
     .
   AEHandler.exit    ( glue code is MACHro'd in here )
   ;
```

The second is a more compact calling sequence.

```
AE: my.AEHandler  ( theAppleEvent reply -- OSErr )
     .
   ( execution code goes here )
     .
   ;AE
```

Special thanks to Chris Heilman of Pocket Forth for the idea of this second way. Both ways are exactly equivalent in function. The advantage of using the second way is that a local variable specification can be used in the definition to get the input parameters. With the first method, `my.AEHandler` would have to call a separate inner word to be able to use local variables.

In either case, the actual handler code can expect on the stack the pointers to `theAppleEvent` and the `reply` Apple Event. The handler must return only an `OSErr` on the stack at handler exit.

To install the handler in the AppleEvent Dispatch table, use the following code:

```
#kCoreEventClass #kAEOpenDocuments  ( CLASS and ID constants )
['] my.AEHandler                    ( address of handler )
myTask                              ( Task address )
0                                   ( system dispatch table flag )
CALL AEInstallEventHandler          ( -- OSErr )
```

In assembly it would be:

```
EXG.L    D4,A7                    \ switch to trapStack
SUBQ.L   #2,A7                    \ allocate space for result
MOVE.L   #kCoreEventClass,-(A7)   \ push Event class
MOVE.L   #kAEOpenDocuments,-(A7)  \ push event ID
LEA      MyHandleODoc,A0          \ get addr of handler
MOVE.L   A0,-(A7)                 \ push addr of handler
MOVE.L   A4,-(A7)                 \ push TASK addr as refcon
CLR.W    -(A7)                    \ FALSE for system dispatch tables
MOVE.W   #$091F,D0                \ routine selector
         _AEInstallEventHandler   \ it's really _Pack8
MOVE.W   (A7)+,D0                 \ get OSErr
EXT.L    D0                       \ long-extend the result
MOVE.L   D0,-(A6)                 \ push on stack
EXG.L    D4,A7                    \ return to FORTH stack
```

I don't think it would be a good idea for the handler to `PAUSE`. Inside Mac, Volume 6 is not quite clear on this point as to the effect of calling `WaitNextEvent` from inside the handler, and to `PAUSE` will have that effect. If a second Apple Event came in and was processed, would the first have to be suspended? Is `AEProcessAppleEvent` reentrant? I just don't know what the correct procedure is. Inside Mac is silent on whether you can process simultaneously two Apple Events without suspending the first. Inside Mac also doesn't completely discuss how to use the routines `AESuspendAppleEvent` and `AEResumeAppleEvent` from inside the Apple Event handler context. I'm also not sure of the effect on the Mach2 multitasking loop. Each time through this `PAUSE`-`AEProcessAppleEvent`-handler loop adds about 300 bytes to the `TRAPSTACK`. Best not to do it, although I think you could get away with it. If you were careful, your routines were reentrant, and you always eventually returned to `AEProcessAppleEvent` from your handler, I think that you could nest quite a few levels deep before you ran out of TrapStack or Subroutine stack.

Installable Name Modules

I have included several words used to support the loading of temporary constant definitions. This came about because I found when I was developing code, being basically lazy, I would rather load in big files of Macintosh constant values rather than cut and paste from these files into my programs every time. The problem was, the default Name Area provided by Mach2 is too small, only 16 kbytes (the handle is actually $5000 bytes long), and while loading these big files I would load right out of the bottom of the name space into something else and crash Mach2.

Basically, I save the current value in `NP`, allocate a handle, and store a pointer to it in NP. When the constant definitions are done, just put back into `NP` the original value. Since Mach2 uses the value in `NP` to determine where to put the next definition, it works. But if you look at the code, you can see it is a little more complicated than that.

The basic sequence looks like this:

```
5000 Insert.MODULE _MyModule_
  .
( compile and load as you normally would )
  .
_MyModule_ restore.Name.Space
```

`Insert.MODULE` requires the size of the desired name space to be on the stack. You must have predetermined the size of the name module needed. What I do to figure this out is to use the FORTH word `?FREE`. By looking at the free space in the name space before and after loading a file normally, the name space size for this load module can be determined. Add a few hundred byte to this (to allow for growth of your load file) and then you have the size of the name space needed.

`Insert.MODULE` creates a word call _`MyModule`_ (or whatever you want) that is used by the other module words. In addition, it allocates a handle of the requested size and make the `NP` point to this handle. The defining _`MyModule`_ code and name data is also stored in this handle. All subsequent words defined up to `restore.Name.Space` are stored in this module.

`restore.Name.Space` resets `NP` to point back to the standard name space, but leaves the just-loaded module linked in so that the name field definitions can be used.

Module loading can be nested. If you are in the middle of loading a module, you can execute `Insert.MODULE` again to begin loading another module. However, you **must** balance your calls with `restore.Name.Space`, and you must back out in the reverse order that you went in, like this:

```
5000 Insert.MODULE MyFirst
     .
   10035 Insert.MODULE MySecond
     .
   MySecond restore.Name.Space
     .
MyFirst restore.Name.Space
```

Failure to follow this rule will have dire and unpredictable consequences.

When you no longer need the definitions, you can delete the module like this:

```
_MyModule_ forget.MODULE
```

Never FORGET a module or a word inside a module. Always use forget.MODULE. Modules currently in use can be forgotten in any order. There is not a restriction to forget in the same or reverse order as they were defined.

There is a gotcha when hashed searches are in effect. Because of the nature of how entries are stored, searched, and deleted in the hash table, I am currently unable to erase the entries in the hash table belonging to the name module. If hashing is enabled, and a module is forgotten, and then a word is later referenced from this module, there is a finite probability that the search algorithm will return an address that is no longer valid. Use of this address could have dire consequences. So the moral of the story is: don't attempt to use a word in the module after the module has been forgotten.

When hashing is disabled, I can always guarantee that an attempt to search for a word in a forgotten module will fail normally without any consequences.

Another good reason for not using hashing is that when you load many modules, the current Mach2 hashing algorithm with its 16-bit hash values starts getting too many hash hits, and seems to have problems finding the correct word. Some day I will fix this hashing feature and put in 32-bit hash values, and then this problem will go away.

You **must** unlink and forget all modules currently in use before using NEW-SEGMENT, INSTALL, WORKSPACE, or (INSTALL). These words save the current dictionary image, and only the dictionary image, not the loaded-in modules. If modules are in use, the offsets that link into them are only valid for that run-time. The first time after using an INSTALL or NEW-SEGMENT, these offsets would be invalid and the search routine would blow up.

The FORTH word EMPTY has been redefined to know about loaded modules and dispose of the handles.

There is another good reason to use this facility. When creating applications that are not stand-alone, by storing the names information of executable words within a module that is later forgotten, the dictionary size is kept small, and the internal words are completely hidden from the user (and you too!). Using WORDS to list the dictionary will show only those words that the user should know about.

However, the following rules apply. Never store VARIABLEs, GLOBAL words, TASK's, Menu's, Windows, Menu-Bars, Controls, and VOCABULARY's in the module. Mach2 maintains for various reasons internal linked lists in the name space for these word types. If any of these types of defintions were stored in a module, and the module were later forgotten, the linked list would be broken with potentially dire problems.

Enhanced Trap Compiler

The trap compiler has been enhanced and brought up to date. I have included a file called `Trap Compiler.4th` that is used to compile the new traps. If this file is loaded, a new `TEXT 2` trap interface resource can be generated by executing the word `compile.traplist`. The first thing this word is going to do is bring up the Standard File dialog to ask you to select the Trap definition file. I have included this file, called `Trap Listing.4th`. `Compile.traplist` will read in this file and generate a new `TEXT 2` resource, which it will write out the file "`my.Trap.Data`". You can then use ResEdit to paste this resource into Mach2. The new or modified trap definition will be available immediately from the `CALL <trap>` interface.

To edit the file `Trap Listing.4th`, you need to know the format. Each line of text in this file must contain one trap definition. The format is very rigid and no comments are allowed anywhere. The basic format is:

