Chapter 18
Amiga Libraries and Structures

The Amiga is a treasure chest full of wonderful tools: high speed color graphics, pull down menus, speech devices, powerful system libraries. This section of the manual tells you how to use JForth to open that treasure chest. We will describe how to call Amiga Library routines and how to use Amiga structures. We will then describe some of the special tools we have provided to make this easier.

Amiga Libraries - Tutorial

The Amiga system software is organized into libraries that can be used by any program. If you want to open a window, read a file, or draw a rectangle you need to call a library routine to do this. The most often used libraries are DOS, which handles files and processes, GRAPHICS, which handles drawing lines, images, text, etc., INTUITION, which is used to control windows, screens, menus and gadgets, etc., and EXEC which controls the guts of the machine.

Let's start by calling a DOS routine that waits for a given time. Before we call any library routine we must first OPEN that library. Enter:

```
DOS?
```

This will open the DOS library. If it is already open then nothing will happen. (JForth always opens the DOS and EXEC libraries when it starts up but it doesn't hurt to call this again.) Any library that is known to JForth can be opened by calling a similar word, eg. GRAPHICS? or INTUITION? .

We now need to know what arguments, or parameters, this function takes. We should look this up in the DOS Developers Manual to find out everything about this routine. But if you just want to see the argument list you can use ARGS to look them up.

```
ARGS DOS_LIB DELAY
```

ARGS will search through the "FD" files which contain information about the libraries. It prints the argument list and its offset in the library. The offset is handy for assembly language programmers. If the Amiga asks you to insert the disk "FD:" don't worry. It just means that you forgot to execute the ASSIGNS file. This file tells JForth what directory to find other files in. If you get this message, enter in the CLI window:

```
EXECUTE JFORTH:ASSIGNS
```

then hit the "Retry" button on the requestor.

ARGS will tell you that the Delay function takes one argument, the TIMEOUT value. If we look in the manual we will discover that this is expressed in 1/50ths of a second. We will also discover that it does not return a value. Routines that do not return a value are called "VOID". Now that we know what we are calling, let's write a word that calls Delay.

```
: Delay()  ( timeout -- , delay for timeout/50 seconds )
    CALLVOID DOS_LIB DELAY
;
```

Notice that we put two parentheses at the end of the word name. This is a JForth convention that indicates we are calling an Amiga library routine. The word CALLVOID word looks up the information about DELAY just like ARGS did. Then it builds the code necessary to call that routine. We can use the JForth word CALL for routines that do return a value. Now let's test our word.

```
150 DELAY()
```

This should cause a delay of about three seconds. Pretty exciting? Don't worry. Before long you will be opening windows and doing graphics.

Passing Addresses to Library Routines

JForth uses what are called relative addresses. All the Forth words like @ and !, or variables, use an offset from the base of the JForth dictionary. This is so that the addresses will be the same every time you run JForth even though the actual Amiga addresses may be different. This greatly simplifies most programming tasks. If you are passing addresses to Amiga library routines, however, you will need to convert them to absolute 68000 addresses. Luckily, this is very easy to do using >ABS which converts a relative JForth address to a 68000 absolute address. Enter:

```
' DUP .HEX  ( relative address of DUP code )
' DUP >ABS .HEX  ( actual absolute address )
```

Let's call an Amiga routine that requires some absolute addresses. Intuition has a function called CurrentTime that will set two variables to the current time. We will need to pass the absolute addresses of those two variables. Enter:

```
VARIABLE SECONDS
VARIABLE MICROS
: CurrentTime() ( addr-seconds addr-micros -- )
    >ABS SWAP >ABS SWAP  ( convert both addresses )
    CALLVOID INTUITION_LIB CURRENTTIME
;
: PRINT.TIME  ( -- seconds micros )
    SECONDS MICROS CurrentTime()
    ." Seconds = " SECONDS @ .
    ." , Micros = " MICROS @ . CR
;
INTUITION?  ( open library! )
PRINT.TIME  ( test it )
```

Notice how the same word that calls the library routine does the address conversion. That way you only have to pass the normal relative addresses. Whenever possible, try to only pass relative addresses between words.

