Part III

The System Loader

The System Loader is a Cortland Toolset that works closely with ProDOS/16. It is responsible for loading all code and data into the Cortland's memory. It is capable of static and dynamic loading and relocating of program segments, data segments, subroutines and libraries.

Chapter 15 explains in general terms how the System Loader works. Chapter 16 details some of its functions and data structures. Chapter 17 gives programming suggestions for using the System Loader. Chapter 18 shows how to make loader calls and describes each call in detail. Chapter 19 is a complete list of System Loader error codes.

Date: August 9, 1986

Author: Lou Infeld

Subject: System Loader ERS

Document Version Number: 00:97

Revision History

00:00	(09/06/85)	Initial Release
00:01	(09/16/85)	New Jump Table Entry format
		Dynamic Segment Level replaced with Usage Counter
		Load Segments are numbered from 1
		Segment Jump Table Transfer function added
		Unload ALL dynamic segments allowed
		Error numbers changed
		Change Segment Attributes function removed
		Lock and Unlock Segment functions added
00:02	(09/18/85)	Support added for old System and binary files
		The Initial Load and Segment Jump Table Load will return an error rather than display an error
		Unload priority added to segment attributes
00:30	(11/26/85)	Tool Dispatcher interface specified
		Library Files redefined
		Load File Table changed to Pathname Table
		Optional Initialization Segment supported
		Memory Segment support removed
		Interface to Memory Manager updated
		Load Segments are again numbered from 0
		Return Address Stack added
		Segment Jump Tables can be linked
		Pathname Tables can be linked
		UserID support added
		Run Time Library version control added
		Position Independent Segment support added
		Usage Counter is now 2 bytes
		Attributes no longer needed
		Memory Manager discussion added
		Get UserID call added
		Controlling Program definition added
		Unload "All" function moved to Memory Manager
		Optional Starting Address added to Init Load
		Init Load will no longer start the application
		Clean Up routine added
		Application Shutdown routine added
		Memory Segment entry format changed
00:40	(01/03/86)	Segment Jump Table changed

		Memory Segment entry format changed
		Usage Counter renamed In-Use Counter
00:50	(01/30/86)	UserID's are now 2 bytes
		Support of old binary files removed
		Starting Address parameter removed from Initial Load
		Initialize function added
		All function numbers changed
		Error $79 added
		Calling sequence changed
00:60	(02/27/86)	System Loader no longer part of ProDOS
		All function numbers changed
		Loader Version function added
		All error numbers changed
		Error $1109 redefined
		Errors $110A and $110B added
		Pathname syntax discussion changed
		Memory Segment entry format changed
		Support of System Files removed
		UserID added as output from Initial Load
00:70	(04/16/86)	System Loader tool number changed to 17 ($11)
		Standard Tool functions added
		UserID added to Memory Segment Table
		Next UserID removed from Memory Segment Table
		File Pathname Pointer in Pathname Table made relative
		Header description added to Pathname Table
		Optional Initialization Segment renamed Initialization Segment and generalized
		Shell Load Files defined
00:80	(05/07/86)	Load Segments start with 1 rather than 0
		Pathname Table description expanded
		Startup Load Files defined
		System prefixes 0,1 and 2 defined
		Loader will not allocate bank 0 memory for application
		Get UserID function redefined
		"Memory not available" error removed
		"System Loader busy" error added
		Function number of Initial Load changed to $08
00:90	(05/19/86)	UserID error removed from Application Shutdown
		UserID added as output from Application Shutdown
		Function number of Initial Load changed to $07
		Restart function removed from Initial Load and made $08
		Pathname Table redesigned
		More details about UserID in Initial Load
		Zero Page and stack allocated in Initial Load
		Zero Page/stack info output in Initial Load and Restart
		UserID added as input to Cleanup Routine
00:91	(05/27/86)	Absolute Bank Segment defined
		Zero Page/Stack Segment defined replacing Aux Type logic

00:92	(06/06/86)	InitialLoad will not allocate default Zero Page/stack
		Techniques used to free memory described
		Function numbers and names changed
		UserID added to Segment Jump Table entry
		Full UserID allowed in InitialLoad
00:93	(06/16/86)	Loader supports DS records
		Memory Manager Interface revised
		Restrictions section added
		New input parameter added to InitLoad
00:94	(06/26/86)	Zero Page/Stack Segment renamed Direct Page/Stack Segment
		Segment Jump Table renamed Jump Table List
		Segment Jump Table Segment renamed Jump Table Segment
00:95	(07/15/86)	General - automatic unload design removed:
		Memory Segment Table entry format changed
		"Unloaded" Jump Table entry format changed
		Return Stack removed
		In-Use Counter removed
		Jump Table Transfer function removed
		Unload Segment function added
		Jump Table discussion changed
		Format of Jump Table List added
		References to "memory segments" changed to "memory blocks"
		GetUserID function removed
		Restart restrictions added
		InitialLoad description updated
		LoadSegNum description updated
		UserID input added to LockSeg and UnlockSeg
		UnloadSegNum description changed
00:96	(07/22/86)	Output of LoadSegNum changed to an Address
		LoadSegNum can be used to load static segments
		List of Error Codes added
		UnloadSegNum description changed
		LockSeg and UnLockSeg removed
		Cleanup Routine description updated
00:97	(08/09/86)	Mark List defined
		Support for cRELOC and cINTERSEG records added
		Outputs and Errors changed for LoadSegName
		The term NIL changed to 0 in several Tables
		Restartable Flag added to UserShutdown
		GetUserID function added
		GetPathname function added
		Pathname Table description modified
		Special Memory Flag renamed in InitialLoad

