Date: August 6, 1986

Author: Lou Infeld

Subject: Object Module Format ERS

Document Version Number: 00:93

Revision History

00:00	(08/07/85)	Initial Release
00:01	(09/06/85)	Support for multi-file loading added Library Files and Run Time Library Files defined Pathname Table defined File types changed Segment Header format changed Segment Jump Table format changed VERSION in Segment Header changed to 1 Segment Jump Table is no longer Load Segment 0 References to Modules eliminated
00:02	(09/16/85)	Segment Jump Table format changed File # added to INTERSEG record
00:03	(09/30/85)	Figure 2 renumbered
00:04	(11/15/85)	Tool Locator interface changed Segment Number added to Segment Header Library Files redefined Source File type changed INTERSEG record discussion expanded Load File Table changed to Pathname Table ENTRY ($F4) record defined Library Segment Dictionary (KIND $04) defined Optional Initialization Segment (KIND $05) defined Pathname Table format changed
00:50	(01/03/86)	KIND values changed Load Segments start with 0 rather than 1 Position Independent Segment support added Segment Jump Table entry changed Binary Bit operators defined
00:60	(03/10/86)	RELOC record changed Pathname support changed CONST record changed Binary Bit operators changed
00:70	(04/15/86)	Binary Bit NOT added Default address in RELOC record changed Relative Offset Operand given a code ($87) Files given catalog mnemonics Fields added to Segment Header KIND field in Segment Header changed

		Private Segment is defined
		GLOBAL and GEQU records have new field
		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
		ORG record no longer contains an expression
		INTERSEG record explanation improved
		Bit NOT operator separated from binary bit operators
		Pathname Table description expanded
		Startup Load Files defined
		System prefixes 0,1 and 2 defined
		$B3 file name changed to S16
00:90	(05/27/86)	Pathname Table rename Pathname Segment and redefined
		Absolute Bank Segment defined
		Zero Page/Stack Segment defined
		Segment Header figure enhanced
00:91	(06/05/86)	Shell Load Files are further described
		File types of Load Files expanded
		Segment Jump Table description changed
		INTERSEG record changed
		RELOC record description enhanced
00:92	(06/24/86)	CBANKSIZE changed to BANKSIZE
		DBANKSIZE removed
		New description of Library Files
		New description of Shell Load Files
		Load Files can contain DS records
		Library Segment Dictionary renamed Library Dictionary Segment
		Zero Page/Stack Segment renamed Direct Page/Stack Segment
		Segment Jump Table renamed Jump Table
		Segment Jump Table Segment renamed Jump Table Segment
00:93	(08/06/86)	Description of "loaded" state of Jump Table entry changed
		ENTRY record changed
		Compressed RELOC and INTERSEG records defined
		Description of RELOC improved

Index

History

Under ProDOS on the Apple // there is one loadable file format. It is the binary file format which consists of one absolute memory image along with its destination address. The file is pure binary data representing a contiguous block of memory. For System files, the destination address is $2000. For binary files, the destination address is in the directory of the file entry. This format is clearly inadequate for the more sophisticated Development Environment planned for Cortland. The major goal of the Cortland Development Environment is to allow software development in any of several languages and assemblers, all producing compatible code files which could be linked together. Therefore, we need a more general format that allows relocation and supports the various needs of many languages and assemblers.

We investigated many existing Object Module formats and found that the ORCA/M ProDOS Object Module format was extremely flexible and supported fairly powerful language features. However, it didn't allow program Segmentation and of course, the ORCA/M Linker only generated binary code files. Together with Mike Westerfield, we have come up with an upgraded version of the ORCA/M Object Module format that will provide the Cortland Development Environment with the ability to support a wide variety of programming languages as well as support a dynamic relocatable Segment Loading facility.

The sequential relocation format of the ORCA/M Object Module is excellent for compilers, assemblers and linkers. However, tests have shown that this format is too slow for use by a Relocating Loader. So the ORCA/M Object Module Format has been extended to support a more traditional relocation dictionary that will speed up the loading process. We therefore have one Object Module Format that supports language, linker, library and loader requirements that is extremely flexible, easy to generate and fast to load.

Overview

This ERS defines four kinds of files: Object Files, Library Files, Load Files and Run Time Library Files.

Object Files are the output from an assembler or compiler and the input to a linker. Object Files must be fast to process, easy to create, independent of the source language, and be able to support libraries in a convenient way. In the Cortland development environment, Object Files will additionally support Segmentation of code and partial assemblies and compiles. They will also support absolute as well as relocatable program Segments.

Library Files are files containing general program Segments that a linker can search. Only code needed during the link process is extracted from the Library File.

Load Files are the output of a linker and contain memory images which a loader will load into memory. Load Files must be very fast to process. In the Cortland development environment, Load Files will additionally be relocatable and support automatic overlay handling.

Run Time Library Files are Load Files that contain general utilities which can be loaded as needed by a loader and shared between applications.