Now that you know how this works, I will show you a shortcut. If you enter

```
ARGS INTUITION_LIB CURRENTTIME
```

you will see that the arguments are passed in A0 and A1. These are Address registers, as opposed to Data registers like D0 or D1. We can use a special form of CALL that will automatically convert any argument that goes into an address register. This works fine with the GRAPHICS, EXEC, INTUITION and most other libraries. Unfortunately DOS uses data registers to pass addresses (!) so we have to do the >ABS ourselves with DOS.

Here is another way of defining CurrentTime() using this feature:

```
: CurrentTime() ( addr-seconds addr-micros -- )
    CALLVOID>ABS INTUITION_LIB CURRENTTIME   ;
```

This will work even with a mix of address and data arguments. Look at:

```
ARGS GRAPHICS_LIB  DRAW
```

We have been writing simple routines whose only function is to call an Amiga Library routine. You could, of course, use CALL in any colon definition no matter how large. We recommend, however, that you use these small "glue" routines to save space and produce more modular code.

Getting Values from Library Routines

Most Amiga Library routines return a value to the calling program. If, for example, you open a window, the Intuition Library will return to you a pointer to a window structure. You can use this pointer to get information about the window or to perform graphics operations on this window. This

pointer will be returned in absolute mode. You should convert this to relative mode before passing it on.

You could use >REL to preform this conversion. A problem can occur, however, if Intuition passes you back a NULL, or zero, pointer. This can happen if Intuition fails to open a window. If you used >REL the NULL would get converted to a nonzero value. You would then be unable to check the pointer to see if it is valid. In this case, therefore, we should use IF>REL for our conversion. It will only convert an address if it is nonzero. Here is how you would call the OpenWindow routine.

```
: OpenWindow() ( newwindow -- window )
    CALL>ABS INTUITION_LIB OPENWINDOW
    IF>REL  ( convert window address )   ;
```

(There is an example of the use of this routine in the JForth Graphics toolbox that we will study later.)

Accessing the Amiga Libraries - Reference

With the installed JForth word set, all the standard Amiga 2.0 libraries are accessible, along with a few more:

```
arp                 asl                 battmem
battclock           clist               commodities
console             cstrings            disk
diskfont            dos exec            expansion
gadtools            graphics            icon
iffparse            input               intuition
keymap              layers              mathffp
mathieeedoubtrans   mathieeesingbas     mathieeesingtrans
mathtrans           misc                potgo
ramdrive            rexxsyslib          romboot
timer               translator          utility
workbench
```

Opening Libraries

The JForth-recommended method for opening a library is to state the name immediately followed by a question-mark. For example, at the startup of a program which will call both the graphics and intuition libraries, the programmer need only state 'GRAPHICS?' and 'INTUITION?'. These words will open the library only once, regardless of how many times they are called.

```
: MY-PROGRAM  ( -- )
    GRAPHICS?              \ opens graphics.library if needed
    ...my-graphics-words... \ run application
    -GRAPHICS    ;         \ close the graphics.library
```

JForth opens two of these libraries for you, EXEC and DOS. These libraries are ALWAYS open when the JForth development environment is up. While not an error, it is a null operation to execute either EXEC? , -EXEC, DOS? or -DOS. They are provided only for compatibility reasons and the surrounding JForth environment will actually manage them.

Your program may specify a particular version of a library by storing the desired version number in the user variable called LIBVERSION. Otherwise, JForth will not care which version is found; any will suffice. Note that, if version number IS important to you, it will have to be stated just prior to each library open operation. This is because the variable LIBVERSION is automatically set to zero after each library is opened.

```
36 LIBVERSION !  GRAPHICS?
36 LIBVERSION !  INTUITION?
```

The words that open libraries, such as GRAPHICS?, provide two types of behavior on an error; either to execute QUIT or not (in a CLONEd application, QUIT will exit the program) or not. See the section *Library Open Verification* ahead for details.

Note: When a library is opened, it's **absolute** address is stored in the word called "name_LIB". For example, if you opened the Intuition library, the Intuition library pointer would be stored in INTUITION_LIB . This pointer points to INTUITIONBASE. To get the address of INTUITIONBASE, enter:

```
INTUITION?
INTUITION_LIB @ >REL  ( get relative address of INTUITIONBASE)
```

The same technique can be used to get EXECBASE or other useful library bases.

Closing Libraries.

Closing libraries is similarly easy: library name preceded by a 'minus' sign. The above example application, at termination, can cleanup the two libraries by executing:

```
-GRAPHICS
-INTUITION
```

Note that the '-NAME' words unconditionally close the library, They should, therefore, only be used when an application terminates.