Index

History

The Apple // under ProDOS has a very basic System Loader. It is the part of the boot code that searches the boot disk for the first System file (any file of type $FF whose name ends with ".SYSTEM") and loads it into location $2000. If a System program wants to load another System program, it has to do all the work by making ProDOS calls.

Some programming environments such as Apple // Pascal and AppleSoft Basic provide loaders for programs running under them. The AppleSoft loader loads either System files, Basic files or binary code files. All these files are loaded either at a fixed address in memory or at an address specified in the file.

Since Cortland will have a large amount of "clean" memory, a more dynamic load facility is needed. Programs should be able to be loaded anywhere that is available in memory. The burden of determining where to load a program should be on a loader and not on the applications programmer. Also programs should be able to be broken into smaller program segments which can be loaded independently.

Therefore on Cortland, there will be a relocating System Loader. Files generated by the Linker will be loadable by the System Loader. The System Loader will provide a very powerful and flexible facility not currently available on the Apple //.

Overview

The System Loader will load programs or program segments by first calling the Memory Manager to find available memory. It will perform relocation during the load as necessary and will load each segment independently. Therefore, a large program can be broken up into smaller program segments each of which is loaded at separate locations in memory. Program segments can also be loaded dynamically as they are referenced rather than at program boot time. Additionally, the System Loader can be called by the program itself to load and unload program (or data) segments.

Definitions

The **Linker** is the program that combines files generated by compilers and assemblers, resolves all symbolic references and generates a file that can be loaded into memory and executed.

The **System Loader** is the part of the Operating System that reads the files generated by the **Linker** and loads them into memory (performing relocation if necessary).

Object Files are the output from an assembler or compiler and the input to the **Linker**.

Library Files are files containing general program Segments that the **Linker** can search.

Load Files are the output of the **Linker** and contain memory images which the **System Loader** will load into memory. Shell Load Files and Startup Load Files are special Load Files used by the Shell and ProDOS16 respectively.

Run Time Library Files are Load Files that contain general program Segments which can be loaded as needed by the **System Loader** and shared between applications.

Object Module Format is the general format used in Object Files, Library Files and Load Files.

An **OMF File** is a file in Object Module Format (i.e. an Object File, Library File or Load File).

A **Segment** is a individual component of an OMF file. Each file contains one or more Segments.

A **Code Segment** is a Segment in an Object File that contains program code.

A **Data Segment** is a Segment in an Object File that contains program data.

A **Load Segment** is a Segment in a Load File.

The **Controlling Program** is the program that requests the **System Loader** to initially load and run other programs and is responsible for shutting these programs down when they exit. A Finder is an example of a Controlling Program.

General

The **System Loader** processes files which conform to the Cortland Development Environment's definition of a Load File (see Cortland Object Module Format ERS). A Load File consists of Load Segments, each of which can be loaded independently. The Load Segments are numbered sequentially from 1.

Certain Load Segments are Static Load Segments. These Segments are meant to be loaded into memory at initial program load time and must stay in memory until program completion.

Another general type of Load Segments is the Dynamic Load Segment. These Segments are not loaded at boot time. They are loaded dynamically during program execution. This can happen automatically by means of the **Jump Table** mechanism or manually at the specific request of the application. When these Segments are not being referenced, they can be purged by the **Memory Manager.**

The last general type of Load Segments is the Position Independent Load Segment. These Segments have the attribute that they can be moved while in memory.

There are several special types of Load Segments. The **Jump Table Segment** (KIND=$02), when loaded into memory, provides a mechanism whereby Segments in memory can trigger the loading of other Segments not yet in memory.

The **Pathname Segment** (KIND=$04). It contains information about the Load Files that are referenced.

The **Initialization Segment** (KIND=$10). It is used for code that is to be executed before all the rest of the Load Segments are loaded.

Absolute Bank Segments (KIND=$11) are relocatable but only within the bank specified by the ORG field.