All four types of files are actually files consisting of individual components called Segments in Object Module Format. However, each uses a subset of the full Object Module Format. Additionally, different compilers and assemblers will probably use subsets of the format depending on the requirements and richness of the language.

Object Files, Library Files, Load Files and Run Time Library Files will be distinguished by their file types:

$B1 - for Object Files	(OBJ)
$B2 - for Library Files	(LIB)
$B3 - for Load Files	(S16)
$B4 - for Run Time Library Files	(RTL)
$B5 - for Shell Load Files	(EXE)

and Source Files (SRC) will be type $B0 and their auxiliary type will represent the language under which it is to be used. File types $B6-$BE are currently reserved for other types of Load Files such as boot time startup files, desk accessories, tools, etc.

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.

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.

Shell Load Files are Load Files which can only be run from a Shell.

Startup Load Files are Load Files which ProDOS loads during its startup (e.g. Patch Files).

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 that contains program code that is usually instructions.

A **Data Segment** is a Segment that contains program code that is usually data.

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

Object Module Format

Design Goals

o Independent of source language
o Simple and fast to generate by language
o Simple and fast to process by linker
o Support partial assemblies and compilations
o Support libraries
o Support a wide variety of language facilities
o Support very general relocation information
o Support source debugging
o Support loadable format
o Simple and fast to process by loader
o Conform to an existing format

General

The present ORCA/M 4.0 Object Module format fulfills most of the above goals and the proposed Cortland Object Module Format will fulfill all these goals.

Each OMF file will contain one or more Segments. Figure 1 represents the structure of an OMF file. Each Segment in an Object File is a separate relocatable entity that contains all the information needed to link it with other Segments. Each Segment in an Load File is a separate relocatable entity that contains all the information needed to load it into memory.

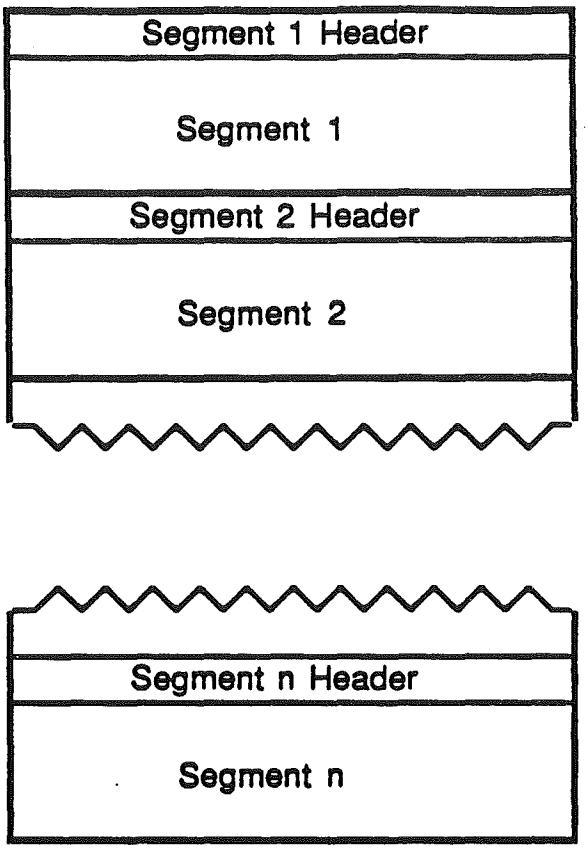

Figure 1 -- OMF File

Each Segment contains a set of records which indicate relocation information or code images. If the file is an Object File, the **Linker** will process each record and generate a Load File containing Load Segments. In this case, relocation information is embedded within the code data. If the file is in a Load File, the **System Loader** will load the code image in the Segment and then process the relocation information which follows.

Segments in Object Files can be combined by the **Linker** into a single Segment in the Load File. For instance, each subroutine in a program can be placed in separate Code Segments so that each subroutine can be separately compiled. The **Linker** can be told to place all the Code Segments into one Load Segment.

Segment Types

To support languages that distinguish program code from data, the Object Module Format defines both Code Segments and Data Segments. Additionally, since the Object Module Format supports dynamically loadable Segments, it defines the concept of Static Segments and Dynamic Segments. Static Segments are to be

loaded at program execution time and are not to be unloaded during execution. Dynamic Segments are loaded and unloaded during program execution as needed.

There are also several special types of Segments: Jump Table Segment, Pathname Segment, Library Dictionary Segment, Initialization Segment, Private Segment, Absolute Bank Segment and Direct Page/Stack Segment.

Segment Header

Each Segment in a OMF file contains a Segment Header that contains all the general information about the Segment, such as its name and length. Segment Headers allow the **Linker** to quickly scan an Object File for the desired Segments and allows the **System Loader** to load individual Load Segments. The format of the Segment Header is illustrated in Figure 2. Following is a detailed description of each of the fields in the Segment Header.