The 'exec' and 'dos' libs cannot be affected by these words. JForth maintains these libraries; the words '-DOS' and '-EXEC' are provided for compatibility reasons and have no code.

Calling Amiga Libraries.

Accessing a library is made easy by JForths 'call compiler'. The format for a system call is as follows:

```
arg1 arg2 ... arg(N)   CALL   libname_LIB  FunctionName

ARGs = the arguments in the same order as in the Amiga
       technical reference manuals.
'Call'         = invokes the 'call-compiler'
'libname_LIB' = name of library followed by '_LIB'
                (lower case ok)
FunctionName   = standard, full-text, Amiga name for the
                 function.
```

For example, the dos function 'Seek' is listed in the AmigaDOS Developer's Manual as follows:

```
Seek( file, position, mode )
```

In JForth, a typical SEEK to the beginning of a file might appear:

```
: REWIND.IT  ( -- prev-position , rewind my file )
  ( file     position        mode   )
  MYFILE @       0      OFFSET_BEGINNING call dos_lib seek  ;
```

CALL builds all the necessary code to access the library. It looks up in the "FD:" files to find out which parameters are passed, then calculates the offset of the routine in the library. It then builds the code necessary to pull the parameters off of the stack, place them in the proper 68000 registers and call the routine. CALL may only be used during compilation, for example, inside of a colon definition. It cannot be used interactively from the keyboard.

CALL will normally return the contents of register D0 on the stack. This is the way values are normally returned from the Amiga libraries. Some library functions, however, do not return a value, and some return two. Others even return special values in the Status Register. JForth provides words that instruct CALL to compile slightly different code for these situations.

Library Open Verification

VERIFY-LIBS (-- var-addr , IMPORTANT !!!!!)

If this variable is TRUE (which it is, by default), JForth will compile a check before **every** new library call to make sure that the library is open. This can save you from crashing when you are first debugging a program and might forget to open a library. Once you have your program debugged and are opening the necessary libraries properly, you should set this variable to FALSE and recompile. Since your program will no longer be making these redundant checks, it will run **faster**. Note: CLONE will automatically remove these checks when generating a target image.

```
VERIFY-LIBS OFF   ( then recompile! )
```

LIB_QUIT (-- var-addr , JForth 3.0 and later)

By default, LIB_QUIT is set TRUE. This will cause QUIT to be executed if an error occurs inside a library-opening word such as GRAPHICS?. (Note that within a CLONEd standalone program, QUIT will cause the application to exit).

To prevent this, the program should set LIB_QUIT to false *before <u>every call</u> to a XXX? word which will open a library*. For example...

```
LIB_QUIT off   INTUITION?
LIB_QUIT off   GRAPHICS?
```

To check if the library opened successfully, the program may check the contents of the XXX_LIB variable for a non-zero value. For example...

```
LIB_QUIT off   ICON?
ICON_LIB @  ( -- lib-pointer / 0 )  \ will be 0 if error occured
```

CALL modifiers

When these words are used before CALL, they will affect the way that the Amiga Library call is compiled. These modifiers only affect the next CALL then they are turned off. In actuality these words are seldom used. We have provided shortcut words that are described in the next section. These modifiers are mainly used if you need several together, for example, RET:DOUBLE and RET:SR.

```
: SAMPLECALL  ( parameters... --  double sr )
    RET:DOUBLE  RET:SR  CALL MATHIEEEDOUBBAS_LIB IEEEDPMul  ;
```

AREGS>ABS (-- , tell CALL to convert addresses with >ABS)

When CALL looks up the parameters in the FD files, it knows which ones are addresses because they get put into address registers. CALL can therefore automatically convert these addresses from JForth relative addressing to Amiga absolute addressing. Warning! DOS passes address in data registers so use >ABS explicitly with DOS calls. NULL addresses are preserved.

RET:DOUBLE (-- , tell CALL to return both D0 and D1)

This is used when you want a 64 bit result. This is used extensively with the double precision floating point libraries.

RET:SR (-- , return Condition Codes from Status Register)

Some of the floating point routines pass back overflow and other flags in the Status Register. This allows you to get those codes. Warning! Do not bypass this facility by calling MOVE-FROM-SR because this instruction is not legal on some 680x0 processors.