The **Direct Page/Stack Segment** (KIND=$12) defines the application's Direct Page and stack requirements. This segment will be loaded into Bank 0 and its starting address and length are passed to the **Controlling Program** who will set the Direct Register and Stack Pointer to the start and end of this segment before transferrring control to the program.

During the initial load, the **System Loader** has all the information needed to resolve all inter-segment references between the Static Load Segments. But during the dynamic loading of Dynamic Load Segments, it can only resolve references in the Dynamic Load Segment to the already loaded Static Load Segments. Therefore, the general rule is that Static Segments can be referenced by any type of segment but Dynamic Segments can only be referenced through JSL calls through the **Jump Table.**

If the **System Loader** is called to perform the initial load of a program, it will load all the Static Load Segments and the Segment Jump and Pathname Tables (if they exist). A RAM based **Memory Segment Table** will be constructed during this process.

If the **System Loader** is called during an interrupt and it is already processing a request, a BUSY error ($1105) will result.

Memory Manager Interface

The **System Loader** and the **Memory Manager** work closely together.

When the **System Loader** loads Static Segments, it calls the **Memory Manager** to allocate corresponding memory blocks which are marked as unpurgeable and unmoveable. Dynamic Segments are marked as purgeable but locked. Position Independent Segments are marked as moveable.

When the **System Loader** unloads a specific segment, it calls the **Memory Manager** to purge the corresponding memory blocks. However, if the **Controlling Program** wishes to unload all segments associated with a UserID (application shut-down), it calls the **System Loader** Application Shutdown function which calls the **Memory Manager** to first purge all Dynamic Segments for the UserID and then make all the Static Segments purgeable. The purpose of this is to keep an application in memory, if possible, in case it needs to be re-loaded in the near future. This will greatly speed up a Finder or Switcher. The complication occurs when the **Memory Manager** has to actually purge one of the segments of a User. The **System Loader** must then purge all the remaining segments. Otherwise, the program will not have all its static segments in memory when it is re-loaded and executed.

The relationship between a Load Segment in a Load File and the corresponding memory block is very close. The average Load Segment will be loaded into a memory block having the attributes:

Locked
Fixed
Purge Level=0 (for Static)
Purge Level=1 (for Dynamic)

Depending on the ORG, KIND, BANKSIZE and ALIGN fields in the Segment Header, other memory attributes will be used:

if ORG>0, the "Fixed Address" attribute is set.
if BANKSIZE=$10000, the "May not cross bank boundry" attribute is set.
if 0<BANKSIZE<$10000 then use Align factor=MAX(BANKSIZE,ALIGN)
    otherwise use Align factor=ALIGN:
if 0<Align Factor<=$100, the "Page Aligned" attribute is set.
if Align Factor>$100, Bank Alignment is forced (not an attribute).
if bit 5 of KIND=1, the "Fixed" attribute is removed.
if KIND indicates Absolute Bank Segment, the "Fixed Address" attribute is
    removed and the "Fixed Bank" attribute is set.
if KIND indicates Direct Page/Stack Segment, the "Fixed Bank" and "Page
    Aligned" attributes are set.

A memory block can be locked by a call to the **System Loader**. However, the other attributes must be changed through **Memory Manager** calls. Since the Memory Handle for a memory block is stored in the **Memory Segment Table, Memory Manager** information is accessible. Other memory block information that may be useful to a program are:

Start location
Size of segment
UserID
Purge Level (0 - UnPurgeable
1 - Least Purgeable
3 - Most Purgeable)

Also, if the Memory Handle is NIL (i.e the Memory Address is 0), the memory block has been purged.

Restrictions

The Object Module Format and the Linker have general capabilities above what is needed or desired for the Cortland computer. The **System Loader**, on the other hand, is designed specifically for the Cortland computer. Therefore, there are certain abilities that are not supported or are restricted. This section will list these differences.

The NUMSEX field of the Segment Header must be 0.
The NUMLEN field of the Segment Header must be 4.
The BANKSIZE field of the Segment Header must be <=$10000.
The ALIGN field of the Segment Header must be <=$10000.

If any of the above is not true, the **System Loader** will return with a "Segment is foreign" error ($110B). The BANKSIZE and ALIGN restrictions will be enforced by the **Linker** and should not make it to the Load File.

ALIGN and BANKSIZE can be any multiple of 2. The **Memory Manager**, and therefore the **System Loader**, can not handle so general a requirement. The **Memory Manager** can currently only be told that a memory block be page aligned or not cross a bank boundary. The **Memory Manager** may handle Bank Alignment in the near future. All is not lost, however, because the **System Loader** will fulfill the general requirements in the following, somewhat inefficient, way:

Any value of BANKSIZE other than 0 and $10000 will result in a memory block that is either page aligned (if BANKSIZE<=$100) or bank aligned (if BANKSIZE>$100). Since the **Linker** will make sure that the segment is smaller than BANKSIZE, the requirement that the segment not extend past the BANKSIZE boundary will be met (there will be wasted space in the memory block however).