```
    trapname trapword [SPECIAL] TrapType [selector]
stack-spec
```

`trapname` can be any string up to 31 characters long. This string is what the trap is named. Mach2 will search for this name. The trapname you type in in your source code must match this name, however, case is ignored. My trap compiler internally converts all letters to uppercase before performing a comparison. Note that in the resource, only the first 14 letters are stored, and this is what is seen when `TRAPLIST` is executed. However, when performing comparisons, a hashing algorithm is used, and this algorithm uses the entire input string. The hash number stored in the `TEXT 2` resource is based on the entire `trapname`.

`trapword` must be a 4-digit hex representation of the trap word. Do not use a leading dollar sign.

The optional word `SPECIAL` is included only if the trap compiler code has a special handler for this trap. Currently the only special handler defined is for the Communications Toolbox words, trap number `A08B`. Do not use this command for any other trap.

`TrapType` can be one of two values:

```
    OSTrap
    ToolTrap
```

Operating system traps take their input values in registers. ToolTraps take theirs on the stack. There are exceptions to both, and they can be seen by closely inspecting `Trap Listing.4th`.

The optional `selector` field has several variants shown below:

```
1234              \ compile a decimal word routine selector on the stack
$1234             \ compile a hex word routine selector on the stack
REG( 1234 )       \ compile a decimal word routine selector in D0
REG( $1234 )      \ compile a hex word routine selector in D0
LREG( 12341234 )  \ compile a decimal longword routine selector in D0
LREG( $12341234 ) \ compile a hex longword routine selector in D0
STACK( 1234 )     \ compile a decimal word routine selector on the stack
STACK( $1234 )    \ compile a hex word routine selector on the stack
LSTACK( 12341234 )   \ compile a longword routine selector on the stack
LSTACK( $12341234 )  \ compile a longword routine selector on the stack
```

These routine selectors are compiled as specified regardless of whether the trap is an operating system or Toolbox trap.

The `stack-spec` has the following form:

`( [A0] [D0] [A1] -- [A0] [D0] [A1] )` (for Operating System Traps)

`( [<W16 | W32>] ... -- [<W16|W32>] )` (for Toolbox Traps)

For operating system traps, the register order as shown above must be followed. This is the order that Mach2 will fill registers from items on the stack. If a register is to be included in either the input or output, it must appear in its proper place relative to the other registers. Each register spec must have a size appended to it. The allowable register specs are:

```
A0.W        A0.L
D0.W        D0.L
A1.W        A1.L
```

Note: The order that Mach2 fills the registers sometimes results in a stack spec that is different than the Pascal calling sequence specified in Inside Mac. Always use the Mach2 register specification, not the Pascal sequence. Someday, when the `CALL trap` mechanism is entirely rewritten, this potential problem will go away.

For Toolbox traps, you can have from zero up to 15 inputs. You are allowed only only result to be returned.

Close inspection of the listing file will show all the variations of this specification. I have taken care to make this interface natural and readable.

Desk Accessory Support

I have included the semi-standard words for compiling DA glue code in the `DEVELOPMENT` vocabulary. The three routines used to help generate a DA code image are:

```
DA.prelude   ( compile glue code for routine entry )
DA.epilog        ( compile glue code for routine exit
)
set.DA.stack=    ( set the stack size in the routine
entry )
```

These words can be used as follows:

```
:RECORD DRVRHeader   ( define the Device Driver header record )
   drvrFlags   word  ( see the section on records for more    )
   drvrDelay   word  ( information on how to define and       )
   drvrEMask   word  ( allocate records                       )
   drvrMenu    word
   drvrOpen    word
   drvrPrime   word
   drvrCntl    word
   drvrStatus  word
   drvrClose   word
   drvrName    0  ( this will be allocated separately )
; RECORD

( ===== the DA Code image begins here ===== )

DRVRHeader my.DA.Header CodeRec
   DC.B  5
   DC.B  0
   DC.B  'myDA'

.ALIGN

( the various routines go here )

( define the handlers for each type of call)
: my.Open
   DA.prelude  set.DA.Stack= 2000
     .
   ( Driver Open code goes here )
     .
   DA.epilog
   ;

: my.Prime
   DA.prelude  set.DA.Stack= 4000
     .
   ( Driver Prime code goes here )
     .
   DA.epilog
   ;

: my.Control
   DA.prelude  set.DA.Stack= 3000
     .
   ( Driver Control call code goes here )
     .
   DA.epilog
   ;

: my.Status
```

```
   DA.prelude  set.DA.Stack= 2000
     .
   ( Driver Status call code goes here )
     .
   DA.epilog
   ;

: my.Close
   DA.prelude  set.DA.Stack= 2000
     .
   ( Driver Close code goes here )
     .
   DA.epilog
   ;

HEADER myDA.End  ( end of the DA Code image )

( now install the routine offsets )
' my.Open    ' my.DA.Header -  drvrOpen    .OF. my.DA.Header !
' my.Prime   ' my.DA.Header -  drvrPrime   .OF. my.DA.Header !
' my.Control ' my.DA.Header -  drvrControl .OF. my.DA.Header !
' my.Status  ' my.DA.Header -  drvrStatus  .OF. my.DA.Header !
' my.Close   ' my.DA.Header -  drvrClose   .OF. my.DA.Header !

( and we are done with defining the DA )
```

Stand-Alone Code Support

The following semi-standard words have been installed to support the generation of stand-alone and callback code images.