RET:VOID (-- , tell CALL not to return a value)

This will instruct the CALL operator to NOT compile the code to put the return value on the stack.

CALL shortcuts

We have provided some special versions of CALL that incorporate these modifiers. These are used more often than using the modifiers directly.

CALL (...parameters... <name_LIB> <function> -- result)

Normal call without modifiers. Any addresses passed should have already been converted using >ABS.

CALL>ABS (...parameters... <name_LIB> <function> -- result)

Just like CALL except parameters destined for address registers are converted automatically. NULL addresses are preserved.

CALLVOID (...parameters... <name_LIB> <function> --)

Don't return anything.

CALLVOID>ABS (...parameters... <name_LIB> <function> --)

Combination of CALL>ABS and CALLVOID .

DCALL (...parameters... <name_LIB> <function> -- double)

The DOUBLE returned occupies 2 stack cells.

As a convention, words that do little more than call an Amiga library routine should have a () suffix placed on them. These Forth words should have the same calling sequence as the Library routine. For example:

```
: DELAY() ( #ticks -- , delay #ticks 1/50ths second)
    CALLVOID DOS_LIB Delay
;
100 DELAY()
```

To find out what parameters a routine expects, you can use the ARGS facility. ARGS will search the '.fd' files and print the line that contains the parameter description.

```
ARGS dos_lib seek   ( prints parameters for seek )
```

Adding Libraries.

The following section applies only if you wish to call a custom library, one not supplied with the original Amiga Workbench release.

The library must be a normal Amiga Library that can be opened using OpenLibrary(). Examples are the ARP library, the Live video digitizer library, Bill Barton's MIDI library, and the AREXX library. (You can also create you own libraries. The details for this are in the Amiga technical literature, and beyond the scope of this discussion.)

Once you've installed your library in the Amiga OS by placing it your LIBS: directory, 'teaching' JForth about it is easy, as illustrated in the following steps, which define a new library called 'GOODIES'. Note that there is NO space between the ':' and the 'L' in this next line.

```
:LIBRARY GOODIES  ( this creates GOODIES_NAME )
                  ( and GOODIES_LIB, used in  )
                  ( the following steps ...   )

: GOODIES?    GOODIES_NAME  GOODIES_LIB  LIB?  ;

: -GOODIES    GOODIES_LIB   -LIB   ;
```

These are the operatives you will use to open your library, in the same manner as that described above

under 'Opening Libraries'.

Finally, you will have to construct a 'function declaration' file for your new library that JForth can read and find the calling parameters. This is an ASCII file that can be created with any popular editor and should be placed in the JForth logical volume 'FD:'. Your new file should reside there, and should be similar in format to those which are Amiga-defined; use them as a model, but note the following:

JForth does not require all of the info contained in a standard '.fd' file, so you needn't bother to build the entire text; the only required portions are:

```
##bias xx
Function1(arg1name,arg2name)(arg1reg/arg2reg)
Function2(arg1name,arg2name,arg3name)(arg1reg/arg2reg/arg3reg)
```

The xx above equals the offset from library base for 1st call in list.

For example...the 1st 3 required lines in the DOS_LIB.FD file are:

```
##bias 30
Open(name,access-mode)(D1/D2)
Close(file)(D1)
```

If you were to type that file, you would see that there are more lines present; but these are all that JForth requires.

Once you have compiled the ':Library' statement (along with the 2 opening and closing operators) AND installed a proper '.fd' file (in the logical volume fd:) JForth will allow you to reference that library and its routines by name.

Amiga 'C' Structure Interface

Structures in the Amiga

The Amiga uses "structures" to describe things like windows, screens, icons, fonts, bitmaps, tasks, etc. A structure contains information about these things like width, color, type, etc. All of this information is collected in one area that can be referred to by a single address. Many of the important Amiga routines pass these addresses as a way of referring to windows, menus, etc. Accessing the features of the Amiga requires you to be able to set and retrieve values in these structures. JForth provides tools for accessing these Amiga structures and for defining new ones of your own design.