Any value of ALIGN will be bumped to either page alignment or bank alignment.

If there is a BANKSIZE other than 0 and $10000 and a non-zero ALIGN, the maximum of the two will be used to determine the alignment to be used.

Data Structures

Globals

SEGTBL	-- Absolute address of Memory Segment Table
JMPTBL	-- Absolute address of Jump Table List
PATHTBL	-- Absolute address of Pathname Table
USERID	-- UserID of current application

Memory Segment Table

The **Memory Segment Table** is a linked list. Each entry corresponds to one memory block known to the **System Loader**. These memory blocks were the result of loading Load Segments from a Load File. The format of each entry in the **Memory Segment Table** is:

Next entry handle	-- 4 bytes
Previous entry handle	-- 4 bytes
UserID	-- 2 bytes
Memory Handle	-- 4 bytes
Load File Number	-- 2 byte
Load Segment Number	-- 2 bytes
Load Segment Kind	-- 2 bytes

where:

"Next entry handle" is the memory handle of the next entry in the Memory Segment Table. This handle is 0 in the last entry.

"Previous entry handle" is the memory handle of the previous entry in the Memory Segment Table. This handle is 0 in the first entry.

"UserID" is the UserID associated with this segment. It is needed in case the "Memory Handle" is NIL and the UserID can therefore not be determined directly from the Memory Manager.

"Memory Handle" is the handle of the memory block obtained from the **Memory Manager**. More information about the segment is available through this handle (e.g. UserID, Purge Priority).

"Load File Number" corresponds to the Load File or Run Time Library File from which the segment was obtained. If this number is 1, this segment is in the initial Load File.

"Load Segment Number" is the segment number of the Load Segment in the Load File.

"Load Segment Kind" is the KIND field from the Segment Header of this segment.

Jump Table List

The **Jump Table List** (or **Jump Table**) is the mechanism that allows programs to reference segments that are loaded into memory only when they are needed. The **Jump Table** is a linked list containing the UserID and Handle to each **Jump Table Segment** (KIND=$02) that the **System Loader** has encountered. Any Load File and Run Time Library File may contain a **Jump Table Segment.** The format of each entry in the **Jump Table List** is:

Next entry handle	– 4 bytes
Previous entry handle	– 4 bytes
UserID	– 2 bytes
Memory Handle	– 4 bytes

where:

"Next entry handle" is the memory handle of the next entry in the Jump Table List. This handle is 0 in the last entry.

"Previous entry handle" is the memory handle of the previous entry in the Jump Table List. This handle is 0 in the first entry.

"UserID" is the UserID associated with this Jump Table Segment.

"Memory Handle" is the handle of the memory block associated with this Jump Table Segment.

When the **Linker** encounters a JSL to an external Dynamic Segment, it creates an entry in the **Jump Table Segment.** It then links the JSL to the **Jump Table Segment** entry it just created. The format of this entry in the **Jump Table Segment** is:

UserID (2 bytes)
Load File Number (2 bytes)
Load Segment Number (2 bytes)
Load Segment Offset (4 bytes)
jsl Jump Table Load Function

where the Load File Number, Segment Number and Offset refer to the location of the external reference. The rest of the entry is a call to the **System Loader** Jump Table Load function. The UserID and the actual address of the **System Loader** function will be patched by the **System Loader** during Initial Load. This format is considered the "unloaded" state of the entry.

When the JSL instruction actually executes, control is transferred to the **Jump Table** entry which in turn transfers to the **System Loader.** The **System Loader** extracts the segment information from the **Jump Table** entry, the file information from the

Pathname Table and loads the Dynamic Segment, changes the entry in the **Jump Table** to its "loaded" state and transfers to the location in the just loaded segment. Typically, the location in the loaded segment is a subroutine and when it exits with a RTL, control is eventually transferred to the location following the original JSL instruction.

The loaded state of a **Jump Table** entry is very similar to the unloaded state except that the JSL to the **System Loader** Jump Table Load function is replaced by a JML to the external reference. A typical loaded entry would look like this:

UserID (2 bytes)
Load File Number (2 bytes)
Load Segment Number (2 bytes)
Load Segment Offset (4 bytes)
jml external reference

Pathname Table

The **Pathname Table** is created by **System Loader** to remember the pathnames associated with each Load File it comes across. At initial load, the **System Loader** creates the first entry in the **Pathname Table** from the pathname specified in the Initial Load function call. During the load, if the **System Loader** comes across a Pathname Segment (KIND=$04), it adds all the pathname entries to the **Pathname Table**. If Run Time Library Files are referenced during program execution, other Pathname Segments may be added.

