# Using Object Oriented Languages for Building Non-Applications in MPW

**by Allan Foster and David Newman**

*Over the past few years object oriented programming has been pushed as an appealing approach for Macintosh programming projects. Unfortunately, the tools provided have only allowed this to be used by Applications.*

*This paper shows a technique for using OOP languages for writing stand alone code for the Mac OS. Examples of these are INITs, XCMDs and the various DefProcs for the Mac managers.*

*This paper will show how to use Global variables in these code resources, as well as how to provide the necessary framework for both C++ and Object Pascal to be used.*

*Using MPW as the development environment, it will be shown how a runtime library combined with a post-link MPW tool are employed to build object oriented stand alone code. Full support for Object Pascal and C++ are provided, including support for C++ static constructors and destructors. It then demostrates how to build multi-segment code resources, and where they would be used.*

## Introduction

There are essentially two types of executable code written for the macintosh. The first, and most visible, is the application. Most of the available development environments provide very strong support for building and debugging applications.

The second type of executable code that is written for the macintosh, is the so called Stand Alone code. Into this realm fall the neat hacks that we all know and love! INITs, XCMDs, CDEVs and all the other executable resources that the mac os uses in support of the applications.

The THINK products have provided us with fairly strong support for building these resources for some time. However the development environment from Apple, MPW, has provided only minimal support, and placed some major restrictions on the developers of these resources.

We will show you how, with a little extra support, you can build the stand alone resources without being limited by the restrictions currently placed on developers.

In order to appreciate the problems that need to be overcome, we need to have a clear understanding of the different types of module references that are generated by the compilers and linker.

## Address References.

There are only two types of modules that the MPW linker is capable of dealing with. The two types being CODE, and DATA.

Each of these are able to reference the other. Therefore, there are four different module references that are possible.

Each of these have special needs that the linker must satisfy in order for them to function properly.

Code to Code.

This type of reference comes in two distinct flavors. The simple form of this reference is generated by a function call to another function within the same segment. In this case, the linker knows up front the offset from the one routine to the other, and a simple PC relative address is generated for the JSR.

```
eg:   JSR   *+350
```

A code to code reference that is across segments creates a problem for the linker. The final location of the segment being referenced is not known by the linker. This information is only known at runtime. and therefore can only be fully resolved at runtime, when the segment is actually loaded.

For any reference to code in another segment, the linker actually generates a reference to a special data module called the jump table. The jump table is nothing more than a special data module that resides at a positive offset from A5.

First the linker creates an entry in the jump table for the routine being referenced, which consists of a little code that resolves the reference at runtime, and then treats this reference as nothing more than a Code to Data reference, and can easily be resolved as explained in the next section.

```
eg:   JSR   128(A5)
```

The second part, is the resolution of the actual jump table entry itself. The only information that the linker has, is the segment number, and the offset into that segment being referenced. Since the linker cannot resolve this reference any further, it just gives up, and provides this information to the runtime routine to actually resolve the reference.

At runtime, the final resolution of the references is performed by the toolbox trap _LoadSeg. Since _LoadSeg has the segment ID, the offset into that segment, as well as the address of where the segment has been loaded, it can easily resolve the reference.

Code to Data.

This reference is generated whenever we have code that references a global variable. Since MPW uses Address Register A5 as the pointer to global space, all data references are simply resolved to an offset from A5. The address generated for the instruction is a simple A5 indirect with displacement.

```
eg:   PEA   -1400(A5)
```

The only time that references from Data occur is when there is initialized data containing references to other data or code. This is common practice in C programming, but not allowed by the pascal syntax.

Data to Data.

This type of reference is found when a variable in global data space, is statically initialized to contain the address of another variable.  The syntax in C is fairly clear.

```
eg:   char  Buffer[1024];
      char  *BufPtr = &Buffer;
```

Since the reference is a data reference, it is able to be resolved to a simple offset from A5, exactly as in the code to data reference above.

Data to Code.