Loading Structure Definitions from ".j" Files

The data structures that the system uses are typically defined in a set of 'C' include files whose names end in ".h". These files contain templates that describe how the data in a structure is arranged. These files also contain the definitions of named constants. A 'C' program that wants to reference these data structures includes the appropriate ".h" files. For JForth programmers, a set of equivalent files has been created for inclusion in JForth programs. These files have names that end in ".j". They reside in the logical volume "JI:". To load the structures needed for the Serial device, you would enter:

```
INCLUDE JI:DEVICES/SERIAL.J
```

Loading Structure Definitions from Precompiled Modules

Most of the structures that you will need have been precompiled for fast access. They are stored in a module file called MOD:INCLUDES.MOD on the JForth disk. To access these files you must first define the MOD: volume by executing the JForth:ASSIGNS file. Please do this, if you haven't already,by entering in the CLI window:

```
EXECUTE JFORTH:ASSIGNS
```

Now you can enter, in JForth:

```
GETMODULE INCLUDES
```

This will link the structure definitions from the include files to your dictionary. Please see the section on Modules for more information on how this works. You may find that the INCLUDES module does not have a structure that you need. You can then include it as above, add it to the INCLUDES module or make your module as described in the Modules chapter.

Using Structures

Now that the structure definitions are loaded we can make copies from the template. To create one of these structures that you have defined, enter the name of the structure type followed by the name of the new copy. Enter:

```
NewWindow MYNW
MYNW .  ( print address of MYNW for fun )
```

For those familiar with 'C' the first line is equivalent to:

```
STRUCT NewWindow MYNW;
```

This creates a NewWindow structure that is used to describe how you want Intuition to open a new window. To see how this member was defined and the names of its members enter:

```
FILE? NEWWINDOW ( then hit 'y' when prompted )
```

If you want a window that is 300 pixels wide you need to set the WIDTH member of this structure. JForth uses the naming conventions from the Assembler ".i" files so we refer to this member as NW_WIDTH. Enter:

```
300 MYNW S! NW_WIDTH
```

The word S! looks up the offset for NW_WIDTH in the structure and adds it to the address on the stack. It then looks up the size of the NW_WIDTH member and uses the equivalent of ! W! or C! to store the value to that address. By using this syntax we are able to greatly optimize the referencing of structures. (Hackers: Try using S! in a small colon definition then use DEF to examine the code.) We can check that we set the value correctly by entering:

```
MYNW S@ NW_WIDTH .  ( should print 300 )
```

The above two lines in 'C' would be:

```
MYNW.width = 300;
printf ("%d", MYNW.width);
```

Note: In 'C' the case is critical so we cannot use upper case like we normally do for these examples. For actual JForth code we normally use lower case too.

Sometimes you may want to find the size of a structure so that you can allocate memory for it dynamically. To find the size of a structure use the SIZEOF() word or use ALLOCSTRUCT.

```
MEMF_CLEAR
SIZEOF() NEWWINDOW  ALLOCBLOCK  ( allocate structure )
```

Making an Array of Structures

Sometimes it is desirable to have an array of structures. Suppose we want to create 10 gadgets. We could define 10 of them with individual names but it might be more convenient to define an array of gadgets and address them by index. The word ARRAYOF will do that for us.

```
10 ARRAYOF GADGET  MY-GADGETS
3 MY-GADGET . \ print address of gadget #3
```

Referencing Substructures

Sometimes structures contain other structures as members. To access the substructure you will need its address. Once you have its address you can use S! and S@ as before. You can find the address of any member of a structure by using the .. word. An Intuition Screen, for example, contains its own RastPort. Let us assume we have a variable called SCREEN-PTR that contains the relative address of a Screen. To fetch the value of the foreground pen in its rastport we could use the following:

```
SCREEN-PTR @  ( -- address-of-Screen )
.. SC_RASTPORT ( -- address-of-Screen's-RastPort )
S@ RP_FGPEN . ( print pen value )
```

In 'C' that would be:

```
printf("%d", SCREEN_PTR->RastPort.FgPen);
```

Often structures will contain pointers to other structures. If it is an Amiga structure, then the pointer will be the absolute address of that other structure. A Window, for example, contains a pointer to a RastPort. Assume we have a Window's relative address in the variable WINDOW-PTR. Let's fetch its RastPort's drawing mode.

```
WINDOW-PTR @ ( -- address-of-Window )
S@ WD_RASTPORT ( fetch Rastport's Address
( automatically convert to relative )
S@ RP_DRAWMODE
```

Note that S@ and S! are intelligent about how they handle values. If a structure member is defined as APTR, an address being stored into that member using S! will be converted to absolute using the equivalent of IF>ABS.