Each entry in the **Pathname Table** is in the following format:

Next entry handle (4 bytes)
Previous entry handle (4 bytes)
UserID (2 bytes)
File Number (2 bytes)
File Date (2 bytes)
File Time (2 bytes)
Direct Page/Stack Address (2 bytes)
Direct Page/Stack Size (2 bytes)
File Pathname (Pascal string)
----

where:

"Next entry handle" is the memory handle of the next entry in the **Pathname Table**. This handle is 0 in the last entry.

"Previous entry handle" is the memory handle of the previous entry in the **Pathname Table**. This handle is 0 in the first entry.

"UserID" is the UserID associated with this entry. In general, each Load File and each Run Time Library will have a different UserID and one entry in the **Pathname Table.** When a Run Time Library is first encountered during an Application execution, the **System Loader** will have a Run Time Library type of UserID assigned to it.

"File Number" is a number assigned by the **Linker** or **System Loader** for a specific Load File. File number 1 is reserved for the initial Load File.

The "File Date" and "File Time" are ProDOS directory items that the **Linker** retrieved during the link process. The **System Loader** will compare these values with the ProDOS directory of the Run Time Library File at run time. If they don't compare, the **System Loader** will not load the requested Load Segment. This facility guarantees that the Run Time Library File used at link time is the <u>same</u> Run Time Library File loaded at execution time.

The "Direct Page/Stack" Address and Size is the information about the Direct Page and Stack buffer that was allocated during the Initial Load of this Load File (not applicable to Run Time Library Load Files). This allows the Restart function to resurrect an application without performing a Get File Info call on the Load File.

The "File Pathname" is the pathname of this entry. ProDOS 16 supports 8 prefixes, three of which have fixed definitions:

0/ - Boot volume
1/ - Application subdirectory (out of which the application is running)
2/ - System Library subdirectory (initially /BOOT/SYSTEM/LIBS)

The pathname must be a complete pathname except if either prefix 1/ or 2/ are used.

<u>Mark List</u>

The **Mark List** is created by **System Loader** to remember the file locations of the relocation dictionary of each Load Segment. The format of the **Mark List** is:

```
------------------------------
Next Available Slot (4 bytes)
End of Table (4 bytes)
Segment Number (2 bytes)
File Mark (4 bytes)
Segment Number (2 bytes)
File Mark (4 bytes)
Segment Number (2 bytes)
File Mark (4 bytes)
...
...
Segment Number (2 bytes)
File Mark (4 bytes)
```

---

where:

"Next Available Slot" is the relative offset of the next empty entry in the **Mark List**.

"End of List" is the relative offset to the end of the **Mark List**.

The **Mark List** is initially large enough for 100 Marks and grows larger as needed.

Functions

General

Since the **System Loader** is a Cortland Tool, its functions are called by making calls through the Cortland Tool mechanism. The calling sequence for **System Loader** functions is the standard Tool calling sequence. Space for the output parameter (if any) is pushed on the stack followed by each input parameter ***in the order specified in the function description***. This is followed by:

```
ldx #$11+FuncNum|8
jsl Dispatcher
```

where "FuncNum" is the **System Loader** function number and the "$11" is the Tool Number for the **System Loader**. Upon return, the A register will contain the status and the Carry will be set if an error occurred. If there is output, each output parameter must be pulled off the stack ***in the order specified in the function description***.

The Jump Table Load function does not use the above calling sequence. It can not be called by an application directly but is called indirectly by a Jump Table entry. In this case the absolute address of the function is patched by the **System Loader**.

Loader Initialization

Function Number:	$01
Input:	none
Output:	none
Errors:	none

This function will initialize the System Loader. It should only be called at system initialization time. All System Loader tables are cleared and no assumptions are made about the current or previous state of the system.

<u>LoaderStartup</u>

Function Number:	$02
Input:	none
Output:	none
Errors:	none

This function does nothing and need not be called.

LoaderShutdown

Function Number:	$03
Input:	none
Output:	none
Errors:	none

This function does nothing and need not be called.

LoaderVersion

Function Number:	$04
Input:	none
Output:	Loader Version (2 bytes)
Errors:	none

This function will return the Version Number of the System Loader.

LoaderReset

Function Number:	$05
Input:	none
Output:	none
Errors:	none

This function does nothing and need not be called.

<u>LoaderStatus</u>

Function Number:	$06
Input:	none
Output:	Status (TRUE or FALSE 2 bytes)
Errors:	none

This function will always return TRUE since the **System Loader** will always be in the initialized state.

InitialLoad

Function Number:	$09
Input:	UserID (2 bytes)
	Address of Load File Pathname (4 bytes)
	Don't Use Special Memory Flag (2 bytes)
Output:	UserID (2 bytes)
	Starting Address (4 bytes)
	Address of Direct Page/Stack buffer (2 bytes)
	Size of Direct Page/Stack buffer (2 bytes)
Errors:	$0000 - Operation succcessful
	$1104 - File not Load File
	$1105 - System Loader is busy
	$1109 - SegNum out of sequence
	$110A - Illegal load record found
	$110B - Load Segment is foreign
	$00xx - ProDOS error
	$02xx - Memory Manager error

A **Controlling Program** (such as ProDOS, Basic, Switcher, etc.) will call the **System Loader** to perform an "Initial Load".