```
:xdef             ( compile the XDEF entry glue code )
;xdef             ( compile the XDEF exit glue code 
and
                    the XDEF entry point )
INIT.prelude      ( compile FORTH environment setup 
glue )
                  ( code for XDEF entry )
INIT.epilog       ( compile FORTH environment teardown 
)
                  ( glue code for XDEF exit )
set.INIT.stack=   ( set the XDEF stack size )
```

These words can be used as shows below:

```
( ===== Begin XDEF code image ===== )

:xdef Password.XDEF.body

HEADER   the.Password.str
   DC.B  8
   DC.B  'Password'
   DCB.B 21,0

.ALIGN

( the various other code routines go here )

: password     ( my.XDEF.ptr -- )
   ( the basic execution word )
   ;

: Password.XDEF
   INIT.prelude  my.INIT.stack= 2000
   password
   INIT.epilog
   ;

' Password.XDEF ;xdef

HEADER my.Password.end

( this is the end of the code image )
```

Record Definitions and Support

The basic form of a record definition is:

```
:RECORD myRecordDef         ( you can put a comment here )
   var1  <type>             ( comments can be put here )
   var2  4                  \ this type of comment is also OK
   var3  SizeOf( other.record )
   var5  <type>
;RECORD
```

The `var1`, `var2`, etc. all become constant definitions and can be used anywhere. The `type` field must resolve to a constant, and can be a predefined type, a literal, or the operator `SizeOf(` can be used to return the size of another record definition created by `:RECORD`. Each of the record variables (`var1`, `var2`, …) must not have been defined before, since each will result in a new constant definition. Two or more record variables that share the same location can be defined by setting the type field of all but the last equal to zero. As in:

```
:RECORD sally
   var1  Handle   ( holds handle to frebistat )
   var2  0        ( holds ptr to silly )
   var3  0        ( holds ptr to metoo )
   var4  pointer  ( holds pointer to tambien )
   var5  Handle   ( another handle )
;RECORD
```

`var2`, `var3`, `var4` will all have the same offset=4 in the record.

Nested record definitions are not supported. Comments are allowed anywhere except between the record variable name and the type identifier. See the file `FileMgr.4th.inc` for more examples.

Record storage is allocated as one of the following six types:

1. Direct Variable

```
... RecordDef myRecord VarRec ...
```

This allocates a variable space record named `myRecord` of type `VarRec`. `myRecord` will behave like a normal `VARIABLE`. The record name and the record type are read from the input stream by the Record definition word `RecordDef`. Use `myRecord` where ever you would use a standard FORTH variable. Enough variable space is allocated to hold one record of type `RecordDef`. If you want the record to be available across segments, you must use the word `GLOBAL` before defining the record, as in:

```
... GLOBAL RecordDef myRecord VarRec ...
```

`myRecord` will then have an entry in the jump table.

2. Indirect Variable

```
... RecordDef myRecord *VarRec ...
```

This allocates a 4-byte variable called `myRecord` in the `VARIABLE` space to hold a pointer to a Record of the size `RecordDef`. The record name and the record type are read from the input stream by the Record definition word `RecordDef`. Use myRecord where ever you would use a standard FORTH variable. It is the programmer's responsibility to ensure that the pointer stored at the address returned by myRecord is a valid pointer to a storage space for the record. If you want the record to be available across segments, you must use the word GLOBAL before defining the record, as in:

```
... GLOBAL RecordDef myRecord *VarRec ...
```

myRecord will then have an entry in the jump table.

3. Direct CODE

```
... RecordDef myRecord CodeRec ...
```

This allocates a CODE space record at HERE. myRecord will then behave like a normal VARIABLE. Use it where ever you would use a standard FORTH variable. The record name and the record type are read from the input stream by the Record definition word RecordDef. Enough space is allocated in the code segment being compiled to hold one record. RecordDef should never be asked to allocate a CodeRec when another word is in the middle of being compiled. If you want the record to be available across segments, you must use the word GLOBAL before defining the record, as in:

```
... GLOBAL RecordDef myRecord CodeRec ...
```

myRecord will then have an entry in the jump table.

4. Indirect CODE

```
... RecordDef myRecord *CodeRec ...
```

This allocates a 4-byte variable at HERE in the CODE space to hold a pointer to a Record of the size RecordDef. Use myRecord where ever you would use a standard FORTH variable. The record name and the record type are read from the input stream by the Record definition word RecordDef. It is the programmer's responsibility to ensure that the pointer stored at the address returned by myRecord is a valid pointer to a storage space for the record. RecordDef should never be asked to allocate a *CodeRec when another word is in the middle of being compiled. If you want the record to be available across segments, you must use the word GLOBAL before defining the record, as in:

```
... GLOBAL RecordDef myRecord *CodeRec ...
```

myRecord will then have an entry in the jump table.

5. Direct Local Variable

Used as follows:

```
: myword { | [ SizeOf( RecordDef ) 4- LALLOT ] myrec }
    .
... var2 .OF. ^ myrec ...
    .
;
```

See the description of the word `.OF.` for more information. No explicit allocation is performed outside of a definition for this type of record. The word `.OF.` will compile the correct offset from `A2` for this record. If `SizeOf( RecordDef )` is 32, and the offset of `var2` is 8, then the instructions

```
LEA         -24(A2),A0
MOVE.L      A0,-(A6)
```

will be compiled for the above definition.

6. Indirect Local Variable

Used as follows:

```
: myword { input |  myrec }
     .
  ... ( mypointer -- ) -> myrec ...
  ... var2 .OF. myrec ...
     .
  ;
```

No explicit allocation is performed outside of a definition for this type of record. At some point during the execution of the word, the local variable `myrec` should be set to a valid pointer to a record. The `.OF.` operator will compile the following code sequence:

```
MOVEA.L  $-8(A2),A0
MOVE.L   var2(A0),-(A6)
```

See the description of the word `.OF.` for more information.

To use records, the syntax is always:

```
<constant> .OF. <recordname>
```

The word `.OF.` is smart and knows how to recognize each type of record, and knows whether the record reference is being compiled or interpreted. If the record reference is being interpreted, the addr of the record variable is left on thestack. If the record reference is being compiled, at execution time the record address will be left on the stack. In addition, `.OF.` performs certain optimizations for compiled code.

Conditional Compilation

The following set of words are in the `DEVELOPMENT` vocabulary. They are used for conditional compilation sequences. I did not find the Mach2 facilities very useful, so I wrote these to behave like C. Some might call that regression, but I have found them useful for include files. Now I never have to retype in all the Macintosh constants, just load them.

```
#ifdef  <symbol>
#ifndef <symbol>
#else
#endif
#define <symbol>
```

They behave just like they do in C. Especially since they are all defined `IMMEDIATE`, they will execute within a colon definition. These words always operate on the current input stream. The basic form is:

```
#ifdef mysymbol
   .
   ( continue interpreting/compiling everything up to the
     next #ifdef, #ifndef, #else, or #endif )
   .
#else
   .
   ( interpret/compile this section if mysymbol is not defined )
   .
#endif
```

`#ifdef`'s can be nested. `#ifndef` behaves the opposite of `#ifdef`. A typical example, taken from one of my include files, shows the various ways this can be used.

