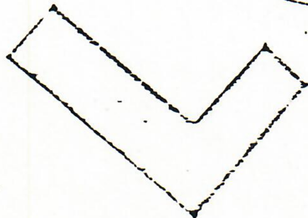
Microsoft Multitasking MS-DOS Product Specification

DYNAMIC LINKING

Microsoft Corporation

October 30, 1984

LOTUS



ABSTRACT

1. Introduction

This document gives a brief overview of the function and use of an MS-DOS 4.0 dynamic linking facility. This document is one of a series of related documents. They are:

- *Microsoft Multitasking MS-DOS Product Specification OVERVIEW*
- *Microsoft Multitasking MS-DOS Product Specification DEVICE DRIVERS*
- *Microsoft Multitasking MS-DOS Product Specification SYSTEM CALLS*
- *286 and 8086 Compatibility*
- *Microsoft Multitasking MS-DOS Product Specification INTRODUCTION*
- *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT*
- *Microsoft Multitasking MS-DOS Product Specification DYNAMIC LINKING*
- *Microsoft Multitasking MS-DOS Product Specification SESSION MANAGER*

Dynamic linking (or "delayed binding") is very straightforward; a program contains long calls to a variety of procedures. These procedures are not included in the .EXE file when the program is initially linked; instead the names of the procedures (and other identifying information) are built into the .EXE file. When the program is to be run, the operating system notes the references to the dynamic procedures, loads them into memory, and finishes linking the callers to the callees. This is a deceptively simple process that yields some very powerful gains as described below.

When reading this document, it should be kept in mind that the dynamic linking mechanism is coupled with the memory segment-management facility that provides some extra features:

Demand Loading

Depending upon the program's preference and the load media (hard disk or floppy), some or all segments can be loaded only on demand. This applies to dynamically linked routines as well. The results are obvious: greatly improved memory utilization as the code segments for error handling, and features that are not used are never loaded. Simply put, there will be segment-based virtual memory.

Code Sharing

It is common to have multiple copies of a procedure or application. Pure code segments are automatically and transparently shared.

Shared Programs

Many dynamically linked services want a fresh "instance" for each calling process. This is automatic and transparent. Shared routines can also allocate "global" segments that are accessible by each instance of the routine. ISAM is a good example; it wants global segments to hold data and index cache sectors.

2. Benefits of Dynamic Linking

Rather than get real "blue sky" about the desirability of powerful and extendible mechanisms, a partial list of concrete benefits obtained from dynamic linking is presented.

2.1. Flexibility

Software is highly inflexible by its nature. An individual program can only do what it is coded to do. It cannot learn, adapt, or change. To get around this, architects define standard interfaces to act as a kind of fire wall: the software on each side of the interface becomes independent of the software on the other side, as long as the interface conventions are maintained. The individual packages are still inflexible, but the system made up of the packages becomes more and more flexible as individual packages are changed and upgraded.

MS-DOS contains three such interfaces today: the device driver interface, the system call interface, and the INT vector interface. These are what allow programs to be device independent, and what allow MS-DOS 1.1 programs to run under MS-DOS 2.0.

Dynamic linking is another such interface, one of tremendous power. The procedure call is the interface. The procedure or package name, the arguments and the calling conventions are all that is visible across the dynamic link. This interface is much more powerful than the ones discussed above because of its great flexibility, large name space, and high efficiency. As an example, dynamic linking of the C language run-time library will allow the production of C programs that are binary compatible between MS-DOS and XENIX; a call to **printf()** is a call to **printf()**, regardless of the form of the system calls issued by that procedure.

The system call interface gave us compatibility between earlier versions of the MS-DOS; dynamic linking gives us compatibility between entire operating systems. Sometimes a system call cannot be upgraded without changing the interface. In this case a new system call is defined; the old one is kept for compatibility. Dynamic linking allows exactly the same process, with the advantage that if there *are* no old programs that currently need the old version, then that code need not be resident in memory.

