Chapter 16
Precompiled Modules

General Information about Modules

What *are* Modules?

Perhaps the best way to describe the MODULE facility is to describe some of the situations in previous JForth releases it was designed to improve upon...

1. In earlier JForths, all required include files (.j) had to be pre-compiled and resident in the dictionary in order to use them.
2. In earlier JForths, the assembler had to be pre-compiled and resident in the dictionary in order to use it.
3. In earlier JForths, the disassembler had to be pre-compiled and resident in the dictionary in order to use it.

In each, significant dictionary area is used up by definitions that are only occasionally required; this space is also taken in all SAVEd-FORTHs...duplicated in each one.

The MODULE facility allows us to save these sections of dictionary in disk-resident files, and provides a versatile interface to them. Virtually no space is needed in the normal development image, yet at any time, the saved binary module may easily be read and 'hooked in', providing full access to the definitions it contains. Just as easily, the module may later be 'unhooked', restoring the dictionary to its previous state.

When the 'piece of dictionary' is 'hooked in', the VLIST (or WORDS) command will display it's names in keeping with the dictionary location at which they are physically linked in.

Modules are generally only useful with large, 'execute-only' routines (such as the JForth assemblers and disassembler) and data definitions (compiled '.j' header files); these services are implemented as modules in JForth, as released.

The purpose of this document is to describe these services such that the JForth user can quickly benefit from their use. Also discussed are the few restrictions placed on source files destined to be saved as modules, if you wish to create your own.

Modules and SAVE-FORTH

A given module is compatible with all SAVE-FORTHed images that are identical at least from that module definition on down. All the standard JForth modules are automatically created when the system is generated, so as long as you have not changed any JForth system functions, the supplied set of *mod:* files should always work for you.

If you change something in the JForth system, and regenerate *com:JForth*, you simply answer *Yes* to the question dealing with module support when prompted during the rebuild process. All *mod:* files will be regenerated. Your task will then be to recompile your own application-specific JForth programs and SAVE-FORTH each one. (Alternatively, you could save a copy of the *mod:* directory with each image, then re-assign *mod:* as appropriate).

Technical Notes on Modules

1. The Module service provides a means of allocating a block of memory of a given size, and then

shifting subsequent compilation to that area (instead of the normal, JForth-resident dictionary).

2. When the program is finished compiling, compilation is returned to the resident dictionary. However, the linked-list of Forth words now 'jumps' to this non-contiguous area and follows the definitions created by the file. The last of these definitions will point again to the normal dictionary, to the word immediately preceding the word that 'jumped'.

3. Fully compiled, the external memory is then 'unhooked' from the link list and saved to a file. When again read from disk, the reverse process occurs...the existing link-list is broken and the binary module is 'hooked-in'.

The benefits realized are multiple:

* The saved utility is made available in a matter of seconds, a great improvement over the time it would take to compile it.

* The program component does not occupy any contiguous dictionary area, even when loaded.

* Saved images do not contain large, duplicate sections of program, increasing effective disk capacity.

* If the utility is never called, it doesn't use any system resources (memory, dictionary search-time, etc.).

JForth versions 1.3 and 2.0 look for all MODULES in whatever directory you have ASSIGNed the nickname 'mod:' to. "JForth:assigns" defines it as "JForth:modules". In JForth 3.0, additional flexibility is added to the GETMODULE and SEALMODULE functions, described below.

It is important to note that, other than .J structure-type files, module definitions may NOT BE DIRECTLY REFERENCED BY OTHER DEFINITIONS! In other words, they may not be directly called by other programs. The JForth compiler will not compile a reference to a routine outside of the normal dictionary, so you will be notified if you inadvertantly attempt it. Techniques for programmatically accessing module definitions are described ahead (see WILLGET and WILLHIDE).

Using the Assembler and Disassembler Modules

The assembler (JU:FORWARD-ASM, which includes the RPN assembler) and disassembler (JU:DISM) have been provided as pre-compiled modules in the default MOD: directory. Their names are MOD:ASSEM.MOD and MOD:DISASSEM.MOD, respectively.

Both have been implemented transparently such that the module is automatically loaded at the first invocation of any of a specific set of words. For example, the command: DEF SWAP will load the disassembler module if necessary, and disassemble the code for SWAP. The disassembler module will then remain linked in and available until specifically released by the programmer.

The following words will automatically invoke the assembler module:

```
CODE  ;CODE  ASM
```

The following words will automatically invoke the disassembler module:

```
DEF  DISM  RELDISM  RISM  ADISM  DISM-DONE?  DISM-WORD?
INIT-DISM  DISM-CYCLES    DISM-NAMES DISM-ORIGIN .REGNAMES?
SHOW-CYCLES
```

Using other Modules

This section describes the words used to directly manage the loading, usage, and unloading of a given module.

A particular defined MODULE will always be in one of 3 possible states. The first state exists when the JForth image is initially booted (or anytime COLD is executed)...

1. DETACHED... The module has not been read or was purged. No Amiga memory is allocated for it. If the module defines a vocabulary, it is not visible in this state.

   The function: GETMODULE <modulename> enters the next state, when something in the module is needed...

2. LOADED .... GETMODULE <modulename> will read the module from disk into a memory area that has been allocated outside of the JForth image. It will be linked into the dictionary FIND path and its definitions (including its one possible vocabulary) will be accessible.