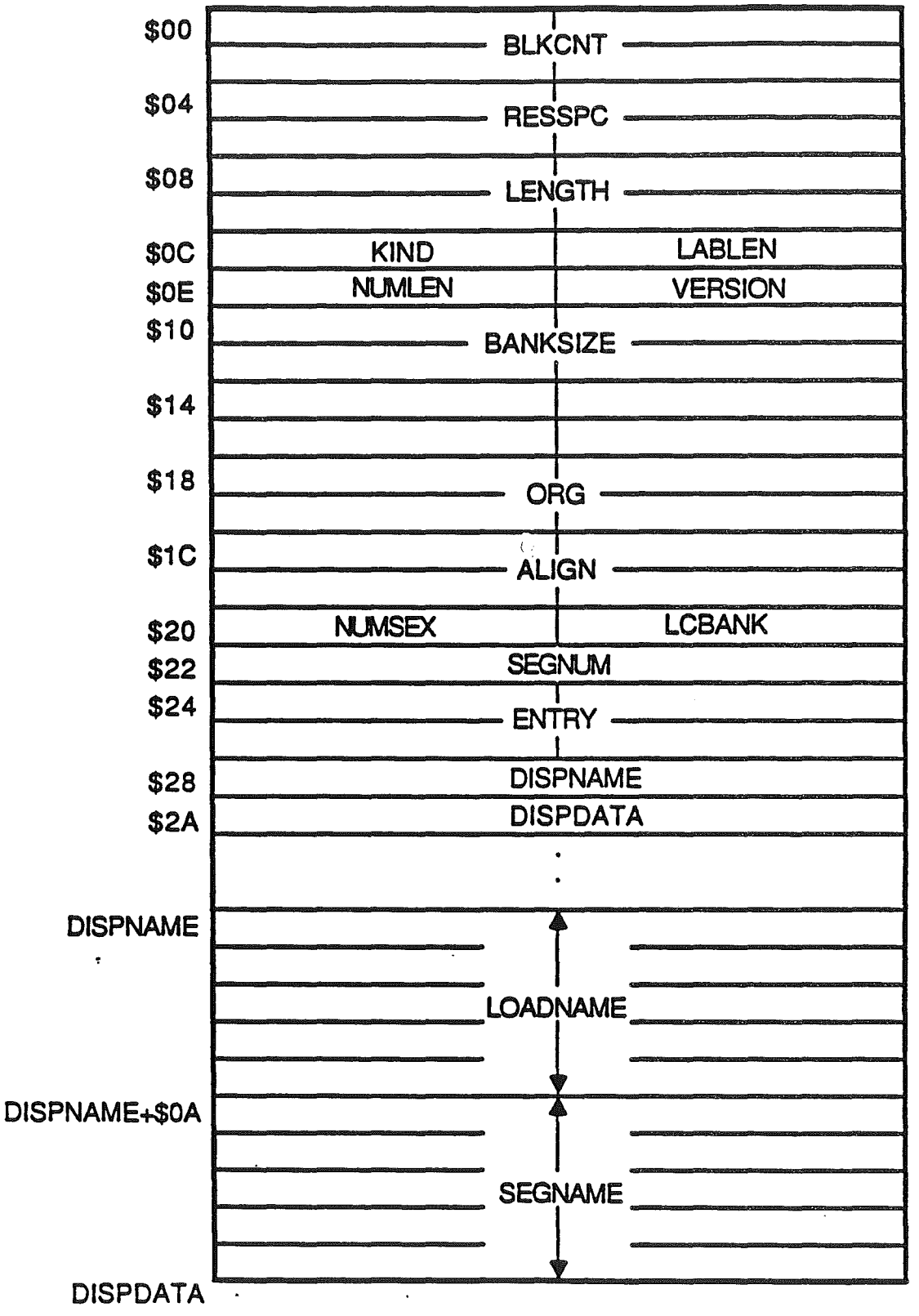

Figure 2 -- Segment Header

BLKCNT -- This 4 byte field is the number of file blocks that the Segment requires in the File where each block is 512 bytes. Note that the Segment Header is part of the first block of the Segment and Segments in the File start on file block boundaries.

RESSPC -- This 4 byte field is the number of zero bytes to add to the end of the Segment. Using this field can significantly reduce the block size of a Segment that ends with a DS for a large amount of memory. (This field may not be needed.)

LENGTH -- This 4 byte field is the memory size that the Segment will require when loaded. It includes the extra memory specified by **RESSPC**.

KIND -- This 1 byte field specifies the type and attributes of the Segment that follows.

Bits 0-4 - type where

$00	- Code Segment
$01	- Data Segment
$02	- Jump Table Segment
$04	- Pathname Segment
$08	- Library Dictionary Segment
$10	- Initialization Segment
$11	- Absolute Bank Segment
$12	- Direct Page/Stack Segment

Bit 5 - Position Independent (1=yes)
Bit 6 - Private (1=yes)
Bit 7 - Static/Dynamic (0=Static, 1=Dynamic)

Attributes can be combined with a specific type. For example a Dynamic Initialization Segment has KIND=$90. A Private Code Segment is an object code segment whose name is only available to other object code segments within the same Object File. A Private Data Segment is a object data segment whose labels are available only to other object code segments within the same Object File. Absolute Bank Segments are relocatable within a specified bank. Direct Page/Stack Segments are used to preset the Direct and Stack registers for an application.