Backward compatibility is an important feature; ISVs are reluctant to release products with new features if those features cannot be supported under earlier MS-DOS releases. Dynamic linking gives us the capability of providing backward support by emulating the new features in the dynamically linked module. Dynamic linking is more powerful than this; it removes the boundary between the resident operating system and the machine's software operating environment. A program invoking a dynamic link has no idea if it is invoking a subroutine function, a kernel function, or even a cross-network service. The phrase: "Dynamic linking can be used to *emulate* the new feature on older operating systems" is actually short-sighted: what has been done is to *implement* the new feature on the old operating system, field upgrading it.

In summary, dynamic linking allows us to put the critical parts of a program (such as its operating system interface) behind interfaces. The programs can then be transparently modified in the field by substituting various implementations of those interfaces. Transportability between machines, environments, release versions and even operating systems is enhanced without "paying" for features not being used.

2.2. Interrupt Vectors

Even with today's rudimentary tools, programmers are endeavoring to share standardized system services. For example, it is common for a program to "terminate and stay resident," setting up entry points by setting interrupt vectors. The limited "name space" for interrupt vectors is causing much conflict and difficulty.

Dynamic linking provides this desirable service (demand-loaded virtual segments are far better than terminate-and-stay-resident segments), and with a much more general name space. There can be multiple name-spaces (by specifying multiple library directories) so that there can be special versions of libraries to provide compatibility support.

2.3. System Flexibility

Note that the operating system (actually, the dynamic link package that is itself partially dynamically linked) arranges the physical hookup between the caller and the callee. The callee need not even be on the same machine; it might be across a network. Alternatively, the called procedure might run on another processor, or under another operating system.

2.4. Standard Services

Dynamic linking allows more than compatibility substitutions; it permits the creation of standard system services. A system might be shipped with an ISAM package, a SORT package, a MATH package and a few others. Users can buy enhanced or higher performance versions of these packages from the original vendor or from third parties, and have their existing applications immediately benefit from the upgrade.

2.5. Examples

Some simple examples of dynamic linking in use are presented in the following sections.

2.5.1. ISAM

Microsoft's ISAM package is accessed via dynamic linking. To begin with, the desired "central ISAM server" model is automatically obtained (although "separate server per client" model is available if preferred). Under MS-DOS 2.0, ISAM is terminate and stay resident and is accessed via a special INT function. Backward compatibility to MS-DOS 2.0 is provided by dynamically linked ISAM modules that issue the INT 21 to the existing 2.0 ISAM. (A special loader

program transparently does dynamic linking under MS-DOS 2.0. Naturally, there is no segment movability or swappability; there are no virtual segments under MS-DOS 2.0—the segments remain resident).

Microsoft will soon release a networked version of ISAM. A new ISAM dynamic library is provided whose entry points determine if the request should be served on the machine, or from across the network. If on machine, these routines in turn make dynamic calls to the ISAM server procedures. If across the network, they establish communication with an ISAM netserver on a remote machine.

For example, if the ISAM netserver is run on a local machine, allowing remote access to local ISAM data bases, it also dynamically calls the local ISAM server. As a result, there is complete flexibility with automatic optimal memory management. If network requests are not being made, the network code is not in memory. If local requests are not being made, the local code is not in memory. If both the local requestor and the local netserver are being used, they automatically and transparently share the same local ISAM package.

2.6. Service Packages

An example of a system service package would be a math/statistics library. A programmer might use such things infrequently, and only to handicap the football pools. The programmer would buy an inexpensive, small fast and "reasonably accurate" package.

Down the hall, a scientist doing critical research pays a large amount of money to buy a large, slow, but extremely accurate package that differentiates between 13 types of negative infinity.

As another ISAM example, perhaps an ISAM package has become bogged down because a data base has grown beyond the capacity of the package. No problem: get a new ISAM package with larger capacity, and all programs can instantly make use of it. It is believed that as soon as a market leader is established in any of these areas (ISAM, stat package, etc.) competitors will either provide interface-compatible interfaces or will provide dynamic routines that offer a compatible interface as an option. Competition under a standard interface will generate rapid improvements in software technology.