```
#ifndef _COMPATIBILITY_    ( then compile this include file )

   #ifdef _MODULES_
      5000 Insert.MODULE _COMPATIBILITY_
   #else
      #define _COMPATIBILITY_
   #endif
   .
#ifndef _SYSEQU_
   ( make sure that dependencies are first loaded )
   INCLUDE" :Includes:SysEqu.Txt"
#endif
   .
#ifdef _EMBEDDED_
   .( Mac Compatibility words compiled for embedded code applications.)
   CR
#endif
   .
: NumToolboxTraps ( -- number )
   $6E NGetTrapAddress.Tool   ( _InitGraf )
   $AA6E NGetTrapAddress.Tool
   =
   IF $200 ELSE $400 THEN
   ;
   #ifdef _EMBEDDED_
      MACH
   #endif
   .
   #ifdef _MODULES_
      _COMPATIBILITY_ restore.Name.Space
   #endif
#endif ( to balance the #ifndef at the start )
```

Additions to the FORTH vocabulary

`(HERE)` — **`GLOBAL VARIABLE`**

A Mach2 system variable definition for `HERE`. This will be eventually be used for inline loadable binary code modules and other such stuff.

`HLE.handler` — **`GLOBAL VARIABLE`**

A Mach2 system variable for a High Level Event Handler chain. Initialized to zero by my startup code. If Mach2 receives a High-level event, and the value at this address is non-zero, Mach2 will `JSR` to this address. A high level event handler can then perform any action desired. The routine must return a value on the stack. If the value returned on the stack is non-zero, Mach2 will do nothing. If the value is zero, Mach2 will assume that no High-Level handler is interested in this event, and will pass it to the Apple Event handler.

The following task words were permanently defined.

```
0    GLOBAL USER NEXT_TASK
4    GLOBAL USER S0
8    GLOBAL USER PS
12   GLOBAL USER RETURN_STK
40   GLOBAL USER HEAD
44   GLOBAL USER TAIL
48   GLOBAL USER CTR
52   GLOBAL USER PTR
56   GLOBAL USER ECHO
60   GLOBAL USER FILEID
62   GLOBAL USER V/WD.RefNum
64   GLOBAL USER CONTEXT
68   GLOBAL USER CURRENT
72   GLOBAL USER TaskWindowPointer
76   GLOBAL USER ABORT-ACTION
80   GLOBAL USER (ABORT)
84   GLOBAL USER (NUMBER)
88   GLOBAL USER (EXPECT)
92   GLOBAL USER (TYPE)
96   GLOBAL USER (?TERMINAL)
100  GLOBAL USER (QUERY)
104  GLOBAL USER PenLocation
108  GLOBAL USER TaskMenuBar
116  GLOBAL USER MenuData
124  GLOBAL USER ControlData
128  GLOBAL USER ControlHandle
136  GLOBAL USER DialogData
140  GLOBAL USER DialogHandle
144  GLOBAL USER UserVector
```

```
148 GLOBAL USER UserData
152 GLOBAL USER CONTENT-HOOK
156 GLOBAL USER DRAG-HOOK
160 GLOBAL USER GROW-HOOK
164 GLOBAL USER GOAWAY-HOOK
168 GLOBAL USER UPDATE-HOOK
172 GLOBAL USER ACTIVATE-HOOK
176 GLOBAL USER DEVICE_EXPECT
180 GLOBAL USER DEVICE_QTERM
184 GLOBAL USER DEVICE_TYPE
```

```
188 GLOBAL USER ATALK_SOCKET
190 GLOBAL USER DIALOG-HOOK
194 GLOBAL USER ZOOMIN-HOOK
198 GLOBAL USER ZOOMOUT-HOOK
202 GLOBAL USER C_Action
212 GLOBAL USER FileI/OID
```

EMPTY (--)

I had to redefine `EMPTY` so that any active name modules would be de-allocated. The action of this is transparent to the user. See the section on Modules for more information.

Additions to the MAC vocabulary

`CmpString ( str1 str2 - flag )` **GLOBAL**

The word `CmpString` was defined for the OS Utilities routine `_CmpString` because the call interface is non-standard. This routine takes two string addresses and compares the strings. If the strings compare, this routine returns a `-1`.

I will eventually install a special handler in the the `CALL` facility for this word. That way we can have the `CmpString` trap with usable trap modifiers callable from FORTH. The interface would then be `CALL CmpString,MARKS,CASE`.

The following Macintosh Record Types were defined. See the section on Records for more information on these constants.

```
4    CONSTANT pointer
1    CONSTANT byte
1    CONSTANT char
2    CONSTANT integer
2    CONSTANT short
2    CONSTANT Boolean
4    CONSTANT longWord
4    CONSTANT long
4    CONSTANT float
8    CONSTANT double
10   CONSTANT extended

( ===== Macintosh-specific types ===== )
2    CONSTANT OSErr
4    CONSTANT OSType
4    CONSTANT Ptr ( Careful!  There is a PTR in the Mach2 
Task
                   Variables in the FORTH vocabulary )
4    CONSTANT Handle
4    CONSTANT Fixed
4    CONSTANT Fract
Ptr  CONSTANT ProcPtr
4    CONSTANT Size
```

(=== SysEnvirons Support Words. ===)

The SysEnvirons SysEnvRec was defined in the Mac Vocabulary .

```
:RECORD SysEnvRec
    environsVersion  short
    machineType       short
    systemVersion         short
    processor             short
    hasFPU            char
    hasColorQD        char
    keyBoardType      short
    atDrvrVersNum         short
    sysVRefNum        short
;RECORD
```

The following set of words are basically pulled out of Inside Mac, Vol 6, and are used to test for the existence of a trap.

NGetTrapAddress.Tool (trap# -- addr) **MACH**

Calls `_GetTrapAddress` for a ToolBox routine. Does not check the passed trap number to see if it is a valid ToolTrap.

NGetTrapAddress.OS (trap# -- addr) **MACH**

Calls `_GetTrapAddress` for a Operating System routine. Does not check the passed trap number to see if it is a valid Operating System Trap.

NumToolboxTraps (-- number)

Test for the size of the Trap Dispatch table. Returns 512 if it is a 64K ROM trap dispatch table. Returns 1024 if the Expanded trap dispatch table is in use.

GetTrapType (trap -- traptype)

Takes the full 16-bit trap word, and returns 0 if it is a OSTrap. Returns 1 if a Toolbox Trap.

TrapAvailable? (trap -- flag) **GLOBAL**

Takes the 16-bit trap value and returns -1 if the trap exists.

Gestalt.Exist? (-- flag) **GLOBAL**

A specific application of `TrapAvailable?`.

SysEnvirons.Exist? (-- flag) **GLOBAL**

A specific application of `TrapAvailable?`.

setmachineType (-- n)

Returns the following values for specific Mac types:

`-2`	Lisa
`-1`	128K or 512K Mac
`0`	Unknown Mac
`1`	Mac 512KE
`2`	Mac Plus
`3`	Mac SE
`4`	Mac II

set.processor.type (-- n)

Extracts the processor type from the Macintosh global `CPUFlag`. If the value is greater than 3, this word returns zero.

set.FPU.exist (-- flag)

Returns 1 if an FPU exists. Returns 0 if it doesn't. This word reads the information from the Macintosh global `HWCfgFlags`.

`set.Color.QD.exist ( -- n )`

Returns 1 if Color QuickDraw exists. Returns 0 if it doesn't.

get.keyboard.type (-- type)

Returns the keyboard type constant using the Macintosh global `KbdType`. This routine performs the following mapping from the value stored in `KbdType` to the returned value.