Accessing Array Members in Structures

Some structures have members that are arrays of values. A BitMap structure, for example, contains an array of pointers for up to 8 individual bit planes. To reference these arrays, use .. to get the base address of the array, then add the offset required to get to the particular member of the array. To do this you will need to know the width of the values in the array. Suppose we want to fetch the address of the third bit plane in a BitMap called MYBM. We could use the following:

```
MYBM .. BM_PLANES  ( address of 0th bitplane )
2 CELLS +  ( offset to 3rd bit plane ) @
```

Examining Structures with DST

When you are working with structures, it is handy to be able to see all of the values in it at once. JForth provides a tool that makes this easy. Enter:

```
INCLUDE? DST JU:DUMP_STRUCT
```

This is a handy debugging tool. Enter:

```
NEWWINDOW MYNW  ( unless already defined )
120 MYNW S! NW_WIDTH
MYNW DST NEWWINDOW
```

DST will use the structure template whose name follows to dump the contents of a structure whose address is on the stack. The first column is the values of the members. The middle column tells you the width of the member in bytes and whether it is Signed or Unsigned. If a member is signed, then 16 and 8 bit members will be sign extended into a 32 bit number when placed on the stack. Try putting the last line above in a word called DMYNW then experiment with setting members of MYNW then dumping them out.

Defining Your Own Structures

JForth provides tools for you to define your own structure. The syntax is very similar to 'C'. A possible 'C' structure and the corresponding JForth structure are shown below to give you an idea of how they relate.

```
/* A 'C' structure. */
struct datrec   {
    ushort howmany, *sval_ptr;
    long   bigval;
```

```
    struct list *alist;
    struct rastport myrastport;
    aptr   some_mem;
    short  table[32];
};
```

In JForth, we should add a prefix to make the member names unique. Let's use "DR_". The same structure in JForth would be defined as:

```
:STRUCT DATREC     ( Start defining a structure )
    USHORT DR_HOWMANY
    APTR   DR_SVAL_PTR ( pointer to a SHORT )
    LONG   DR_BIGVAL
    APTR   DR_ALIST        ( only a pointer )
( member is a complete structure )
    STRUCT RASTPORT DR_MYRASTPORT
    APTR   DR_SOMEMEM
    32 2 * BYTES DR_TABLE    ( make room for array )
;STRUCT     ( Terminate definition )
```

For more examples of how to define your own structures, look at the ".h" files in JI: and compare them to the 'C' includes.

Structure Glossary

The words to support the use of structures are in JU:C_STRUCT and JU:MEMBER. They are loaded as part of the normal JForth image.

Structure Accessing Words

By using S@ and S! instead of ..@ and ..!, you can almost completely ignore relative versus absolute addressing. It will be taken care of for you automatically.

.. (struct-addr <member-name> -- member-addr)

Calculate the address of a structure member by adding an offset to the base of the structure.

..@ (struct-addr <member-name> -- value)

Fetch the value stored in a structure member. This will automatically use the appropriate @ word for that member. Here is a table of the equivalent @ operator used for various member types:

```
BYTE uses C@
SHORT uses W@
LONG uses @
RPTR uses @
APTR uses @    ( see S@ )
```

The member must be either 1,2 or 4 bytes wide for this to work. For members that are bigger than 4 bytes, eg. an array, use a combination of " .. " and the normal @ and ! words. With this system, you no longer have to worry about the size of your member, just how you use it! If the member is defined as signed BYTE or SHORT then it will be sign extended to 32 bits. This allows you to store negative numbers in 8 or 16 bit members. See B->S in the main glossary for more information about sign extension.

..! (value struct_addr <member-name> --)

Store value into structure member. The word ! , W! , or C! will be used depending on the width of the member as in ..@.

S@ (struct-addr <member-name> -- value)

Equivalent to ..@ except APTR members are converted to relative. The following two lines are thus functionally equivalent:

```
MNW ..@ NW_Title  IF>REL
MNW S@ NW_Title
```

S! (value struct_addr <member-name> --)

Equivalent to ..! except APTR members are converted to absolute. The following two lines are thus functionally equivalent:

```
0" Plots" IF>ABS MNW ..! NW_Title
0" Plots" MNW S@ NW_Title
```

SIGNED-MEMBERS (-- addr , variable to control compilation)

If this flag is TRUE (the default) then when ..@ is compiled it will distinguish between SIGNED and UNSIGNED members. Version 1.2 treated all members as UNSIGNED. If you are having a compatibility problem with 1.2 involving structures, try setting this variable to FALSE and recompile your application.