2.7. Commercial Interface Packages

A company may want to connect its Local Area Network (LAN) to a commercial "long-haul" packet net. The existing software can locate the "address" of correspondents by using a dynamically loaded directory package that was provided by the network vendor. This package takes calls in some standard format and generates vendor-specific network requests. It may converse with gateway machines, peruse clearing-house files, or whatever, as long as it provides the necessary information. Note that the "network address" for the intended target need not be standardized. It is just a "magic cookie" generated by the vendor's dictionary package for specification as an address to the vendor's network interface.

Later, the company may find that some other vendor is cheaper or faster. They can be wired into the second vendor's network, replace the old vendor's dictionary package with the new vendor's package, and continue business as usual.

2.8. Hardware Interface Packages

Although not a substitute for device drivers, dynamic linking can be very valuable in handling hardware. For example, one may own a high-resolution laser printer. A vendor supplied package does vector-to-raster conversion, font generation, and so forth. Device drivers allow independent communication with various devices; dynamic libraries will allow performance of higher-level functions in the same vein.

2.9. Operating System Related Services

There are a variety of services that are logically a part of the operating system (and that perhaps must track the MS-DOS releases), but they do not belong in the kernel. A couple of recent examples are error message lookup routines and pathname parsing routines.

Programmers do not want fixed linking with a routine that prints alpha explanations of error messages, because the next release of MS-DOS will contain error codes missing from the table. It isn't desirable to make this function an MS-DOS call because of the amount of wasted RAM. Instead, it is a dynamically linked routine that is re-released with each new MS-DOS release. Similarly, a new pathname-parsing package can be distributed with every MS-DOS release so that programs which parse pathnames can automatically handle any extensions made to the format.

2.10. High-Speed Operating System Calls

As observed earlier, the dynamic linking interface can be used to connect programs to the general library routines, operating system routines, the DOS kernel itself, or even the network. Some MS-DOS calls, for example buffered I/O and writing to the screen, are performance critical. INT 21 takes time because system mode is entered and exited.

Under MS-DOS 4.0, many parts of the operating system can be called in user mode via dynamic links while physically residing in the kernel. For example, buffered file writes would run in user mode until the buffer is discovered to be full; the routine would then issue the INT 21 to cause the buffer to be written.

LOTUS

3. How does it work?

This section provides a general overview of how dynamic linking works. The various details of the mechanism and directions for building the various "magic" files are presented later in this document.

A program calls a dynamically linked routine just as it would any far external routine: it declares the procedure a "far external" and then calls it at will. When the program is assembled or compiled, a standard external reference record is generated, no different than any other external procedure reference.

At link time, the programmer specifies one or more libraries which contain routines to satisfy those externals. External routines which are to be "true linked" are extracted from the library and linked in; external routines which are to be dynamically linked have special definition records in the library. These definition records tell the linker that the routine in question is to be dynamically linked; they provide the linker with a *module name* and an *entry name*. At this point the user has an .EXE file that is ready to be run. Some of the program's external references were satisfied by procedures taken from the libraries, others are described in the .EXE file as module/entry name pairs to be dynamically linked at load time.

Note that there is no distinction between calling a "regular" far procedure and a dynamically linked one. In fact, the programmer may not be aware that many of the procedures he has called will be dynamically linked.

When the program is to be run, the DOS loader loads it into memory and discovers the presence of dynamic links. The DOS uses the module name to locate the actual code for the procedure from a file in a special directory; the file's name is the module name. The DOS then loads the code and data segments in the called module, and links the callers to the called procedures.

Actually, this description is an oversimplification. Dynamic link modules must be fully *conforming* and as such are movable, swappable, shareable, and demand loadable. (See the document, *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT* for a more detailed discussion of these features.)