```
KbdType     $03 $13 $0B $02 $01 $06 $07 $04 $05 $08 $09
            |   |   |   |   |   |   |   |   |   |   |
SysEnvirons $01 $02 $03 $04 $05 $06 $07 $08 $09 $0A $0B
            |   |   |   |   |   |   |   |   |   |   |
            |   |   |   |   |   |   |   |   |   |   Apple Keyboard II (ISO)
            |   |   |   |   |   |   |   |   |   Apple Keyboard II
            |   |   |   |   |   |   |   |   Apple Extended Keyboard (ISO)
            |   |   |   |   |   |   |   Apple Standard Keyboard (ISO)
            |   |   |   |   |   |   Portable Keyboard (ISO)
            |   |   |   |   |   Portable Keyboard
            |   |   |   |   standard Apple Desktop Bus keyboard
            |   |   |   Apple extended Kbd
            |   |   Macintosh Plus keyboard
            |   Macintosh keyboard and keypad
            Macintosh keyboard
```

get.AppleTalk.Version (-- version)

Get the AppleTalk version currently in use from the .MPP driver.

HGetVInfo (volume.ID @file.ioPB @vol.name -- result) GLOBAL

This routine uses the volume reference number, a passed-in address for a VolumeInfo parameter block, and the address of the volume name string, and calls the ROM routine `_HGetVInfo`.

get.THE.blessed.WD (-- WDRefNum) GLOBAL

This routine gets the Working directory number of the blessed folder that contains the current open system file - use this routine when `SysEnvirons` is not available. If an error occurred the routine will return zero.

fake.SysEnv (SysEnvRec version -- SysEnvRec result)

Call this routine exactly like `_SysEnvirons`. It completely bypasses the `_SysEnvirons` trap and collects all of its data by looking at various low-memory globals.

GLOBAL SysEnvirons (SysEnvRec version -- SysEnvRec result)

Call this routine exactly like `_SysEnvirons`. If `_SysEnvirons` exists, this word calls it. If the trap does not exist, this word will use `fake.SysEnv` to return as much information as possible.

(CALL).SysEnvirons (SysEnvRec version - SysEnvRec result) GLOBAL

Call this routine exactly like `_SysEnvirons`. If `_SysEnvirons` exists, this word calls it. If the trap does not exist, this word will use `fake.SysEnv` to return as much information as possible. The only difference between this routine and `SysEnvirons` is that this routine does not switch the A7 stack to the TrapStack when calling `_SysEnvirons`. This makes it useful for calling in system callback routines such as CDEF's and idle procs. If you really need to make stand-alone code, the file `Compatibility.4th` can be loaded to create a stand-alone version that will be MACHro'd in at compile time.

Definitions in the `DEVELOPMENT` vocabulary

`VOCABULARY DEVELOPMENT` **`GLOBAL`**

A new vocabulary was defined to hold my non-standard code development words.

`MODULE.list` **`GLOBAL VARIABLE`**

A Mach2 system variable that holds the pointer to the start of the linked list for any loaded name modules. Initialized to zero by my startup code, manipulated by the module words `Create.MODULE`, `forget.MODULE`, `restore.Name.Space`, and a redefined `EMPTY`. See the `Modules.inc` file for more information. The linked list structure looks like this:

```
              |------------------------------------|
              |                                    |
---------     |   ---------         ----------     |
|  ptr  | ---|   | handle | <--    | handle | <--- |
---------         ----------   |    ----------
                  |   0    |   --- |  ptr   |
                  ----------        ----------
```

The handle is the handle to each allocated code module.

`StripAddress.mask` **`GLOBAL VARIABLE`**

A Mach2 system variable that holds the `_StripAddress` mask as recommended by Apple Tech Notes. Set at startup time by my startup code.

`MACH2.SysEnvRec` **`GLOBAL VARIABLE`**

A Mach2 system variable that holds a `_SysEnvirons` record filled in at Mach2 Startup by my startup code.

`MACH2.flags` **`GLOBAL VARIABLE`**

A Mach2 system variable that holds various startup flags set at Mach2 startup by my startup code.

bit 0 = Gestalt exists
bit 1 = Apple Events exists
bit 2 = SysEnvirons exists
bit 3 = WaitNextEvent exists
bit 4 = Used internally during System 7 COLD startup

`get.A5 ( -- A5.value )` **`MACH`**

This word returns the current value in the A5 register. Currently defined as:

```
CODE get.A5
   MOVE.L   A5,-(A6)
   RTS
END-CODE MACH
```

`find.Next  ( lfa -- lfa vocab.id )` **`GLOBAL`**

This word takes the link field address of a word and attempts to find the next defined word in the same vocabulary. If it finds the next word, it returns the `lfa` of that word and an internal constant that indicates which vocabulary this word is from. If it does not find the word, usually because the passed-in word is the last in its vocabulary or the vocabulary that the word belongs to is not in the search order, the returned `lfa` is zero and the `vocab.ID` is undefined.

Remember that if you have a word, the `lfa` contains the offset to the previous word. This word uses the internals of the Mach2 dictionary structure to search each vocabulary in the `CONTEXT` search order in turn until it finds the word.

The `vocab.ID` can be used to recover the following items for each vocabulary:

To get the `LAST` word defined in a particular vocabulary, use the following code fragment:

```
MOVE.L  vocab.ID,D0      \ put the vocab.ID value in D0
MOVE.L  $-532(A0),A0     \ get the pointer to the dictionary start
MOVE.L  $32(A0,D0.W),D0  \ get the offset from the start of the
                         \ dictionary to the LAST word defined in the
                         \ vocabulary indicated by vocab.ID
LEA     (A0,D0.L),A0     \ create the lfa of the LAST word in the
                         \ vocabulary by adding the offset to the
                         \ dictionary start
MOVE.L  A0,-(A6)         \ push the LAST lfa on the stack
```

To get the lfa of the `VOCABULARY` word (`FORTH`, `MAC`, …) that corresponds to the `vocab.ID` returned by `get.NEXT`, use the following code fragment:

```
MOVE.L  vocab.ID,D0      \ put the vocab.ID value in D0
MOVE.L  $-532(A0),A0     \ get the pointer to the dictionary start
MOVE.L  $36(A0,D0.W),D0  \ get the offset from the start of the
                         \ dictionary to the VOCABULARY defining word
LEA     (A0,D0.L),A0     \ create the lfa of the VOCABULARY defining
                         \ word
MOVE.L  A0,-(A6)         \ push the VOCABULARY lfa on the stack
```

The `vocab.ID` returned by this word is one of the nibble values in `CONTEXT` multiplied by 8. Thus, if `vocab.ID = 8`, then the nibble value is one, which is the `FORTH` vocabulary. The following list documents the vocabulary nibble ID's.

0	Default vocabulary
1	FORTH
2	MAC
3	ASSEMBLER
4	SANE
5	I/O
6	TALKING
7	DEVELOPMENT

MODULE.VAR (--)

This word is used by the loadable name module code to create and compile the module defining word. Each module word is defined as a name-field word, with its pfa immediately following the segment field. A module word returns the address 10 bytes past the segment field, which is where various module parameters are stored. These parameters allow the module to be unlinked from the dictionary search order.

MODULE.VAR compiles the following in the name field:

```
DC.W    1          \ segment field
LEA     6(PC),A0   \ get addr of module parameters
```

```
MOVE.L  A0,-(A6)  \ push on stack
RTS
```

Insert.MODULE (names.size --) **GLOBAL**

Used in the following form:

```
      5000 Insert.MODULE _my.MODULE_
```

This word takes a size value, allocates and locks a handle, creates a word with the name read from the current input stream, saves the current dictionary state, and manipulates `NP` so that subsequent compilations will store the name field information in the allocated memory block. There is no error checking to determine if subsequent compilations exceed the size of the memory block. If the word is unable to allocate the requested size handle, it will `ABORT` with a message.

If the word created by `Insert.MODULE` is executed, it returns the address of the saved dictionary parameters.

`Insert.MODULE` creates the following structure in the name module.

```
(handle points to here)
DC.L  nnnn        \ original NP value before module was inserted
DC.L  nnnn        \ pointer to LAST word defined in this module
DC.L  lfa         \ lfa of module name
DC.B  n           \ count and flag byte
DC.B  'xxx'       \ variable-length module name string
DC.W  1           \ segment field
LEA     6(PC),A0  \ get addr of module parameters
MOVE.L  A0,-(A6)  \ push on stack
RTS               \ exit
DC.L  nnnn        \ handle to this module
DC.L  nnnn        \ pointer to next module in the chain, zero if none
( other words in this module follow immediately after this header)
```

restore.Name.Space (addr --) **GLOBAL**

Used in the form:

```
      _SYSEQU_ restore.Name.Space
```

where `_SYSEQU_` is a module name that was previously defined and is currently being stored in. There is no error checking to verify that this is a correct name module, or that we are currently saving names in this module.

This word returns `NP` to point to the default Name Space, but leaves the just-defined names linked into the search order.

forget.MODULE (addr --) **GLOBAL**

Used in the form:

```
      _SYSEQU_ forget.MODULE
```

This is the word used to unlink and delete a name module. Never `FORGET` a module or a word inside a module. Always use this word. Modules currently in use can be forgotten in any order. There is no restriction to forget in the same or reverse order as they were defined. The word preceding `forget.MODULE` must always be a word created by `Insert.MODULE`.

There is a gotcha when hashed searches are in effect. Because of the nature of how entries are stored, searched, and deleted in the hash table, I am currently unable to erase the entries in the hash table belonging to the name module. If hashing is enabled, and a module is forgotten, and then a word is later referenced from this module, there is a finite probability that the search algorithm will return an address that is no longer valid. Use of this address could have dire consequences. So the moral of the story is: don't attempt to use a word in the module after the module has been forgotten.

When hashing is disabled, I can always guarantee that an attempt to search for a word in a forgotten module will fail normally without any consequences.

Another good reason for not using hashing is that when you load many modules, the hashing algorithm with its 16-bit hash values starts getting too many hash hits, and seems to have problems finding the correct word. Some day I will fix this hashing feature and put in 32-bit hash values, and then this problem will go away.

Remember to unlink and forget all modules currently in use before using `NEW-SEGMENT`, `WORKSPACE`, `INSTALL`, or `(INSTALL)`. These words save the current dictionary image, and only the dictionary image, not the loaded-in modules. If modules are in use, the offsets that link into them are only valid for that run-time. The first time after using an `INSTALL`, `WORKSPACE`, or `NEW-SEGMENT`, these offsets would be invalid and the search routine would blow up.

The following standard trap modifier words were defined in the DEVELOPMENT vocabulary. These are all IMMEDIATE words. Use them immediately after a trap definition in a CODE word, such as:

```
CODE myword
   MOVE.L  D0,D0
   MOVE.L  A0,A0
           _MyTrap ,IMMED
   MOVE.L  D0,A0
   RTS
END-CODE
```

,IMMED	(set bit 9 - the immediate bit - of the trap word)
,MARKS	(set bit 9 - for diacSens = FALSE, for _CmpString)
,NEWOS	(set bit 9, clear bit 10 - for OS GetTrapAddress calls)
,CLEAR	(set bit 9 of the trap word to clear an allocated handle or pointer)
,CASE	(set bit 10 - the case-sensitive bit - for _CmpString)
,ASYNC	(set bit 10 - the asynchronous bit - for device driver calls)
,SYS	(set bit 10 to get a system heap operation)
,AUTO-POP	(set bit 10 to have the trap return pop the top return address)
,NEWTOOL	(set bit 9 and 10 - for ToolBox GetTrapAddress calls)

The following five words are compiled in the code segment, and their address can be gotten by ticking the name, *i.e.*

```
… ' str.#ifdef …  ( when interpreting )
```

Note that these words were not defined as `GLOBAL`'s, so they cannot be compiled from another segment.

```
HEADER str.#ifdef
   DC.B  6
   DC.B  '#ifdef'
.ALIGN

HEADER str.#ifndef
   DC.B  7
   DC.B  '#ifndef'
.ALIGN

HEADER str.#else
   DC.B  5
   DC.B  '#else'
.ALIGN

HEADER str.#endif
   DC.B  6
   DC.B  '#endif'
.ALIGN
```

#endif (--)

An immediate word that does nothing. Used as a placeholder in conditional compilation.

---

exec.word (name.string --)

A simple word that executes the name string found on the stack. There is no error handling for this routine, so you must have already verified that the name string is a valid name.

(#ifdef) (lfa found.flag --)

The basic word used by both `#ifdef` and `#ifndef`. If `found.flag` is TRUE, then the `LFA` should be the `LFA` of the symbol immediately following `#ifdef` or `#ifndef`. If found.flag is TRUE, `(#ifdef)` will interpret/compile all the words up to the next `#ifdef`, `#ifndef`, `#else`, or `#endif`. If `found.flag` is false, `(#ifdef)` will skip over all text up to the next `#else` or `#endif`.

#ifdef (--) **IMMEDIATE**

An `IMMEDIATE` word used in the following form:

```
#ifdef mysymbol
         .
      ( continue interpreting/compiling everything up to the
        next #ifdef, #ifndef, #else, or #endif )
         .
#else ( the #else part is optional )
         .
      ( interpret/compile this section if mysymbol is not defined )
         .
#endif
```

`#ifdef`'s can be nested. `mysymbol` can be any FORTH word defined in the dictionary and in the current search order. If the symbol exists, `#ifdef` allows interpretation of all the text following `mysymbol` up to an `#else` or `#endif`. If `mysymbol` does not exist, `#ifdef` looks for the next `#else` or `#endif` and begins interpreting there.

`#ifndef ( -- )` **IMMEDIATE**

An `IMMEDIATE` word, the opposite action of `#ifdef`. If the symbol is not defined, `#ifndef` will allow interpretation to proceed immediately following the symbol name. If the symbol is found, `#ifndef` looks for the next `#else` or `#endif` before allowing interpretation to continue.

`#else ( -- )` **IMMEDIATE**

An `IMMEDIATE` word. Properly used only within an `#ifdef … #endif` sequence. If the `#ifdef/#ifndef` test is true, all the code following an `#else` is ignored up to the corresponding `#endif`. If the `#ifdef/#ifndef` test is FALSE, all the code after the `#else` up to the `#endif` is interpreted/compiled.

`#define ( -- )` **IMMEDIATE**

An `IMMEDIATE` word used as follows:

```
#define mysymbol
```

to create a do-nothing word. Each word defined in this way uses two bytes of code space. Defined for completeness with the C-language syntax.

These next set of words set up the saving of the CURRENT and CONTEXT states. Used when compiling source from multiple files. Allows the current search order and current vocabulary to change independently from file to file. Use as follows:

```
push.VOCAB.state
ONLY MAC ALSO FORTH ALSO DEVELOPMENT DEFINITIONS
          .
       ( definitions )
          .
       ( don't leave anything on the stack )
          .
pop.VOCAB.state
```

`push.VOCAB.state ( -- CONTEXT CURRENT )` **GLOBAL MACH**

`pop.VOCAB.state ( CONTEXT CURRENT -- )` **GLOBAL MACH**

`#define _RECORDS_`

Define the symbol for Records for conditional compilations. I have included a separate file that contains the record definitions.

`$1F CONSTANT count.mask`

(masks out the name flags in the dict. header)

`$40 CONSTANT MACH.bit`

(used for getting the MACH bit setting)

LINK>SEG (link.field.address -- segment.field.address) GLOBAL

Simple word used to access the segment field in a name definition.

MCOMPILE (addr --) GLOBAL

Macro compile from an address up to an RTS. There is no error checking on this routine, so you must have enough code space when this routine is called.

is.MACH? (lfa -- flag) GLOBAL

Takes a link field address on the stack and tests for the MACHro bit in the count byte of the name string. Returns -1 if the MACHro bit is set.

is.name.field.word? (lfa -- flag) GLOBAL

Takes a link field address on the stack and tests the segment field for the existence of a Name-Field word (such as a `CONSTANT`, `HEADER`, or local variable). Returns -1 if so.

macro.compile (lfa --) GLOBAL

An internal word used by the record compiler. This word will macro compile the code for a word whose `lfa` is on the stack. If the `lfa` points to a name space word such as a local variable or a `HEADER`, it will macro-compile the word. If the word is a normal `A5`-relative variable, it will macro compile the reference. If the word is PC-relative variable created by `CodeRec` or `*CodeRec` , it will compile the PC-relative reference.

No error checking is performed inside this word for the `lfa`. Use this word carefully.

.OF. (n --) when interpreting IMMEDIATE

(--) when compiling

An `IMMEDIATE` word that performs the actual record compilation. Used in the following form:

```
.
... <offset> .OF. <record> ...
.
```

`<offset>` must be something that resolves to a constant. `<record>` must be either a direct `CODE` or `VARIABLE` record, an indirect `CODE` or `VARIABLE` record, or a local direct or indirect variable. `<record>` is read from the current input stream. Can be used inside and outside of a colon definition. When in the compiling state, `.OF.` also performs compilation optimization for `!`, `W!`, `C!`, `@`, `W@`, and `C@`. As an example a Variable record might go from

```
... var2 .OF. myRecord @ ...
```

and after simple compilation to

```
LEA         $-1234(A5),A0
MOVE.L      A0,-(A6)
MOVE.L      (A6)+,A0
MOVE.L      (A0),-(A6)
```

and `.OF.` optimizes this to

```
MOVE.L      $-1234(A5),-(A6)
```

The other cases are similar in their code optimization.

---

is.white.space?   (addr -- flag)   **GLOBAL**

Tests the character at addr to see if it is either a space or a tab.

GET.NEXT.WORD   (sep.char -- addr)   **GLOBAL**

Imitate `WORD` but remove any white space surrounding the word.

;RECORD   (--)

A do nothing word used to end the record definitions. Always used with `:RECORD`.

SizeOf((-- n)

A word used to get the size of a defined record. Used as follows:

```
... SizeOf( RecordDef ) ...
```

The folowing four words are do-nothing words used to determine what kind of record is being allocated.

VarRec	**(--)**
***VarRec**	**(--)**
CodeRec	**(--)**
***CodeRec**	**(--)**

:RECORD   (--)

This word starts the record definitions process. Used in the form:

```
:RECORD myRecordDef (optional comment )
      var1  type ( optional comment )
      var2  type ( optional comment )
      var3  type ( optional comment )
;RECORD
```

`:RECORD` creates a new record compiling word. In effect, `:RECORD` creates a word that is the equivalent of `CONSTANT` or `VARIABLE`. The word `myRecordDef` will then be used to create various instances of the record.

AEHandler.entry (-- A0 the.AE reply)   **MACH**

This word is a MACHro used to compile the glue code needed to execute an Apple Event handler that has been called. This word sets up the FORTH environment, task-switches into the appropriate task, and passes the `theAppleEvent` and `reply` parameters to the actual handler code. When the handler is installed, the programmer has the responsibility to provide the task data area address as the Event Handler `refcon`. This glue code expects the `A7` stack frame to be:

```
  (A7):  return address
 4(A7):  refcon, must be the pointer to task space
 8(A7):  reply
12(A7):  AppleEvent
16(A7):  OSErr
```

The routine is currently defined as:

```
LINK     A0,#0                \ setup a stack frame
MOVEM.L  D0-D7/A1-A4/A6,-(A7) \ save all registers
MOVE.L   8(A0),A4             \ setup the Task pointer
MOVE.L   A7,D4                \ setup the TrapStack pointer
MOVE.L   8(A4),A6             \ get the Task A6 stack
MOVEM.L  (A6)+,D5-D7/A2-A3/A7 \ we are now back in the task
MOVE.L   A0,-(A6)             \ store addr of stack frame
MOVE.L   16(A0),-(A6)         \ theAppleEvent
MOVE.L   12(A6),-(A6)         \ reply
\ from here the FORTH environment is setup and ready
\ the code here must have the following stack sequence:
\  ( A0.frame theAppleEvent reply -- A0.frame OSErr )
```

Note that the glue code stores the A0 stack frame pointer on the stack. The handler routine must not overwrite or delete this value, or horrible things will happen when the handler attempts to exit back to `AEProcessAppleEvent`.

AEHandler.exit (OSErr --) **MACH**

This routine suspends the task, saves the task environment, and switches back to the native environment provided by `AEProcessAppleEvent`. In addition, it saves the `OSErr` result to the stack frame. If your event handler is entered using `AEHandler.entry`, you must use this word as the last word in the handler definition.

The word is currently defined as:

```
MOVE.L   (A6)+,D0             \ get the OSErr
MOVE.L   (A6)+,A0             \ restore the stack frame
MOVE.W   D0,20(A0)            \ store the OSErr result