This type of reference does not occur very frequently, but is not uncommon in C.  The following C statements will produce a Data to Code reference.

```
eg:     void    function1();
        void    function2();

        ProcPtr         fTable[] =
                        {       &function1,
                                &function2
                        };
```

Here we have an array of pointers to functions, which we have called fTable.  The most common occurence of this type of reference occurs in C++.  A C++ vTable is nothing more than a statically initialized array of function pointers, similar to the example above. The vTable is the C++ class method dispatcher, and as such EVERY class in C++ requires a vTable.

The same problem presents itself here as was found in the inter segment code to code references above.  Since the final location of the segments that contain the functions being referenced is not known by the linker,  it needs to generate jump table entries for the references.

These references then simply become Data to Data references, which are resolved as described above, and the jump table is again left for the runtime to resolve.

Let us now investigate what effect this has on object oriented languages under MPW.

**Object Oriented Languages**

There are two main object oriented languages supported under MPW.  These being C++ and Object Pascal.  There are others available, but these two are by far the most common.

Fairly good descriptions of the implementations of these languages may be found in several different books dealing with the subject.  Without going into any detail about the implementations of each of these languages, let us look at what support they require from the runtime environment.

There is, however, one feature of C++ that does present an interesting implementation problem.  C++ provides for Constructors and Destructors in its classes.

The designers of C++ realized that when a class is instantiated, there is likely to be a need for initialization, and a corresponding need for clean up when the instance is disposed of.  This initialization would have to be done simply as a result of creating an instance, not by any concious action by the programmer.

The constructor is a method in a C++ class that is automatically executed whenever an instance of the class is created.  The destructor is automatically executed whenever the instance is disposed of.

The problem arises when there are global instances of a class.  The constructor needs to be executed before the main code is run, and the destructor needs to be run when the main code is finished executing.

This presents another problem for the runtime startup code, besides having to build the A5 world, with all the statically initialized global variables, it also needs to execute the constructors of any global class instances.

**Current Restrictions**

MPW places one restriction on the devloper of stand alone executable resources.  This restriction leads to several major limitations in what may be done in stand alone executables, as well as in the tools necessary to develop them.

Stand alone executables, may not have any DATA modules or any references to DATA modules.  From the discussion above, the consequences are quite severe, immediately presenting two major limitations.

**No Globals**

In the discussion above, it was shown that global variables are in fact data modules. Since stand alone executable resources cannot have any data modules, there can be no use of global variables.

The reason for this is actually pretty simple. Since the MPW linker locates all data modules as offsets off of A5, it assumes that there is an A5 world in place at runtime to accomodate this data. Since stand alone executables as produced by the MPW linker do not have the startup code required to build a valid A5 world, the MPW linker, does not allow anything to be based off of A5.

Not being able to use global variables is a fairly substantial limitation. Many developers have been very thankful of the THINK products for removing this limitation from their development system.

Aside from the inconvienience, this restriction has a profound impact on using C++ as a development language.

As can be seen from the discussion above, C++ uses global variables for the vTables used as the method dispatch mechanism. These vTables are statically initialized global variables. Since stand alone executables cannot contain or reference any data modules, they cannot reference the C++ vTables, and therefore cannot use C++.

**No Jump Table**

All address references have to be either PC-Relative, or Absolute. Again, refering back to the discussion about module references, not being able to have any data references rules out the posibility of having a jump table, since the jump table is in fact data module.

It is immediately evident why there has been a long standing belief in Mac development that stand alone executable resources cannot be larger than 32K. Since the 68000 can only do PC-Relative addressing using a word sized offset, the largest branch that may be taken is 32K in either direction. Using a jump table, and multiple segments would relieve this restriction, but then that would require Data references.

A Mac application is able to use multiple segments, and in this way it is able to circumvent the 32K size restriction. But this requires the use of a valid A5 world.