"Shareable" means that if more than one program wants to use the dynamically linked routines, the system does not load in more than one copy of the pure segments but shares a single copy among the users. Naturally, each client has its own set of any impure segments.

"Demand loading" is an adaptation of the swap mechanism. The DOS has the capability of swapping out a conforming segment and then swapping it back in when it is referenced. Pure segments cannot have been changed from their original form, so the DOS may discard these segments rather than swapping them. When they are referenced, the DOS reloads them from their original .EXE files. (DOS discards pure segments if they were originally loaded from a fixed media device or if there is no swap device.) Demand loading takes this process one step further. Some segments are not brought into memory at all when they are named by a loading program. They will be loaded only when (and if) they are referenced. The author of the dynamically linked routines has the option of

flagging each segment "load immediately" or "load on demand."

3.1. Run-time Dynamic Linking

In addition to "load-time linking" – the fixing up of dynamic link references when the client program is first loaded into memory, MS-DOS 4.0 also supports "run-time linking." Run-time linking is similar to the load-time linking, except instead of coding an explicit reference to an external procedure in the program source, the client program declares at run-time that it wants to call specific routines. As in load-time linking, the DOS loads the desired routines and fixes up a linkage for the client. The client supplies MS-DOS with a module name and a list of entry point names; MS-DOS loads the module and returns a list of addresses corresponding to the entry points.

In short:

	load-time	run-time
load request	automatic upon detection of special link record	Procedure call
use	far call	far call indirect thru vector
release	automatic at process death	automatic at death or via procedure call

The exact form of the routines used to accomplish run-time dynamic linking is described in Appendix A.

4. Library Modules

This document has discussed dynamic linking as a mechanism to provide load-time linking to library subroutines to supplement link-time access. A dynamic-link module which contains one or more such subroutines is called a *library module.*

There are three types of library modules which are supported: "no data," "global data," and "instance data." A no data module consists of one or more code segments but no data segments. The code uses no data values other than its parameters and other stack locations. When a routine in the module is

LOTUS

executed, the caller's DS remains in effect, allowing the library routine to reference structures and buffers in the caller's data segment if near pointers to them are passed. A far pointer may be passed to the library routine so that it can access an arbitrary data segment. In such a case, the segment locking rules described in the *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT* document apply.

The compiler and linker work in concert to ensure that a no data routine does not make non-pointer DS references.

A global data library routine has a single data segment. Although multiple programs may call this routine, each execution ("activation") of the routine uses the same data segment. The only way that this type of library routine can access the caller's data is via a call-by-value with far pointers passed as arguments. Note that this type of library routine must be careful to use RAM-based semaphores to interlock writes to its data segment because the DOS may context switch between activations of the same library routine (and thus the same data segment) at any time.

An instance data library routine has a separate data segment for every different calling process. The data segment is created when the program is linked to the library routine; it is destroyed when the calling program terminates. As with global data routines, the library routine can communicate with its caller only through values and long pointers passed as parameters. Because there is a separate data segment per activation, each activation can modify its data freely without the use of semaphores.

In all cases, dynamic linking is a subroutine call mechanism, not an interprocess communication mechanism. This means that when a process calls a dynamic subroutine, that subroutine runs as a component of the calling process. For example, if three processes are calling a global data library routine, and if that routine wants to read some data file, it will have to open the file once for each calling process and remember a different handle value for each opening. The isolation of the multiple instances of a global data routine can be an advantage. A global data library routine can have instance data by simply issuing the global-allocate system call when one of its client tasks calls it. The memory allocated will belong to the current process, not to the library routine. The routine must keep track of the different handles of the different client segments in its global data segment, or it must pass the handle back to the client and have the client "remind" it of the handle. Finally, when a client terminates the client-specific data segment vanishes. (The global data segment associated with the library routine remains until the last user of the library terminates.)

The distinction between the task model and dynamic linking is important because it keeps separate tasks from interfering with each other accidentally (encapsulation) or deliberately (security).