If a complete UserID is specified, the **System Loader** will use that when allocating memory for the Load Segments. If the Main ID portion of the UserID is 0, a new UserID is obtained from the UserID Manager based on the Type portion of the UserID. If the Type portion is 0, an Application type UserID is requested from the UserID Manager.

If the Don't Use Special Memory Flag is TRUE (i.e. not 0), the **System Loader** will **NOT** load any static load segments into Special Memory. However, dynamic load segments will be loaded into any memory.

ProDOS is called to open the specified Load File using the input pathname. If any ProDOS errors occurred or if the file is not a Load File type ($B3-$BE), the **System Loader** will return the appropriate error.

If the Load File was successfully opened, the **System Loader,** adds the Load File information to the **Pathname Table**, and calls the Load Segment by Number function for each Static Load Segment in the Load File.

If an Initialization Segment (KIND=$10) is loaded, the **System Loader** will immediately transfer control to that segment in memory. When the **System Loader** regains control, the rest of the static segments are loaded normally.

If the Direct Page/Stack Segment (KIND=$12) is loaded, its starting address and length are returned as output. This buffer is treated as a locked dynamic segment and is therefore purged at Application Shutdown.

If any of the static segments could not be loaded, the **System Loader** will abort the load and return the error.

After all the Static Load Segments have been loaded, return is made to the **Controlling Program** with the starting address of the first Load Segment (not an Initialization Segment) of File Number 1. Note that the **Controlling Program** is responsible for setting up the stack and Direct Page registers and actually transferring control to the loaded program.

Restart

Function Number:	$0A
Input:	UserID (2 bytes)
Output:	UserID (2 bytes)
	Starting Address (4 bytes)
	Address of Direct Page/Stack buffer (2 bytes)
	Size of Direct Page/Stack buffer (2 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Application not found
	$1105 - System Loader is busy
	$1108 - UserID error
	$00xx - ProDOS error
	$02xx - Memory Manager error

A **Controlling Program** (such as ProDOS, Basic, Switcher, etc.) can call the **System Loader** to perform a "restart" of an application still in memory. Only software that is "reentrant" can be successfully restarted. For a program to be "reentrant", it must initialize its variables and not assume that they will be preset at Load time. The **Controlling Program** must determine whether a given program can be restarted.

An existing UserID (ignoring the Aux ID) must be specified, otherwise the **System Loader** will return error $1108. If the UserID is not known to the **System Loader**, error $1101 will be returned.

Applications can be "restarted" only if all the segments in the Memory Segment table with the specified UserID are in memory. Note these segments are the application's static segments. If this is the case, the **System Loader** resurrects the application by calling the Memory Manager to lock and make all its segments unpurgeable. The UserID and the starting address obtained from the first segment are returned as well as the Direct Page/Stack information from the Pathname Table.

If there is a Pathname Table entry for the UserID but not all the segments are in memory, the Cleanup Routine will be called to purge the UserID (and any other "deceased" UserIDs) from all its tables, call the UserID Manager to delete the UserID and then perform an Initial Load instead of a Restart. In this case, a new UserID will be established for the application.

<u>LoadSegNum (Load Segment by Number)</u>

Function Number:	$0B
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
Output:	Address of segment (4 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1104 - File not Load File
	$1105 - System Loader is busy
	$1107 - File Version error
	$1109 - SegNum out of sequence
	$110A - Illegal load record found
	$110B - Segment is foreign
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will load a specific Load Segment into memory. This is the workhorse function of the **System Loader**. Normally, a program will call this function to manually load a Dynamic Load Segment. If a program calls this function to load a Static Load Segment, the **System Loader** will <u>not</u> patch any existing references to the newly loaded segment.

First the **Memory Segment Table** is searched to see if there is an entry for the requested Load Segment. If there is already an entry, the handle to the memory block is checked to verify it is still in memory. If it is still in memory, this function does nothing further and returns without an error. If the memory block has been purged, the **Memory Segment Table** entry is deleted.

Next the "Load File Number" is looked up in the **Pathname Table** to get the Load File pathname.

Next the Load File type is checked. If it is not a Load File (types $B3-$BE), error $1104 is returned.

Next the Load File's "last_mod" value is compared to File Date and File Time values in the **Pathname Table**. If these values do not match, error $1107 is returned. This indicates that the Run Time Library File at the specified pathname is <u>not</u> the Run Time Library File that was scanned when the application was linked together.

ProDOS is then called to open the specified Load File. If ProDOS has a problem, its error code is returned.