Not being able to use a jump table makes it is very difficult to write a multi segment stand alone executable. In a normal application, the linker takes care of all of the bookkeeping and ensures that the jump table is maintained correctly. In order to do this in stand alone executable code, the programmer has to assume the responsibility and bookkeeping chores. This is a tedious, and very error prone task., and is not something that the programmer should have to be concerned with.

Obviously, then, if any serious work is going to be done with object oriented languages in stand alone executables, these restrictions are clearly unacceptable, and need to be removed.

**Removing the restrictions**

Now that we have investigated the restrictions placed upon us, as well as the implied limitations , we can go ahead and remove them.

**Providing Global Variables**

From the discussion above, it can be seen that the MPW compilers and linker locates all data modules variables as offsets from A5. This means that global variables are located at negative offsets from register A5.

It is immediately obvious where the 32K limit on global variables comes from. Since the linker generates register reslative addressing modes to access global variables, this limits the range to 16 bits, or 32K on either side of the location pointed to by the register.

There is no problem convincing the compilers and linker to generate code that references the data modules relative to A5, as long as there is provision made to build an A5 world for the code when it executes. This is done by providing some runtime startup code. This startup routine has to allocate space for the globals, and set A5 to point into it.

This is all explained in tech note 256, which along with providing the necessary information on what to do, also provides sample code that does it.

The MPW linker emits two functions for dealing with building the A5 world. Pascal conventions demand that all function and procedure names be converted to upper case, so these routines are not callable from Pascal, since their names are in mixed case. Here are the C prototypes.

```
short       A5Size();
void        A5Init(long A5Ptr);
```

These two functions are in fact very simple. The first function, A5Size, simply returns the size of the global space. Since this is a C calling convention, the result is returned in register D0. The second function, A5Init, performs the static initialization of any pre-initialized global variables. By making these functions accessable to us, the linker has in fact provided everything needed to build and initialize an A5 world for our stand alone code.

By including the techniques from the tech note into the SARuntime, the limitation of not being able to use global variables disappears.

**Providing a Jump Table**

Since the previous section has already provided the stand alone code with a valid A5 world, we should be able to use it for a jump table as well.

Getting the MPW linker to actually generate a jump table for stand alone code is a lot more difficult. The MPW linker will only generate a jump table if it is building an application. Therefore we cannot link the code as a stand alone resource, and have a jump table generated.

Before we can continue, we need alittle background on how the jump table works for standard mac applications.

Not every routine in an application requires a jump table entry. The are only two different situations where a jump table entry is required.

Any routine that is called from a segment other than the one it resides in requires an entry.

Any routine whose address is taken for an indirect call requires a jump table entry. These could be indirect calls from the code itself, of more commonly, procedures passed as callback functions to toolbox routines.

A routine that is only called from within the segment is resides in, does NOT require an entry, since the call may be made with a PC-Relative offset, and this can be calculated by the linker when the application is linked.

Entries in the jump table may be in one of two states. The segment may be either Loaded, or Unloaded. See figure 1 for the differences in the tables.