LABLEN -- This 1 byte field indicates how long each label field will be. If this field is 0, it indicates that each label field is of variable length with the length specified in the 1st byte. Note that this field also determines the length of the **SEGNAME** field of the Segment Header. However, the **LOADNAME** field will always be of length 10. Labels are always left justified and padded with blanks.

NUMLEN -- This 1 byte field indicates how many bytes each number field will be.

VERSION -- This 1 byte field indicates the version number of the Object Module Format with which the Segment is compatable. This field should be 1 for the initial specification of the Object Module Format.

BANKSIZE-- This 4 byte field indicates the maximum bank size for the Segment. If this Segment is in an Object File, the **Linker** will assure that the Segment is not larger than this value. If this Segment is in a Load File, the **System Loader** will assure that this Segment is loaded into a memory segment that does not cross this boundary. For a 64K bank boundary, this field should be $00010000 (this is the case for Code Segments on Cortland). A value of 0 in this field indicates that the Segment can cross bank boundaries.

ORG -- This 4 byte field indicates the absolute address where this Segment is to be loaded in memory. Normally, this field is 0 indicating that the Segment is relocatable and can be loaded anywhere in memory.

ALIGN -- This 4 byte field indicates whether this Segment must be aligned to a particular boundary. For example, if the Segment is to be aligned on page boundary, this field should be $00000100. If the Segment is to be aligned on a bank boundary, this field should be $00010000. A value of 0 indicates no alignment is needed.

NUMSEX -- This 1 byte field indicates the order of the bytes in a number field. If this field is 0, the least significant byte is first (this is the case for Cortland). If this field is 1, the most significant byte is first.

LCBANK -- This 1 byte field indicates in which bank in the Language Card the segment is to be loaded (0=bank 1, 1=bank 2). This field is only meaningful if the ORG field is an address in the Language Card area of banks 0,1,E0 or E1. Note, the **System Loader** does not support loading segments into the alternate bank of the Language Card.

SEGNUM -- This 2 byte field is the Segment number. It should correspond to the relative position of this Segment in the File (starting at 1). This field is used by the **System Loader** as a check while searching for a specific Load Segment in a Load File.

ENTRY -- This 4 byte field indicates the offset into the segment which corresponds to the Entry Point of the segment.

DISPNAME -- This 2 byte field indicates the displacement within this Segment Header corresponding to the LOADNAME field. Currently the value of this field is 44.

DISPDATA -- This 2 byte field indicates the displacement within this Segment Header corresponding to the start of the Segment Body. Currently the value of this field is 54+LABLEN.

LOADNAME -- This field is the name of the Load Segment which will contain the code generated by the **Linker** for this Segment. Note that more than one Segment in a Object File can be merged by the **Linker** into a single Segment in the Load File. The length of this field is always 10 bytes long.

SEGNAME -- This field is the name of the Segment. It 's length is determined by the LABLEN field.

DISPNAME should be used when referencing LOADNAME and SEGNAME and DISPDATA should be used when referencing the start of the Segment Body so that future expansion of the Segment Header will not affect existing software. LOADNAME and SEGNAME will always be the last two fields in the Segment Header but new fields may be added after the DISPNAME and DISPDATA fields.

Segment Body

The body of each Segment is made up of sequential records, each of which starts with a unique one byte operation code. These records contain either program code or information for the **Linker** or **System Loader**. All names and labels included in these records are LABLEN bytes long. All numbers, offsets and addresses are NUMLEN bytes long unless otherwise specified. They also obey the NUMSEX convention specified. Some records have an expression syntax which has to be evaluated by the **Linker**. This expression facility provides an extremely flexible Linker language. Following is a detailed description of each of the defined Segment records.

END ($00) - This record indicates the end of the Segment.

CONST ($01-$DF) - This record contains absolute data that needs no relocation. The operation code specifies how many bytes of data follows.

ALIGN ($E0) - This record contains a number that indicates an alignment factor. The **Linker** will insert as many zero bytes as is necessary to move to the corresponding memory boundary indicated by this factor. The value of this factor is in the same format as the ALIGN field in the Segment Header and can not be greater.

ORG ($E1) - This record contains a number with which the present location counter is to be incremented or decremented. If the new location counter is greater than the old location counter, zeros have to be inserted to get to the new address. If the new location counter is before the old location counter, but after the start of the Segment, the location counter is changed but subsequent code overwrites the old code.

RELOC($E2) - This is a relocation record used in the relocation dictionary of a Load Segment. It is used to patch an address in a Load Segment with a reference to another address in the same Load Segment. It contains two one byte counts followed by two offsets. The first offset is the starting byte to patch relative to the start of the segment. The second offset is the location of the reference relative to the start of the segment. The first count is the number of bytes to be relocated and the second count is a bit shift operator, telling how many times to shift the relocated address before inserting the result into memory.