5. Restrictions Upon Library Modules

5.1. Library Modules Must Be Conforming

Library routines must be fully conforming, as described in the document *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT.* Conforming code maintains properly formatted stack, loads and stores values from/to the segment registers only in prescribed ways, and properly locks movable segments before generating far pointers to them.

Note that the calling ("client") routine need not be conforming, as long as it loads BP with zero before calling a dynamically linked routine and places only proper code segment values into CS (i.e., avoids "segment games" with the CS register). These should be easy restrictions to meet; non-conforming programs routinely play "segment games" with DS and ES but only rarely with CS.

The following discussion serves as a brief recapitulation of the ramifications of locking for each of the three library module formats. This discussion is not intended to be a complete definition of conformation; the programmer must also study the document *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT.*

No-Data Modules

These library routines have no data segments of their own. The caller's DS is in effect when they execute. This means that the caller can supply this type of library routine with near pointers to the caller's data. This, in turn, means that the caller's data segment does not have to be locked because no far pointers were generated.

However, if the module requires the caller to pass a *far pointer*, the segment which contains the pointed-to object must be fixed or locked.

Single-Data Modules

These library routines have a single data segment shared by all activations. As noted earlier, it is the responsibility of the library routine to watch for critical sections.

In this case, it is illegal for the caller to pass a short pointer because the library routine cannot make use of it; all addresses passed to this routine must be far pointers. Note that neither the compiler, assembler, linker nor the DOS can detect this kind of error.

Just as the caller can provide the library routine with a far pointer, the library routine could conceivably return a far pointer to its own data segment. The library routine should call **LockObject** to generate this far pointer (even if the segment is already locked) so that a matching **UnLockObject** can be performed when the pointer is no longer needed. The programmer must be careful to ensure that either a subsequent call to the subroutine unlocks its data segment, or that the handle of the module's automatic data segment calling routine has been passed so that the caller

can do the unlock. This mechanism is complex and allows disastrous bugs to be introduced, therefore it should be avoided if possible.

Multiple-Data Modules

These library routines have a separate data segment for every instance. They have no concerns about critical sections, but they are restrained like single-data segment routines.

5.2. Pure code

In addition to being conforming, dynamically linked library modules must contain all of their code in pure code segments. This restriction exists because any segment which calls an impure code segment is itself impure. (This is a result of the way the *call thunk* works.) Since dynamic library modules may call other such modules, a single impure library code segment could make many code segments in the system effectively impure. The resultant loss of segment sharing would seriously hamper the operation of MS-DOS 4.0.

5.3. Compatibility Between Dynamic Externals and True-Linked Externals

A programmer codes a call to a dynamically linked external in exactly the same way as to an external which will be linked at link time. However, depending upon the functionality of the routine there may be a significant difference between the two techniques.

Consider the case of a subroutine which needs some amount of static data. In the traditional link-time link process ("true link") the subroutine is assigned the memory it needs from the process's static data area. The linker fixes up the instructions in the subroutine that reference that area so that they point to the proper locations. Should some other program also be using that library routine, it repeats this process. Undoubtedly the static data area will end up at a different offset in the second program as it did in the first, but this presents no problem because each program has its own copy of the subroutine's code, and each copy was fixed differently (by the linker), to point to the proper location.

However, if this subroutine were part of a dynamic link library module, there would be only one copy of the code. The library might be prepared in one of two forms:

Instance Data Module

In this case, the subroutine's static data would be part of the module's private data segment. The offset to the static cells is the same, regardless of the calling task, because each different activation of the subroutine uses its own copy of the module's data segment.

The disadvantage to this scheme, however, is that if the calling program wants to pass addresses within its automatic data segment, it must generate far pointers that require the automatic data segment to be locked during the call.