   When the immediate need for the module is gone, it may be desirable that it not slow down the FIND process until needed again; the module may be 'put away' temporarily with the following command: HIDEMODULE <modulename>. (This is less of a consideration if the HASHed vocabulary search is in effect.)

3. HIDDEN .... HIDEMODULE <modulename> will unlink the modules definitions from the dictionary FIND path, but the module remains in the memory area that has been allocated outside of the JForth image. Its definitions (including its one possible vocabulary) are not accessible; but can be quickly made so with GETMODULE <modulename>, without re-reading from disk.

The programmer, at any time, may return to the DETACHED state, reclaiming all memory used by the module. The DETACHMODULE <modulename> is provided for this purpose.

A fourth (FORTH?) command: .MODULES is provided to display the status of all defined modules. Status is also displayed by the MAP command.

Note that all normal JForth words (FILE?, VLIST etc.) operate normally on module definitions, once GETMODULE has loaded them.

NOTE: The 3.0 versions of GETMODULE and SEALMODULE will accept *one device-type specifier* preceeding the module name (full pathnames are *not* supported). Examples of the legal JForth 3.0 syntax for these commands include:

```
SEALMODULE RAM:MyModule
GETMODULE RAM:MyModule
SEALMODULE devs:MyOtherModule
GETMODULE devs:MyOtherModule
```

Files in INCLUDES Module

The following '.j' files have been compiled into the released version of MOD:INCLUDES.MOD...

```
ji:intuition/intuition.j
ji:intuition/intuitionbase.j
ji:graphics/rastport.j
ji:graphics/text.j
ji:graphics/regions.j
ji:exec/interrupts.j
ji:exec/libraries.j
ji:exec/execbase.j
ji:libraries/dos.j
ji:libraries/dosextens.j
ji:devices/narrator.j
ji:graphics/sprite.j
ji:workbench/workbench.j
ji:workbench/startup.j
```

Creating a Custom Module

The best candidates for new modules are custom structure definitions or larger utilities such as the Assembler or Disassembler. These programs are usually INTERPRETED and EXECUTED, rather than called from other programs. Nonetheless, the ability to support a limited programmatic interface is provided.

Programs that are destined to become modules must conform to the following restrictions:

1. The compiled program may not exceed 32K. This restriction does not apply, however, if the module contains ONLY compiled '.j.' structure-based definitions. Modules of this type may be of any size.
2. No address of any component within the module may be saved within a variable, array, or any other type of storage element. This includes DEFER and GLOBAL-DEFER children or routines which will be used as vectors for such; these may NOT be defined within modules.
3. The use of ' is limited to ONLY those words that are inside the standard JForth dictionary. For example:

   ```
   : NOOP-CFA   ' NOOP  ;
   ```

   is legal within a module, because the word NOOP is in the normal dictionary. However:

   ```
   : NOOP2  ;
   : NOOP2-CFA  ' NOOP2  ;
   ```

   is incorrect since NOOP2 would be resident in the module.

4. The module may define, at most, one VOCABULARY and it must tree from the FORTH vocabulary level.
5. Do not define USER variables within a module...use normal VARIABLEs instead.

The convention in effect for defining the standard modules is to create one source file, which, when INCLUDEd, will create the module definition and .MOD file pair. For example, the file to define a module for a calculator utility and create the .MOD file would be called MAKECALC and would look something like:

```
\ Create MOD:CALCMOD.MOD from XX:CALC.F (the source file)
\

MODULE CALCMOD   \ define the module in the dictionary

4 MAKEMODULE CALCMOD   \ allocate 4K memory &
                       \ shift compiling to there
                       \ CRASH if not enough mem!

INCLUDE XX:CALCMOD.F   \ compile the source file

cr ." Writing MOD:CALCMOD.MOD" cr cr   \ nice message
SEALMODULE CALCMOD                     \ write the file,
                                       \ will be < 4K
DETACHMODULE CALCMOD                   \ cleanup
```

At this point, if a transparent interface (such as DEF and ASM use) is desired, a WILLGET command must be entered for each word that should automatically load and call the module. The format of this command is:

```
WILLGET <module-name> <function-name>
```

where: <module-name> is the same name as entered for the MODULE and

<function-name> is the name of the word in the module to be executed.

For example, if the main entry point for the above calculator program is called CALC, one more line in the file will be handy...

```
WILLGET CALCMOD CALC
```

will cause the word CALC to automatically load the module, then find and execute the word CALC within it. This is the method for gaining programmatic access to the contents of a module. Note that while even the names of words defined as VARIABLEs, CONSTANTs and STRUCTures may be specified in the WILLGET command, a VOCABULARY name cannot.

A variant of WILLGET is provided which operates identically, except that words defined with it will HIDEMODULE when they finish execution. It's usage...

```
WILLHIDE <module-name> <function-name>
```

For further examples of creating MODULEs, refer to JF:MakeASSEM, JF:MakeDISASSEM and JF:MakeINCLUDES.

If a transparent WILLGET interface to the module is not desired (or reasonable, as in the case of the INCLUDES module), the GETMODULE <modulename> command must be used to gain access to the module definitions.

WARNING: You cannot Clone a program that calls routines in a module in any way, even via a defined WILLGET interface.