Next the Load File is searched for a Load Segment corresponding to the specified "Load Segment Number". If there is no segment corresponding to the "Load Segment Number", error $1101 is returned. If the SEGNUM field does not correspond to the "Load Segment Number", error $1109 is returned. If the NUMSEX and NUMLEN fields are not "0" and "4", error $110B is returned.

If the Load Segment is found and its Segment Header is correct, a memory block is requested from the **Memory Manager** of size specified in the LENGTH field in the Segment Header. If the ORG field in the Segment Header is not 0, a memory block starting at that address is requested. Other attributes are set according to Segment Header fields (see Memory Manager Interface section).

If the UserID specified is not 0, it is used as the UserID of the memory block. If the UserID specified is 0, the memory block will be marked as belonging to the UserID of the current User (in USERID).

If the requested memory is not available, the **Memory Manager** and the **System Loader** will try several techniques to free up memory:

> The **Memory Manager** will purge memory blocks that are marked purgeable
>
> The **Memory Manager** will move moveable segments to enlarge contiguous memory
>
> The **System Loader** will call its Cleanup routine to free its own unused internal memory

If all these techniques fail, the **System Loader** will return with the last **Memory Manager** error.

Once enough memory is available, the Load Segment is loaded into memory and the relocation dictionary (if any) is processed. Note only the following Object Module Format records are supported by the **System Loader:**

```
LCONST ($F2)
DS ($F1)
RELOC ($E2)
INTERSEG ($E3)
cRELOC ($F5)
cINTERSEG ($F6)
END ($00)
```

Any other records encountered will result in a $110A error.

A new entry is added to the **Memory Segment Table.**

Finally, the **System Loader** returns with the Memory Handle of the memory block.

Note that since Load Segments in a Load File are numbered sequentially starting at 1, to find Load Segment 5, the **System Loader** must scan through the first 4 Load Segments before finding Load Segment 5. Each Load Segment Header must be processed because Load Segments as well as Load Segment Header are variable length. It is simpler than it sounds because Load Segments start on block boundaries and the number of blocks in each Load Segment is the first field in the

Segment Header. The following logic sample will load the first block of a specified Load Segment:

```
segnum:=Load Segment number;
fileid:=Load File;
open(fileid);
block:=0;
for i:=1 to segnum do
  begin
    seek(fileid,block);  {find block}
    get(fileid);          {read block}
    block:=block+fileid^.BLKCNT; {add BLKCNT to  block}
  end;
```

UnloadSegNum (Unload Segment by Number)

Function Number:	$0C
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
Output:	none
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will unload a specific Load Segment that is currently in memory.

The **System Loader** searches the **Memory Segment Table** for the "Load File Number" and "Load Segment Number". If there is no such entry, error $1101 is returned.

Next the **Memory Manager** is called to make the memory block purgeable using the Memory Handle in the table entry.

All entries in the **Jump Table** referencing the unloaded segment are changed to their "unloaded" states.

If the input UserID is 0, the UserID of the current user (in USERID) is assumed.

If both the Load File Number and the Load Segment Number are specified, the specific Load Segment is made purgeable whether it is static <u>or</u> dynamic. Note , if a static segment is unloaded, the application can not be ReStarted. If either input is 0, only dynamic segments will be made purgeable.

If the input Load Segment Number is 0, all dynamic segments in the specified Load File are unloaded.

If the input Load File Number is 0, all dynamic segments for the UserID are unloaded.

LoadSegName (Load Segment by Name)

Function Number:	$0D
Input:	UserID (2 bytes)
	Address of Load File Name (4 bytes)
	Address of Load Segment Name (4 bytes)
Output:	Address of segment (4 bytes)
	Load File Number
	Load Segment Number
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1104 - File not Load File
	$1105 - System Loader is busy
	$1107 - File Version error
	$1109 - SegNum out of sequence
	$110A - Illegal load record found
	$110B - Load Segment is foreign
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will load a named Load Segment into memory.

The Load File type is checked. If it is not a Load File (types $B3-$BE), error $1104 is returned.

ProDOS is then called to open the specified Load File. If ProDOS has a problem, its error code is returned.

Next the Load File is searched for a Load Segment corresponding to the specified "Load Segment Name". If there is no segment with Segment Name requested, error $1101 is returned.

Now that the **System Loader** has located the requested Load Segment (and knows the Load Segment Number), it checks the **Pathname Table** to see whether the Load File is represented. If so, it uses the File Number from the table. Otherwise, the **System Loader** adds a new entry to the **Pathname Table** with an unused File Number.

Next the **System Loader** attempts to load this Load Segment by calling the Load Segment by Number function. If the Load Segment by Number function returns an error, the Load Segment by Name function, in turn, returns this error. If the Load Segment by Number function is successful, the Load Segment by Name function returns the Load File Number, the Load Segment Number and the Memory Address of the segment in memory.