No Data Module

If the subroutine takes pointers for some of its arguments and the programmer doesn't want to have to lock the automatic data segment, he might prepare the library routine as a no data library module. Since the subroutine will be using the caller's DS it can use near pointers. Unfortunately, the subroutine now has no place to store its static data. It could issue a **LocalAlloc** call to allocate the memory from the automatic data segment, but the offset of that memory will be different for every different client of the subroutine. Further, the subroutine has no static data area to store the pointer to the allocated static data area.

To sum up, a problem arises with library routines that:

1) Need to have some private static data.
2) Take pointers into the caller's automatic data segment as arguments *and* do not want to use far pointers or handle:offset pairs.

The most common reason that programs make the second requirement is to maintain source-level compatibility with an existing library routine that is not currently called via dynamic links. In this case, it may be undesirable to force users of the subroutine to change their source code to include the generation of far pointers and the **LockObject** and **UnLockObject** calls.

There are two recommended techniques to maintain an existing near pointer calling sequence to a dynamically linked subroutine which requires static memory. In both cases a special "transfer" subroutine is written. For a given library, the routine SUBR is renamed to LSUBR, and made into a dynamically linked routine. Then a stand-in SUBR is created that does the required segment locking and far pointer generation, and then calls LSUBR. The LSUBR routine actually does the work that the user thinks SUBR is doing. This renaming process is necessary to provide source compatibility with existing programs.

Method 1: A fix up transfer routine.

The transfer subroutine "SUBR" is defined as follows:

```
SUBR (a, b, c)
  char *a;          /* arbitrary parameters */
  char *b;
  int c;
  static int dshandle = 0;
{
        if (dshandle == 0)
            dshandle = GetDSHandle();
        LockObject (dshandle);
        LSUBR ( (farp)a, (farp)b, c);
        UnLockObject (dshandle);
}
```

This routine simply does the requisite segment locking and generation of the necessary far pointers so that the source of the caller of SUBR does not have to be changed. The module containing LSUBR would have an *instance* data segment.

Method 2: Pass a Pointer to the Static Data

Once again a "transfer" routine SUBR is used to call the real dynamically linked subroutine. In this method, however, the routine LSUBR is part of a no data library module. The transfer routine SUBR allocates the necessary static memory and passes LSUBR a near pointer to it.

```
SUBR (a);
   char *a;          /* arbitrary parameters */
{
        static struct workarea[10];

        LSUBR (a, &workarea);
}
```

Note that there is one problem with SUBR as shown here: the size of the static data area required is fixed in the .EXE file of the calling program, but the LSUBR routine that uses that memory is dynamically linked and may be updated. Perhaps a new release of the LSUBR library will require a larger static data area.

A more flexible implementation of the above is:

```
SUBR (a);
  char *a;
{
      static HANDLE han;
      static int init = 0;

      if (init == 0) {
            init++;
            han = LSUBR_INIT();
      }

      LSUBR (a, han);
}
```

In this case, the first call to the SUBR transfer routine calls another dynamically linked entry point "LSUBR_INIT." This in turn calls **LocalAlloc** to allocate the memory as a local object. Since LSUBR has no static memory of its own in which to store the handle of its static memory, it returns to SUBR the value (of the handle of its static memory) that was presented to LSUBR upon the call to SUBR.

Note that in both of these scenarios LSUBR must be written to access its data items via the passed handle or pointer. In the C language this would be done with a structure definition and the "->" operator.

5.4. Shared Language Run-time Libraries

It is a common characteristic of high level languages that the programs they produce include many unnecessary run-time library routines. This is tied to the fact that when the linker sees that Routine A calls Routine B, it has to link in Routine B, whether or not Routine B will ever be called. For example, the C language **printf** routine has a format specification for floating point output in exponential form. **Printf** contains a call to a subroutine to produce this format and consequently every C program that uses **printf** also loads the exponential format subroutine, regardless of whether or not the program uses that floating point format.

For this reason, it is highly desirable that the language run-time libraries be dynamically linked. A dynamically linked run-time library will solve the "fat program" problem in two ways:

1) There will be only one copy of the run-time library in memory, and it will be shared by all programs written in that language.