For example, suppose the Linker comes across an instruction like this:

```
LDA  LAB|4
```

where the label "LAB" is a reference to a location in the same Load Segment. If this instruction is at relative location $100 in the Load Segment and LAB is at relative location $200 in the Load Segment. The RELOC Load Segment record would look like this:

```
$E2
$02          - 2 bytes to be generated (address)
$04          - address to be shifted left 4 bits
$00000101    - relative location of address portion of instr
$00000200    - relative location of LAB
```

INTERSEG($E3) - This is an inter-segment relocation record used in the relocation dictionary of a Load Segment. It is used to patch an address in a Load Segment with a reference to another address in a different Load Segment. It contains two one byte counts followed by an offset which is the relative location in the current segment of the instruction to be patched. Next is a two byte file number, a two byte segment number and an offset which correspond to the location of an external reference. The first count is the number of bytes to be relocated and the second count is a bit shift operator, telling how many times to shift the relocated address before inserting the result into memory. This record is used for Static as well as Dynamic Segment references. For Static Segments, the external reference information is that of the Static Segment. But for Dynamic Segments, the external reference information is for the Jump Table entry that corresponds to the Dynamic Segment address.

For example, suppose the Linker comes across an instruction like this:

```
JSL  EXT
```

where the label "EXT" is a reference to a location in a Static Segment. If this instruction is at relative address $100 within its segment and "EXT" is at relative address $200 in segment $0005 in file $0002. The INTERSEG Load Segment record would look like this:

$E3
$03
$00
$00000101
$0002
$0005
$00000200

If the external reference is in a Dynamic Segment, the inter-segment reference is to the **Jump Table** entry instead of the actual Dynamic Segment. Using the same example as above, suppose "EXT" is in a Dynamic Segment. The **Jump Table** entry that corresponds to the global location "EXT" would look like this:

$0000
$0002
$0005
$00000200
Call to **System Loader**

If the **Jump Table Segment** is in File 1, Segment 1 and the above Call to the **System Loader** is at relative location $1100 in the **Jump Table Segment**, the INTERSEG record would look like this:

$E3
$03
$00
$00000101
$0001
$0001
$00001100

INTERSEG records are used with any long address reference to a Static Segment, but are restricted to JSL instructions to Dynamic Segments.

See the discussion of the **Jump Table Segment** later in this document.

USING ($E4) - This record contains the name of a Data Segment. After this record is encountered, local labels from that Data Segment can be used in this Segment.

STRONG ($E5) - This record contains the name of a Segment which needs to be included during linking even if no external references have been made to it.

GLOBAL ($E6) - This record contains the name of a global label followed by three 1 byte attribute fields. The label is assigned the value of the current location counter. The first attribute byte is the number of bytes generated by the line that defined the label. The second attribute byte is the type of operation in the line that

defined the label. The third attribute byte is the private flag (1=private). The ORCA Assembler generates the following type attributes:

A	Address Type DC Statement
B	Boolean Type DC Statement
C	Character Type DC Statement
D	Double Precision Floating Point Type DC Statement
F	Floating Point Type DC Statement
G	EQU or GEQU Statement
H	Hexadecimal Type DC Statement
I	Integer Type DC Statement
K	Reference Address Type DC Statement
L	Soft Reference Type DC Statement
M	Instruction
N	Assembler Directive
O	ORG Statement
P	ALIGN Statement
S	DS Statement
X	Arithmetic Symbolic Parameter
Y	Boolean Symbolic Parameter
Z	Character Symbolic Parameter

GEQU ($E7) - This record contains the name of a global label followed by three 1 byte attribute fields and an expression. The label is given the value of the expression. The three atribute fields are defined in the GLOBAL record description above.

MEM ($E8) - This record contains two numbers which represent the starting and ending addresses of a range of memory that must be reserved.

EXPR ($EB) - This record contains a 1 byte count followed by an expression. The expression is to be evaluated and its value is truncated to the number of bytes specified in the count. The order of the truncation is most significant to least significant.

ZEXPR ($EC) - This record is the same as EXPR except that if any bytes are truncated, the Linker must check and make sure they are 0.

BEXPR ($ED) - This record is the same as EXPR except that any bytes truncated must match the corresponding bytes of the location counter. This allows the Linker to make sure an expression evaluates to an address in the current bank.

RELEXPR ($EE) - This record is used to generate relative branch values that involve external locations. For example, a "BNE LOC" instruction where LOC is external will generate this record. The record contains the following components:

Length - 1 byte
Offset - NUMLEN bytes
Expression - variable length

The "Length" indicates how many bytes to generate for the instruction. The "Offset" indicates where the origin of the branch is relative to the current location counter (1 for Cortland). The "Expression" indicates the destination of the branch.

LOCAL ($EF) - This record is similar to the GLOBAL record except the label in the record is local. The **Linker** ignores local labels in Code Segments and only recognizes local labels in Data Segments if a USING record has been processed for that Segment. However, a symbolic debugger would use all of these records.