Structure Defining Words

:STRUCT (<name> -- , Start defining a structure.)

;STRUCT (-- , Terminate a structure definition.)

APTR (<name> -- , Amiga absolute pointer)

This member must be an absolute address when used by the Amiga. If you use S@ and S!, then addresses will be automatically be converted between relative(JForth) and absolute(Amiga) as needed.

ARRAYOF (n <structure> <name> --)

Allocate an array in the dictionary with the given name that has room for N of the specified structures. The name will take an index and return an address just like ARRAY.

```
20 ARRAYOF  GADGET  MY-GADGETS
3  MY-GADGET . \ print address of 3# gadget
```

ALLOCSTRUCT (<structure> -- addr-structure | 0)

Dynamically allocate a structure from memory. This address must eventually be freed using FREEBLOCK.

BYTE (<name> -- , Define a SIGNED 8 bit member)

You do not have to worry about word alignment of subsequent members. SHORT and LONG will automatically place themselves at a word boundary after a BYTE member. This is the way 'C' does it. In assembly you have to put in dummy bytes sometimes to avoid address errors.

BYTES (#bytes <name> -- , define multibyte member)

This is used for defining the other member types. LONG is defined as " 4 BYTES " . Arrays can be put in a structure by multiplying the width of the array units by the number of units. To make a structure member that is 10 LONG words, use:

```
10 4 * BYTES MY_LARRAY
```

LONG (<name> -- , define a 32 bit member)

RPTR (<name> -- , define a relative pointer)

A pointer member that contains a JForth relative address. This will bot be converted by S@ or S!.

SHORT (<name> -- , define a SIGNED 16 bit member)

UBYTE (<name> -- , define an UNSIGNED 8 bit member)

USHORT (<name> -- , define an UNSIGNED 16 bit member)

STRUCT (<struct-type> <name> , structure as member)

Define a structure member that is another type of structure. JForth will look up how big that structure is and make room for the right number of bytes.

Member UNIONs

A 'C' structure sometimes has several members that occupy the same memory space. These members are said to be part of a UNION. These are useful when you want to use the same part of a structure for different purposes. Let's suppose that you are passing a structure that contains a type field and then data that varies with the type. With one type, you want to pass an X,Y pair as SHORT values. With the other type you want to pass a single 32 bit address. You could do this by creating a structure like the following:

```
\ Create flexible structure.
:STRUCT FLEXDAT
    SHORT DATA_TYPE   \ 0 for X,Y, 1 for PTR
    UNION{           ( Start union )
       SHORT XPOS
       SHORT YPOS
    }UNION{
\ Has same offset as xpos, but is 32 bits wide.
       APTR DATA_PTR
    }UNION
    LONG MORE_DATA
;STRUCT

: REPORT.DATA  ( addr-flexdat -- , Report appropriate data. )
    DUP S@ DATA_TYPE 0 = ( check type )
    IF DUP S@ XPOS ." X = " .
       DUP S@ YPOS ." , Y = " . cr
    ELSE DUP S@ DATA_PTR    ( get pointer to data )
         @ ( get actual data ) ." Data = " . cr
    THEN
    S@ MORE_DATA ." more data = " . cr
;
FLEXDAT FD-1
HEX 01230092 FD-1 ..! DATA_PTR
0 FD-1 ..! DATA_TYPE
REPORT.DATA
1 FD-1 ..! DATA_TYPE
REPORT.DATA
```

In the preceding example, XPOS and YPOS occupy the same position in the structure as DATA_PTR does. If the parts of a union are not the same size, then the size of the largest part will be used. Subsequently defined members will be after that largest part. The following words must always be used in the given order:

UNION{ (-- old-offset new-offset)

Start first half of a union.

}UNION{ (old-offset new-offset -- old-offset max-offset)

Mark next part of union. Reset offset in structure so that subsequently defined members will overlap previously defined members.

}UNION (offset2 --)

Terminates union.

You don't have to worry about these offsets. They are used for communication between the UNION words and can be ignored. Just be careful that you don't put other stuff on the stack that might interfere with these.

Addressing Considerations - Important!!!