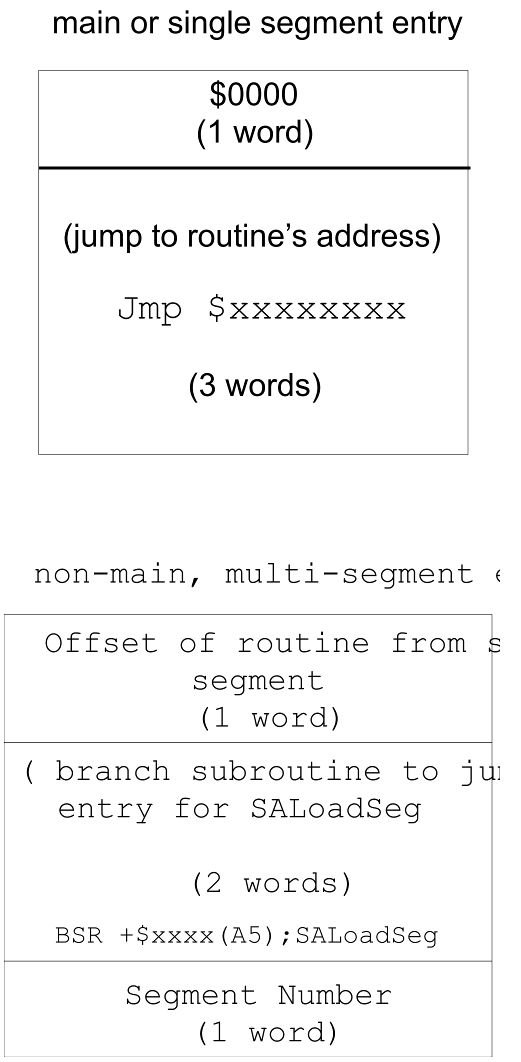

Figure 1

To enable us to use the jump table created for applications, we need to slightly modify the way it works.

In an application, a jump table entry that is in the loaded state has a very simple format. it is simply a jump instruction to the absolute address of the routine in the code segment. Since the segment cannot move, once it is loaded, it will not change. This is true for our Stand alone executables as well, so this format does not present a problem. We will see that in the stand alone executables, this case only occurs for the main segment.

The jump table entry for an unloaded segment is a little more difficult to deal with. A normal jump table for an application uses the _LoadSeg trap to load the segments as needed.  The stand alone code cannot use _LoadSeg, since it is designed specifically for applications.

The stand alone code needs to provide an alternative to the _LoadSeg trap, without altering the structure of the jump table.  We do this in the runtime library with the SALoadSeg routine.  In order to keep the same structure in the jump table, the entries are built as shown in figure 2.

Instead on putting the trap word for LoadSeg into the jump table entry, we replace it with a BSR to the Jump table entry for our own SALoadSeg routine. The reason for a BSR instead of a branch will become clear shortly.

By providing a replacement loadseg,  we can provide multiple segments to stand alone executable resources, without the developer having to do any extra work.

Obviously then, we need some way to convert the application into the stand alone executable we are trying to build.  This is by the post-link tool, MakeSA

MakeSA not only creates a jump table for our stand alone resource, but also generates the tables needed by the startup code for constructors, destructors, aas well as the modified segment loader.

We will go into the details of how MakeSA works in the next section.

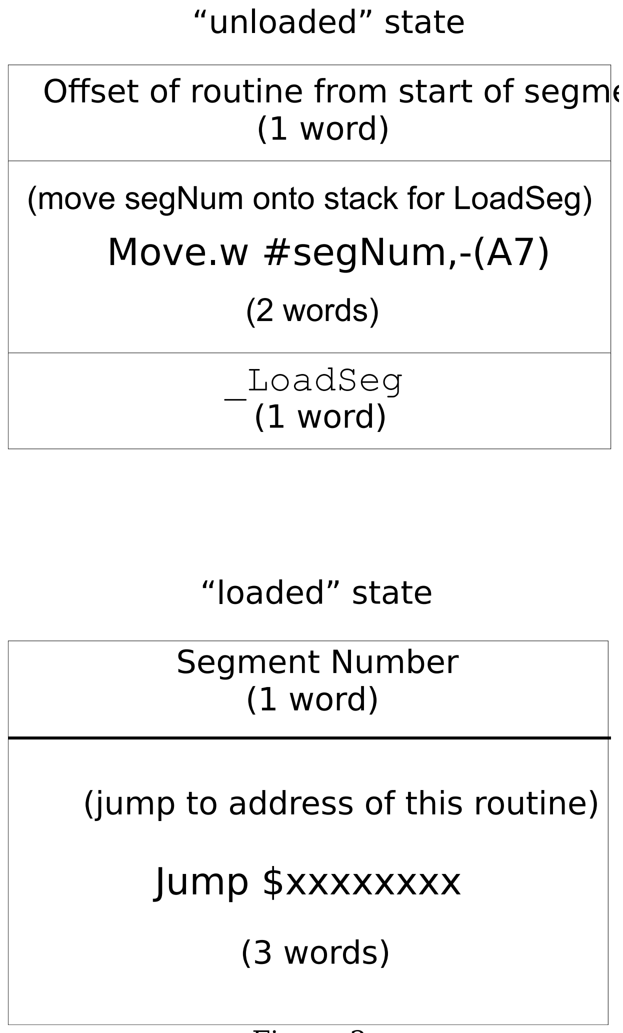

Figure 2.