EQU ($F0) - This record is similar to the GEQU record except the label in the record is local.

DS ($F1) - This record contains a number indicating how many bytes of zero to insert at this location counter.

LCONST ($F2) - This record contains absolute data. It is similar to the CONST records except allows for a much greater number of bytes. The data bytes are preceded by a number indicating the number of bytes.

LEXPR ($F3) - This record is similar to the EXPR record except that:

1) if the expression evaluates to a single label with a fixed, constant offset AND
2) that label is in another Segment AND
3) that Segment is a dynamic Code Segment

then the **Linker** must create an entry for that label in the **Jump Table Segment**. The **Jump Table Segment** is defined in the description of Load Files. It provides a mechanism to allow dynamic loading of Segments as they are needed. Only a JSL instruction should generate this record.

ENTRY ($F4) - This record is used in the Run Time Library Entry Dictionary. It consists of a 2 byte number and an offset followed by a label. The label is a code segment name or entry. The number is the Segment Number and the offset is the relative location withing the segment of the label.

cRELOC ($F5) - This is the compressed version of the RELOC record. It contains 2 byte offsets rather than 4 byte. It can only be used if both offsets are less than $FFFF (65535).

cINTERSEG ($F6) - This is the compressed version of the INTERSEG record. It contains 2 byte offsets rather than 4 byte and also does not have the 2 byte File

Number. It can only be used if both offsets are less than $FFFF (65535) and the File Number associated with the reference is 1 (i.e. Initial Load File). References to Run Time Library Segments must use the normal INTERSEG records.

Expressions

Several of the Object Module Format records contain expressions. Expressions form a reverse-polish stack language which can be used to specify extremely flexible classes of data generation.

Each expression is made up of a series of operators and operands. The operands represent some form of value, like a constant or label, which must be loaded onto an evaluation stack.

Operators take one or two values from the evaluation stack, perform some mathematical or logical operation on them, and place a new value onto the evaluation stack. The final value on the evaluation stack is used as if it were a single value in the record. Note that the evaluation stack mentioned above is purely a programming concept and does not relate to any hardware stack on the computer.

Termination Operator

All expressions end with the termination operator 0.

Binary Math Operators

Binary math operators take two numbers as signed integers from the top of the evaluation stack, perform the operation using the two numbers and place the single integer result back on to the evaluation stack.

$01 - Addition	(+)
$02 - Subtraction	(-)
$03 - Multiplication	(*)
$04 - Division	(/)
$05 - Integer Remainder	(MOD)
$07 - Bit Shifting	

The bit shift operator shifts the first number by the number of bit positions specified by the second number. If the second number is positive, the first number is shifted to the left, filling with zeros in the vacated bit positions (logical shift). If the second number is negative, the first number is shifted right, preserving the sign bit as it does so (arithmetic shift).

Unary Math Operators

Unary math operators take the number as a signed integer from the top of the evaluation stack, perform the operation on it and place the integer result back on the evaluation stack.

$06 - Negation (-)

Comparison Operators

Comparison operators take two numbers from the top of the evaluation stack, perform the comparison using the two numbers and place the single integer result back on to the evaluation stack. Each operator compares the number at the Top Of Stack (TOS) -1 with the number at TOS. If the comparison is true, a 1 is placed on the stack; otherwise, a zero is placed on the stack.

$0C - Less than or equal (<=)
$0D - Greater than or equal (>=)
$0E - Not equal (<> or !=)
$0F - Less than (<)
$10 - Greater than (>)
$11 - Equal to (= or ==)

Binary Logical Operators

Binary logical operators take two numbers as boolean values from the top of the evaluation stack, perform the operation using the two numbers and place the single boolean result back on to the evaluation stack. Boolean are defined as numbers having the value 0 for FALSE or any non-zero value for TRUE. Logical operators always return 1 for TRUE.

$08 - AND (Logical AND)
$09 - OR (Inclusive OR)
$0A - EOR (Exclusive OR)

Unary Logical Operators

Unary logical operators take the number as a boolean from the top of the evaluation stack, perform the operation on it and place the boolean result back on the evaluation stack.

$0B - NOT (Complement)

Binary Bit Operators

Binary bit operators take two numbers as binary values from the top of the evaluation stack, perform the operation using the two numbers and place the single binary result back on to the evaluation stack. The operations are performed on a bit by bit basis.

$12 - Bit AND
$13 - Bit OR

$14 - Bit EOR

Unary Bit Operators

Unary bit operators take the number as a binary value from the top of the evaluation stack, perform the operation on it and place the binary result back on the evaluation stack.

$15 - Bit NOT

Location Counter Operand

The location counter operand ($80) loads the value of the current location counter onto the top of the stack. Note that the location counter is loaded before the bytes from the expression are placed into the code stream, so this is the value of the location counter before the expression is evaluated.

Constant Operand

The constant operand ($81) is followed by a number which is loaded on the top of the stack.