MOVEM.L  D5-D7/A2-A3/A7,-(A6) \ save the task state
MOVE.L   A6,$8(A4)            \ save off the A6 stack
MOVE.L   D4,A7                \ restore the callers stack
MOVEM.L  (A7)+,D0-D7/A1-A4/A6 \ restore all registers
UNLK     A0                   \ unlink the stack frame
RTD      #12                  \ and return to the system
```

AE: (--)

This word is an integrated version of `AEHandler.entry`. It acts just like a colon definition except that the glue code for an Apple Event Handler is compiled at the beginning of the word. Compilation then proceeds normally.

This word is currently defined as:

```
CODE AE:
      JSR     CREATE           \ create the handler
      JSR     RECURSIVE        \ hide the handler name
      SUBQ.L  #4,$-1EC(A5)     \ recover code space used by CREATE

      COMPILE AEHandler.entry \ compile the glue code

      MOVE.L  D5,(A3)+         \ push MACH2 internal constant
      MOVE.L  D6,D5            \ onto return stack
      MOVE.L  #$99887766,D6
      JMP     ]                \ start normal compilation
END-CODE IMMEDIATE
```

;AE (--)

This word is an integrated version of `AEHandler.exit`. It acts just like a semi-colon ending a compiled word except that the glue code for an Apple Event Handler is compiled at the end of the word. Compilation/interpretation then proceeds normally.

This word is currently defined as:

```
CODE ;AE
      COMPILE AEHandler.exit  \ compile the glue code
      JMP     ;               \ finish up this definition
END-CODE IMMEDIATE
```

:XDEF (-- branch marker)

This word compiles the entry point for an external definition in a code image. This word is usually the first word defined in the code image, and is used as follows

```
:XDEF my.XDEF     ( -- branch marker )
 .
. ( various code routines and local constant data definition
 .
: my.entry point ( -- )
      ( this is the word jumped to by the entry point my.XDEF )
        .
      ;

' my.entry point ;XDEF  ( branch marker address -- )
```

The `branch` and `marker` data should not be deleted or overwritten during compilation. `:XDEF` is currently defined as:

```
: XDEF:                 ( - branch marker )
      CREATE  -4 ALLOT  ( create the code image header )
      $4EFA W,          ( JMP nnn(PC) )
      0 W,              ( entry point to be filled later )
      0 ,               ( length of routine to be filled later )
      HERE 6 - 76543    ( marker )
      ;
```

;XDEF (branch marker entry --)

This word completes the external stand-alone code definition. `branch` is the address of the offset location for the jump instruction. `marker` is a unique constant used to verify that the `branch` has not been lost or overwritten, and `entry` is the address of the first FORTH word that is executed when this external code is called. See the definition of `:XDEF` for an example of how this word is used.

`;XDEF` is currently defined as:

```
: ;XDEF           {  branch  marker  entry | -- }

      marker 76543 <>  ABORT" XDEF Mismatch!"
      entry branch - branch W!
      HERE branch - 2+ branch 2+ !
      ;
```

INIT.prelude (-- pointer) (when executing) **MACH**

This word is used as the glue code to setup a FORTH environment for an `INIT` or `XDEF`. This word is typically used as the first word in the entry point word definition. Used as follows:

```
: my.INIT.entry
   INIT.prelude set.INIT.stack= 4096
   (now we can call real FORTH code )
     .
   INIT.epilog
   ;
```

INIT.prelude is currently defined to compile the following code sequence. Note that this sequence save a few registers for convenience, but does not save all.

```
CODE INIT.prelude
   MOVE.L  A3,-(A7)      \ save A3
   LINK    A6,#-2048     \ allocate a 2K FORTH stack - set by
                         \ set.INIT.stack= to user-decided value
   MOVE.L  A7,A3         \ setup local loop return stack
   MOVEM.L A0-A1,-(A7)   \ save these registers
   MOVE.L  A0,-(A6)      \ pass pointer to INIT
END-CODE
```

Set.INIT.stack= (--) **IMMEDIATE**

This is an `IMMEDIATE` word that is always used immediately following `INIT.prelude`. This word sets the size of the stack for the code compiled by `INIT.prelude`.

This word is used as follows:

```
: my.INIT.entry
   INIT.prelude set.INIT.stack= 4096
   (now we can call real FORTH code )
   .
   INIT.epilog
   ;
```

INIT.epilog (--) **MACH**

This word is used to compile the glue code to exit the FORTH environment and return to the caller. This word is typically the last definition in the `XDEF` entry point word definition.

This word currently is defined to compile the following code sequence.

```
CODE INIT.epilog
   MOVEM.L (A7)+,A0-A1      \ restore stuff
   UNLK    A6               \ unlink the A3 stack
   MOVE.L  (A7)+,A3         \ restore A3
END-CODE
```

DA.prelude (-- ioPB DCE) **MACH**

This word compiles a code sequence at the beginning of a Desk Accessory or Driver entry point that will setup a FORTH environment and pass in the I/O parameter block and the Device Control Entry. Used as follows:

```
: my.Open
```

```
   DA.prelude set.DA.Stack= 2000
      .
   ( Open code goes here )
      .
   DA.epilog
   ;
```

This word currently compiles the following code sequence:

```
CODE DA.prelude
   LINK    A6,#-2048    \ allocate a 2K FORTH stack - this number
                        \ modified by set.DA.stack=
   MOVEM.L A0-A1,-(A7)  \ save these registers
   MOVE.L  A6,A3        \ setup local loop return stack
   SUBA.W  #1792,A3     \ leave space for Mach2 FP stack - this
                        \ number modified by set.DA.stack=
   MOVE.L  A3,D7        \ setup the D7 floating point stack
   MOVE.L  A0,-(A6)     \ pass parameter block
   MOVE.L  A1,-(A6)     \ pass DCE
END-CODE
```

Note that DA.prelude allocates space for the Mach2 Forth floating point parameter stack and sets up the D7 register for this stack.

Set.DA.stack= (--) **IMMEDIATE**

This word compiles a user selected size for the `A6` parameter stack in the glue code compiled by `DA.prelude`. This word must always immediately follow `DA.prelude`.
A literal number must always follow this word in the input stream. Used as follows:

```
: my.Open
   DA.prelude set.DA.Stack= 2000
      .
   ( Open code goes here )
      .
   DA.epilog
   ;
```

DA.epilog (return.code --) **MACH**

This word compiles the glue code used to exit from the FORTH environment and return to the Mac OS environment. There must be an `OSErr` return code on the stack before this routine is called. This word is typically the last definition in the DA entry point word definition. Used as follows:

```
: my.Open
   DA.prelude set.DA.Stack= 2000
      .
   ( Open code goes here )
      .
   DA.epilog
   ;
```

This word currently is defined to compile the following code sequence.

```
CODE DA.epilog
   MOVE.L  (A6)+,D0     \ pass return code
   MOVEM.L (A7)+,A0-A1  \ restore stuff
   UNLK    A6
END-CODE
```