JForth uses addresses that are relative to the base of the JForth kernel. This greatly simplifies the usage of JForth because dictionary addresses don't change between successive runs for JForth. One advantage is that you can store JForth dictionary addresses in variables, do a SAVE-FORTH, rerun your code and those addresses are still valid.

The Amiga, however, must use absolute addresses to perform its work. This implies that you may only pass absolute addresses to its library routines. You must also use absolute addresses when setting a pointer in a structure that the Amiga will use. Word like CALL>ABS , S@ and S! help keep track of when these conversions are needed and do them for you. It is rare, therefore, to have to worry about this issue. It is important, however, to understand it so that you can handle unusual situations.

Suppose that you want to place a pointer to a null terminated string inside a NewWindow structure (which, of course, is being prepared for AmigaDOS to process). The word 0" will return a relative address. You must convert this to absolute by using >ABS before placing it in the structure. (S! automatically converts the relative address to absolute; if we use ..! to write the value, we will have to manually convert it using >ABS).

Some Amiga routines use the NULL value for an address to indicate an error, or a special condition. If you use >ABS or >REL on a NULL it is no longer NULL . For these cases you should use IF>REL and IF>ABS to preserve NULL . These are simply defined as:

: IF>REL DUP IF >REL THEN ;

Here is an example of using >ABS and IF>REL

```
INCLUDE? NEWWINDOW.SETUP JU:AMIGA_GRAPH
\ Make an instance of a NewWindow structure.
NEWWINDOW IDEALWINDOW
IDEALWINDOW NEWWINDOW.SETUP  ( Set default values )

: NAME&OPEN ( -- window , Name the window and open it. )
    0" WORK WINDOW"  >ABS   ( convert address )
    IDEALWINDOW ..! NW_TITLE  ( store in structure )
    IDEALWINDOW >ABS    ( Convert for Amiga call )
    CALL INTUITION_LIB OPENWINDOW
\ Convert absolute window address to relative for JForth.
    IF>REL  ( preserve NULL )    ;

: CHECK.WINDOW ( window -- , Check window pointer <> NULL )
    NOT IF ." Window not opened!!" ABORT THEN   ;
```

```
NAME&OPEN           \ Get relative address of open window
DUP CHECK.WINDOW
  ..@ WD_RPORT      \ Fetch absolute address of RastPort
  IF>REL            \ convert to relative
  ..@ rp_FgPen .    \ print pen color
```

The absolute address of the RastPort can be passed directly to Amiga graphics routines. If you want to access members of this structure using JForth, you must first convert its address to relative.

Note that in the above example we could have almost entirely avoided having to consider absolute versus relative by using S! and CALL>ABS. I say “almost” because we would still have to convert any address returned by the Amiga using IF>REL just like at the end of NAME&OPEN.

As a general guideline, when passing addresses between JForth words, pass RELATIVE addresses. Do any required conversion to or from absolute, inside your words, before interfacing with the Amiga.

H2J - Convert "xx.h" to "xx.j"

H2J is handy if you want to interface to an Amiga Library that has an associated ".h" file. An example might be the A-Squared Live library or the ARP library. The ".h" file will contain the definitions of constants and structures to be used with the Library. To use the Library from JForth you will need a ".j" file containing JForth style structure and constant definitions.

When we developed JForth 1.2, we needed something that would convert the Amiga include files. Thus H2J was born.

You can either use the Cloned version of H2J or compile it and use it directly from JForth. To compile H2J, enter:

```
INCLUDE JA:H2J.f
```

This will load ODE and whatever else it needs.

H2J takes two filenames, an input and an output filename. You may use full pathnames.

```
H2J infile outfile
```

To convert newlib.h to JForth style, enter:

```
H2J newlib.h newlib.j
```

H2J will prompt you at various times for one of two things. When it encounters a new structure definition, it will ask you to enter a prefix to add to the member names to make them unique. Like Assembly, Forth requires you to use unique names for the structure members. For a Window structure, for example, we use "wd_". If the structure members already has a prefix, just hit return.

H2J will also ask you to verify it's conversion if it encounters an unusually tricky line. It is usually correct so unless you know it is wrong, just hit return. If it is wrong, type in the line the way it should be.

Don't panic if some little thing goes wrong. You can always go back and edit the file. We find that H2J will convert about 80% of the files completely. The other 20% will require minor tweaking.