Label Reference Operands

Operand codes $82 to $86 are all followed by the name of a label.

For operand code $83, the value assigned to that label is placed on the top of the stack.

For operand code $84, the length attribute of the label is placed on the top of the stack.

For operand code $85, the type attribute of the label is placed on the top of the stack.

See the discussion of label attributes in the description of the GLOBAL Segment record.

For operand code $86, the count attribute is placed on the top of the stack. The count attribute is 1 if the label is defined and 0 if it is not.

The operand code $82 is the "weak" reference. The weak reference is an instruction to the **Linker** that asks for the value of a label if it exists. It is not an error if the **Linker** cannot find the label. However, the **Linker** will not load a segment from a library if only weak references to it exist. If a label does not exist, a 0 is loaded onto the top of the stack instead of the label's value. This capability is generally used for creating jump tables to library routines that may or may not be needed in a particular program.

Relative Offset Operand

The segment relative operand ($87) is followed by a number which is treated as a displacement from the start of the segment. Its value is added to the value of the location counter when the segment started, and the result is loaded on the top of the stack.

Library Files

Library Files contain Object Segments which the **Linker** can search for external references. Any Object Segment, which contains a global definition which was referenced during the link process, will normally be extracted from the Library File and added to the output Load Segment that the **Linker** is currently creating.

Library Files are of type $B2 and consist of a Library Dictionary Segment (KIND=$08) followed by one or more object segments. The body of the Library Dictionary Segment consists of three LCONST records followed by an END record. The three LCONST records contain:

1. File names
2. Symbol table
3. Names for symbol table

The File names record is broken up into one or more sub-records each consisting of a two byte file number followed by a file name in ProDOS format: a length byte followed by the ASCII characters.

The Symbol table record consists of a sub-record for each global symbol:

Byte	Length	Description
$00	4	Disp into name record to start of symbol name
$04	2	File number of the file that the name occurred in
$06	2	Private flag. If true, the name is valid only in the file where it occurred
$08	4	Disp to the first byte of the object segment in which the symbol occurs

Note that the Symbol table record contains no actual symbols but offsets into the Names record that follows.

The Names record consists of a series of symbol names each with a length byte followed by up to 255 ASCII characters.

All global symbols that appear in an object segment are placed in the dictionary. This includes entry points and global equates. Duplicate symbols are not allowed. Even if the symbol appears inside the segment, the displacement is to the start of the segment. The information needed is a segment for the linker to link - exact locations within the segment are resolved by the linker.

The structure of a Library File is slightly different than an object file. In a library file Segments are not aligned to 512 byte boundaries. Instead of a BLKCNT, Segment Headers in a Library File contain a BYTECNT.

Library Files are created from corresponding Object Files by a utility program called "MakeLib"

Load Files

Load Files contain the Load Segments which are moved by the **System Loader** into memory. Although they conform to the Object Module Format, they are restricted to a small subset of that format. This is due to the need to quickly relocate and load the Load Segments. They can not contain any unresolved symbolic information.

The Segment of KIND=$02 in a Load File is the **Jump Table Segment.** It contains the actual calls to the **System Loader** to load Dynamic Segments. Each time the **Linker** comes across a JSL to a label in a Dynamic Segment, it generates an entry in the **Jump Table Segment** for this label and replaces the label information with the corresponding segment information.

The Segment of KIND=$04 in a Load File is the **Pathname Segment.** It contains a cross reference between File Numbers and Pathnames which the **System Loader** will need to reference Load Segments.

The Segment of KIND=$10 in a Load File is an Initialization Segment. During the initial loading of static segments, the **System Loader** will not only load Initialization Segments, but will transfer control to them to perform any initialization required by the Applications program. After an Initialization Segment returns control back to the **System Loader**, the **System Loader** continues the normal Inital Load of all the other Static Segments in the Load File. Note that if a program is ReStarted by the **System Loader**, the Initialization Segment will not be executed again.

One use of an Initialization Segment is to initialize the graphics environment used by an Application and to display a "Splash Screen" for the duration of the complete program load. This type of segment must obey several rules:

1. it must not reference any other segments (no INTERSEG records)
2. it must exit with a "RTL".

The format of each Load Segment is basically a loadable binary image followed optionally by a relocation dictionary. The loadable binary image is a Long Constant (LCONST) record. The relocation dictionary will contain Relocation (RELOC) records and Intersegment (INTERSEG) records only. DS records to zero fill memory are also supported.

Load Segments are numbered by their relative location in the Load File where the first Load Segment is number 1. They also contain a Segment Number in their headers. These are included as a check for the Loader.

Jump Table Segment

The **Jump Table Segment** is the Load Segment of type (KIND) $02 in a Load File. This table is created by the **Linker** to allow dynamic loading of Code Segments as they are needed during program execution. The **System Loader** maintains a list of **Jump Table Segments** in memory. This list is called the **Jump Table List** or **Jump Table** for short.