2) If rarely used routines (such as the exponential formatter) are put in their own code segments, The MS-DOS segment swapping feature will swap out unused routines until they are actually used.

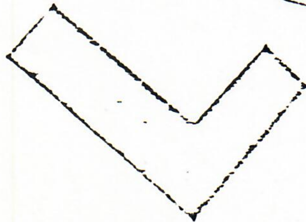
Since the operating system itself is invoked by means of library routines, dynamically linking the libraries will also provide an unprecedented degree of portability between operating systems themselves. It is anticipated that programs that *only* make use of the standard language run-time library will be binary-compatible between MS-DOS and XENIX.

A consideration of the library routine **printf** will show that a problem exists. **Printf** needs to take a pointer as part of its argument string and it needs to have its own static data (for buffers). However, the calling sequence to **printf** cannot be changed by requiring that it take far pointers. One of the solutions outlined previously could be used; they work quite well for **printf**. Unfortunately, they add too much CPU overhead for some of the more frequently called library routines.

The language run-time libraries solve this problem by means of a dedicated pointer WORD that appears at a constant location in the automatic data segment of every routine which calls those run-time routines. There are 24 of these WORDs allowing us to support up to 24 different libraries with this mechanism.

When the program is loaded, the library module's initialization entry point is called. This routine uses **LocalAlloc** to allocate the static data the library routines need; the handle of that local object is stored in the dedicated offset from the automatic DS of the caller. Each library subroutine that needs static data uses this pointer to find the local object and thence to find its section of the static data area.

Because of the difficulty in arbitrating the assignment of these limited fixed-offset cells, this mechanism will not be documented external to Microsoft.



6. Process Modules

Library modules can provide a variety of services, but there are fundamental limitations on their operation. They always execute as the calling process. This means that a shared library routine cannot have its own open files, file handles, assigned resources (such as a device or some shared memory), etc.

Some services, for example an ISAM package or a print spooler, want to have the same autonomy that is granted to processes:

- The ability to stay around regardless of the existence of clients.
- The ability to have its own open files (and probably, to restrict access by others to those files).
- The ability to have shared memory and other system resources assigned, and yet provide a dynamic-link service interface to other processes.

MS-DOS 4.0 allows processes to make dynamic-link entry points available. When the DOS loads the process into memory and starts its execution, it also registers the dynamic-link entry points it offers. This process can be started manually by user command or batch file, or automatically by MS-DOS when it brings the module into memory to satisfy a client's dynamic link request.

Recall that a library module can be configured to be no data, instance data, or global data. A process module must use the global data form. Only one copy of the process module can be run at one time; an attempt to Exec a second copy produces a "multiple definition of dynlink entry" error message.

When a client process calls one of the dynamic-link entry points of a process module, the called code is running as the client process, *not* as the module process. The dynamic link call behaves just like a library module dynamic link call; the call itself does not involve a process switch. As is the case for a call to a library module with a global data segment, the automatic code segment belongs to the process module. The automatic data segment is the module's data segment, but the calling process continues to use its own stack segment and stack area.

The net effect is that a process module shares access to its code segments and automatic data segment with the client processes, but its stack and frame variables are private. Also, a client can only call subroutines in the process module that are listed as dynamic link entry points. Further, any resources or global memory segments which the process module allocates belong to it exclusively. They cannot be accessed by the client process, even when the client process is executing code in the process module. (Of course, the 8086/8088 is an unprotected machine and cannot enforce memory privacy rules. They are enforced when running in protected mode on the 286. MS-DOS itself supports the privacy of the other resources.)

The best way to understand process modules it to consider them as independent service tasks which provide a client interface via dynamic links instead of (or in addition to) via named pipes or shared memory. When the client makes a

request, one of the offered entry points is called. This code has access to the process module's data segment, but it is not running as the process itself. If the request can be satisfied by running some of the process module's subroutines as the client task, then they would return to their callers. If the request requires some action on the part of the module's own process, then the process module's code would have the client process communicate with the module's process via some form of inter-process communication (IPC): perhaps named pipes or a block of shared memory.