The major restriction that MPW places on developers of stand alone code resources, that of not being able to use DATA modules, has been removed. There is now, only one more implementation detail that needs to be dealt with in order to use object oriented languages in the resources.

**C++ Constructors and Destructors**

C++ has a few special features that need to be addressed.  This is the handling of the static contructors and destructors.

First a quick explanation of these unique beasts!

C++ provides two methods for classes that make the initializing and disposing of the classes much easier to maintain.  This is the contructor, and its associated destructor.

In the following code example, the very act of defining the local variable theClass to be of type TObj, will cause the constructor for TObj to be invoked, and the destructor to be called when the function exits.

```
void example()
{
      TObj  theClass;

      theClass.doit();

};
```

To provide a similar functionality in C, one would need to write the following code:

```
void example()
{
      TObj  theClass;

      INITTObj(theClass );

      theClass.doit();

      DisposeObj(theClass);
};
```

The benefits of using constructors and destructors in C++ is pointed out in many of the books written about C++. .  It is in fact one of the advantages of using C++..

The problem comes in with global declarations of objects.  Since it is a global, it conceptually exists when the code starts executing, but the constructor still needs to be executed.  In an application, the runtime startup code is responsible for executing the constructors, and making sure that they are executed in the correct order.  The problem for stand alone resource, is in making sure that the constructor gets called before the main function begins executing, as well as making sure that the constructors are called in the right order!

In a normal application, the MPW linker generated a special code segment that is called "%_Static_Constructors_Destructor_Pointers".  This is nothing more than a table of offsets into the jump table of the constructor functions that need to be executed, along with the associated destructor functions that will be executed when the application quits.

MakeSA works with this segment, and builds a table that is used by the runtime startup.  The constructor functions are then executed before passing control to the main function in the stand alone resource. The destructor functions are not executed until the stand alone executable terminates.

**Making it transparent**

Our final goal was to make the library as transparent as possible.  We did not want to have to force developers to call a function to setup the A5 world, or perform any other housekeeping.  We achieved this by providing our own runtime with its own entry point, that does everything necessary before executing the actual stand alone code.

All parameters are left undisturbed on the stack, so the main function is declared exactly as required by the calling code, and is invoked with the correct parameters on the stack.

Therefore, the developer need not make special calls to setup the runtime environment, nor to tear it down.  We did provide special routines for the developer to call should they want to exit the stand alone temporarily.  Typically, for a callback into the parent application.  These routines are similiar to SetupA5 and RestoreA5 in the MPW libraries.  These will allow the standalone to call outside of the standalone's A5 environment, and restore that environment upon returning.

We also have provided the full source code for both the tool and the library, so any special handling can be dealt with by the developer if necessary.

**Implementation details.**

The runtime routines provided are responsible for the setup and tear down of the execution environment.  Appendix A provides a step by step explanation of how the runtime provides this support, but there are a few areas that should be explained.

Several technical problems needed to be solved before this runtime could be implemented. They were:

1 Building a jump table with the entries having the same size as the entries in a standard application jump table.
2 Managing segments without dealing with self modifying code.
3. Managing the constructors and destructors.

4. Ensuring that the runtime was called at the end of execution, without disturbing the parameters on the stack that need to be passed on to main.

**Finding a new jump table format**

Standard jump table entries are 8 bytes in size, and since the linker builds the jumps into the jump table with this assumption, we could not change the size of the entries.

For a single segment resource, the jump table was not a problem, since we could just build it with the same format as a loaded segment in a standard jump table. The startup code knows where the executable resource is loaded into memory, and that that address is not going to change, so the runtime can fill the jump table entries with absolute jumps to the correct routines.