Each entry in the **Jump Table** corresponds to a JSL call to an external (inter-segment) routine in a Dynamic Segment. The **Jump Table** initially contains entries in the unloaded state. When the JSL instruction is encountered during program execution, a jump to the **Jump Table** occurs. The code in the **Jump Table** entry in turn jumps to the **System Loader**. The **System Loader** figures out which segment is referenced and loads it. After the **System Loader** loads the referenced segment, it changes the entry in the **Jump Table** to the loaded state. The entry stays in the loaded state as long as the corresponding segment is in memory. If the **System Loader** unloads a segment, all **Jump Table** entries that reference that segment are changed to their unloaded states.

The unloaded state of a **Jump Table** entry contains the actual code to call the **System Loader** to load the needed Segment. A typical entry will look like this:

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. A Load File Number of 0 indicates that there are no more entries in this **Jump Table Segment.**

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 Segment

The **Pathname Segment** is the Load Segment of type (KIND) $04 in a Load File. This table is created by the **Linker** to help the **System Loader** find the Load Segments of Run Time Libaries that need to be loaded dynamically. It provides a cross reference between File Numbers and File Pathnames.

The **Pathname Segment** contains one entry for each Run Time Library File referenced by the rest of the Load Segments. The format of each entry is:

File Number (2 bytes)
File Date (2 bytes)
File Time (2 bytes)
Pathname (Pascal string)

where:

"File Number" is a number assigned by the **Linker** for a specific Load File. File number 1 is reserved for the Load File in which the **Pathname Segment** resides (usually the application Load File). A File Number of 0 indicates that there are no more entries in this segment.

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 same Run Time Library File loaded at execution time.

The Pathname may be a partial pathname. ProDOS 16 supports 8 prefixes, three of which have fixed definitions:

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

Run Time Library Files

Run Time Library Files ($B4) contain dynamic Load Segments which the **System Loader** can load as they are referenced through the **Jump Table**. Usually, these files contain general routines that can be shared by more than one application.

Run Time Library Files are also scanned by the **Linker** during the link process. When the **Linker** finds a segment referenced in the Run Time Library File, it does not extract the segment out of the File as is the case with Library Files. Instead, the **Linker** generates an INTERSEG reference to the segment for the **Loader** to resolve and adds an entry to the **Jump Table Segment.** This is the same thing the **Linker** does when it encounters a reference to a Dynamic Segment in any Load File.

The information that the Linker needs to find referenced segments is contained in the last Load Segment. This segment contains a table of ENTRY ($F4) records corresponding to each Segment Name and global Entry included in the whole Run Time Library File.

Run Time Library Files at load time must reside in the subdirectory that was specified during link time. The pathname of the Run Time Library File is actually stored in the Pathname Segment in the Load File of the application program.

Run Time Library Files are created by the **Linker** from the corresponding Object Files.

Shell Load Files

Shell files are Load Files with the file type $B5. They are intended for execution by a shell that supports standard input and output, and can be launched alternatively by ProDOS (via the QUIT command) or manually loaded by the shell.

Shell files expect that the launching program has established some form of TTY input and output, accessible via the global i/o hooks. ProDOS should do this for $B5 file types by pointing to the Pascal drivers for the 80-column screen and keyboard. If they require any support other than TTY i/o and Text tools calls, they need to check the shell identifier string to see if the appropriate environment is in place. If the shell environment does not support the application, it should exit with an error message displayed on the standard error output.

The identifier string (and the input line) is pointed to by the X (high word) and Y (low word) registers on entry. ProDOS, which does not support the identifier string or input line, will pass zeroes in X and Y when it launches a program. The area pointed to by these registers consists of an eight-character shell identifier, followed by a null terminated string, which should be the complete input line as the shell program is supposed to see it. It would be permissible, for example, for the calling shell to remove input and output redirection and pipeline symbols, but the command name and all parameters must be passed on. The User ID is passed in the accumulator.

If the program gets its own User ID for any reason, it is responsible for intercepting the RESET vector and interrupts to insure that control can never pass back to the shell before it gets a chance to dispose of resources associated with that User ID. The practice of asking for a second User ID is highly discouraged.

On exit from the program, the accumulator must be set to an error code. A zero indicates that no error occurred, while any other value signals an error. Some shells may specify what the various error numbers mean. The error $FFFF is reserved as a non-specific error for all shells, and should be recognized as such if error-trapping is supported.

Programs should exit via the ProDOS **QUIT** call. If the shell has manually launched the program, it is responsible for intercepting the **QUIT** call (which means checking all ProDOS calls) so that ProDOS does not process the **QUIT**. (CPW is an example of a program which does this.)

Shells should observe ProDOS conventions for register initialization and bank zero allocation for applications without Direct Page/Stack Segments. (i. e., initialize 1K bank zero region.)

References

"System Loader ERS" by Lou Infeld -- Apple Computer
"ProDOS ORCA/M User's Guide" -- The Byte Works
"Cortland Programmer's Workshop Core" -- The Byte Works
"The Tool Locator ERS", by Steve Glass -- Apple Computer