For example, consider a simple ISAM server package. It wants to be the exclusive user of an ISAM file and to provide its clients with controlled access to that file. Upon startup, the ISAM process would open the file and setup some buffers. It would probably put these buffers in a manual, movable segment rather than the automatic data segment. This is because all of ISAM's clients would be able to read and write the automatic data segment and ISAM doesn't want one client to be able to see data belonging to another client. When MS-DOS links the client and ISAM together it would give the client access to all of ISAM's initial data segments. Any resources that the ISAM process obtains while executing: open files, more memory segments, etc., would belong to it exclusively.

When a client wants to obtain ISAM services, his program calls an ISAM_OPEN dynamic link routine. This routine might allocate an area of shared memory to hold client-specific buffers and then use a named pipe to announce the new client to the ISAM process. Note that although the client process and the ISAM process may be executing the same code, they use different segments and must use IPC mechanisms to communicate.

Perhaps the client later wants to read the next record in a sequence. It might call ISAM_READ with its request. ISAM_READ would look in the shared-memory segment that is shared between client and ISAM to see if the information is already in memory. If it is, then ISAM_READ can return without having to communicate with the ISAM process. If the data is not available, it might be in the ISAM master buffer or it might be on disk. In either case, the code cannot access the data directly because both the ISAM master buffer and the disk file are inaccessible to the client process. The ISAM_READ code must use some form of IPC to request that the ISAM process get the data and put it in the ISAM/client shared memory area.

6.1. Keeping track of clients

Some service packages may be casual and not take much notice as clients come and go. Others may have client-specific information and/or resources and may need to be notified of new client arrivals and client terminations. Three mechanisms are provided for this:

Client Arrival

Client Death

Process Module Death

After a process module's last client has terminated, the module may want to clean up and terminate rather than remain resident in memory and occupy system resources. This must be done carefully to avoid the problem of a new client arriving while the process module is getting ready to exit. In fact, the arrival could take place one instruction before the module issues the **Exit** system call.

The **StopModule** call is provided to deal with this situation. **StopModule** is called when the process module believes that its last client has terminated. If there are no current users of the module's dynamic entry points, then those entry points are removed from the system dynamic link tables. When **StopModule** returns with a "OK" status then the process module is assured that it will not have another client and it can terminate without interruption. Note, however, that if a new client comes along a *new* copy of the process module will be loaded for it, because the old copy has terminated. This means that authors of process modules need to interlock their global resources (such as files), so that the incoming copy waits until the outgoing one has terminated. This could be done via a semaphore, for example.

6.2. Initialization Synchronization

As discussed previously, MS-DOS will start a process module running when it sees a dynamic link request to it, if it isn't running already. This means that it is possible for a client program to make a dynamic call to one of the process module's service points *before* the module's own process has started. It is up to the author of the process module to handle this situation. A recommended method is to define some already-set RAM-semaphores in the process module's data segment. Those dynamic link entry points that may be called prematurely should be blocked until those semaphores are cleared. The process module's process clears the semaphore when it has finished initializing the module.

7.3. Withdraw Request

StopModule()

Unlike **LoadModule** and **FreeModule**, **StopModule** is not called by a client program, but only by a process module. If there are are no clients currently dynamically linked to the process module, **StopModule** removes the module's dynamic-link entry points from MS-DOS's "available" list, and returns a boolean TRUE (FFFFh). If there *are* still clients linked to the process module **StopModule** does nothing and returns a boolean FALSE (0).

StopModule is provided so that a process module can, when it believes that it has no more clients, clean up and terminate without the possibility of a new client arriving during that process. Should a program request a link to the process module a new copy will be started, even though the old copy may not have exited yet. For this reason, process modules that manipulate global resources (such as files or devices) should use some mechanism such as system semaphores to delay the running of the new copy until the old copy has terminated.

LOTUS