UnloadSeg

Function Number:	$0E
Input:	Address in Segment (4 bytes)
Output:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will unload the Load Segment which contains the specified address.

The **Memory Manager** is called to locate the memory block containing the specified address. If no memory block contains the address, error $1101 is returned. The UserID associated with the Handle of the memory block returned by the **Memory Manager** is extracted (from the **Memory Manager**'s internal table). The **Memory Segment Table** is scanned looking for the UserID and Handle. If an entry is not found, error $1101 is returned.

If the entry in the **Memory Segment Table** is for a Jump Table Segment, the specified address should be pointing to the **Jump Table** entry for a dynamic segment reference. The Load File Number and Segment Number of the **Jump Table** entry are extracted.

If the entry in the **Memory Segment Table** is not for a Jump Table Segment, the Load File Number and Segment Number of the **Memory Segment Table** entry are extracted.

The UnloadSegNum function is now called to actually unload the segment. The outputs of this function can be used as input to other **System Loader** functions.

GetLoadSegInfo

Function Number:	$0F
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
	Address of User Buffer (4 bytes)
Output:	filled User Buffer
Errors:	$0000 - Operation succcessful
	$1101 - Entry not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will return the Memory Segment Table entry corresponding to the specified Load Segment.

The Memory Segment Table is searched for the specified entry. If the entry is not found, error $1101 is returned. If the entry is found, the contents except for the link pointers are moved into the User Buffer.

GetUserID

Function Number:	$10
Input:	Address of Pathname (4 bytes)
Output:	UserID (2 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Entry not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will search the Pathname Table for the specified Pathname. If a match is found, the corresponding UserID is returned. If the input Pathname does not start with prefix 1/ or 2/, it is first expanded to a full pathname before the search. A **Controlling Program** can use this function to determine whether to perform a Restart of an application or an Initial Load.

GetPathname

Function Number:	$11
Input:	UserID (2 bytes)
	File Number (2 bytes)
Output:	Address of Pathname (4 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Entry not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will search the Pathname Table for the specified UserID and File Number. If a match is found, the address of the Pathname in the Pathname Table is returned. ProDOS uses this call to get the pathname of an existing application so that it can set the Application prefix before restarting it.

UserShutdown

Function Number:	$12
Input:	UserID (2 bytes)
	Restartable Flag (2 bytes)
Output:	UserID (2 bytes)
Errors:	$0000 - Operation succcessful
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function is called by the **Controlling Program** to close down an application which has just terminated. If the UserID specified is 0, the current UserID (USERID) is assumed.

If the Restartable Flag is FALSE (0), all Memory Blocks for the UserID (with the AuxID set to 0) are purged and the Cleanup Routine is called to purge the **System Loader**'s internal tables of the UserID. The application can not be Restarted.

If the Restartable Flag is TRUE (not 0), the **Memory Manager** is called to purge all Dynamic Segments. The **Memory Manager** is again called to make all the Static Segments purgeable for the specified UserID. The application is now in a "zombie" state and can be resurrected by the **System Loader** very quickly because all the static segments are still in memory. However, as soon as any one static segment is purged by the **Memory Manager** for whatever reason, the **System Loader** must reload the application from its original Load File.

Jump Table Load

Function Number:	none
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
	Load Segment Offset (4 bytes)
Output:	none
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1104 - File not Load File
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function is called by an "unloaded" **Jump Table** entry to load a Dynamic Load Segment.

This function calls the Load Segment by Number function with the the Load File Number and Load Segment Number. If any errors occurred, the **System Loader** will report a System Death.

If the Load Segment by Number function has sucessfully loaded the segment, the **Jump Table** entry is made "loaded" by replacement of the JSL to the Jump Table Load function with a JML to the absolute address of the reference in the Dynamic Load Segment.

The **System Loader** will now transfer control to the absolute address.

Cleanup Routine

Function Number:	none
Input:	UserID
Output:	none
Errors:	none

This function is internal to the **System Loader.** Its function is to cleanup the **System Loader**'s internal tables in order to free memory.

If the UserID is 0, the **Memory Segment Table** is scanned and all dynamic segments for all UserID's will be purged.

If the UserID is not 0, all Load Segments (both dynamic and static) for that UserID will be purged. In addition, all entries for the UserID in the **Memory Segment Table** and **Pathname Table** will be removed and the UserID itself will be deleted.

Error Codes

$0000	Operation successful
$1101	Segment /Application/Entry not found
$1102	not used
$1103	not used
$1104	File is not a Load File
$1105	Loader is busy
$1106	not used
$1107	File version error
$1108	UserID error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Segment is foreign

References

"Object Module Format ERS" by Lou Infeld -- Apple Computer
"ProDOS ORCA/M User's Guide" -- The Byte Works
"Cortland Development Environment Core" -- The Byte Works
"The Tool Locator ERS", by Steve Glass -- Apple Computer