A problem presents itself when trying to deal with multiple segments. Since we cannot use the _LoadSeg trap, since it is already in use by the running application, we had to come up with another method of loading segments.

In order to load a segment and jump to the required routine, we need two peices of information. We need to know the resource ID of the segment we are going to load, and we need to know the offset into that segment that the routine we need resides. Both of these values are word sized, and as such consume 4 bytes of the 8 alloted to a jump table entry. This leaves 4 bytes into which executable code needs to be placed in order to load the segment.

The solution to this problem is to do a BSR to the jump table entry for the SALoadSeg routine in the runtime library, and since that routine is in the main segment, its jump table entry will simply contain the absolute jump to the right address.

The reason that a Branch Subroutine was used here instead of a Branch, is actually fairly simple. The SALoadSeg function needs a pointer to the jump table entry so that the segment number and offset can be fetched. The return address on the stack, left by the BSR, points right back into the correct jump table entry This value is popped off the stack, and used to get the required segment ID, and offset.

Any inter segment jump always executes the SALoadSeg function. The disadvantage of this is that inter segment jumps have a little more overhead in their execution. There are a few advantages of this technique, not the least of which, is the absence of self modifying code.

**Loading and Unloading of segments.**

Since our SALoadSeg routine is going to be executed on every inter segment jump, it needs to be fairly quick.

In order to manage segments, we have an array of segment handles, that are intially set to nil. When we are called upon to load a segment, we get the segment ID, and use it as an index into this table. If the resulting entry is not nil, then the segment is already loaded, and we have its handle, so we can just jump into the correct routine.

If the handle is nil, then we call getResource to load the segment, and store its handle into the array, so future calls will find it already loaded!

SAUnloadseg simply does a releaseResource on the handle, and sets its entry to nil, so the same precautions need to be made with this routine as with the _UnloadSeg trap. Do NOT unload a segment that is in the calling chain.

This design of the jump table can lead to at least two interesting areas of exploration. The first would be the to have the unloading of segments done automatically when they are no longer needed. This becomes possible, since every inter-segment jump has to go through the SALoadseg function, and as such, reference counting could be done on loaded segments.

The second interesting area, would be to allow segments to be unloaded even if they are in the calling chain. This is possible, again because all inter-segment jumps have to go through the SALoadSeg function. Keeping track of the segment and offset the the call is coming from, and restoring the return address when it returns, if the segment has moved.

**Constructors and Destructors.**

We build a table for the constructors and destructors at the head of our resource. These tables consist of nothing more than offsets into the jump table of routines that need to be called.

Before calling the main function of the stand alone code, we walk this list and execute the required routines.

The destructors are dealt with in the same manner. MakeSA is responsible for making sure that the destructors are called in the reverse order of the constructors , as required by C++.

**Regaining control after execution**

We needed the runtime to regain control after execution of the stand alone code, but since we do not know the number of type of the parameters passed, we cannot simply JSR to the main function. This is because the JSR would leave an extra address on the stack, and therefore mess up the parameters to the main routine.

Instead, the runtime startup saves off the return address into a global variable, and pushes the address of our Runtime cleanup routine onto the stack in its place. Since there is now a return address on the stack, the runtime can simply jump to the main function. When the main function returns, it will return to our cleanup routine, which can then go ahead and call the destructors, dispose of the A5 world, and return to the original caller.

If the actual return address is needed, it is readily available in the global variable that it was saved into.

**How It Works**

Now that we have discussed how it can all be done, and we know the problems that have been solved, we can discuss the implementation that we have provided.

This is done in two parts. The first will explain the runtime support routines, and the second part will discuss the MPW tool, MakeSA, that performs the post link phase of the build.

Building the Table.

MakeSA builds the table in Figure 3 and puts it at the begining of the executable resource.

Since all executable resources are called with the entrypoint at offset 0, the is a BSR at this point to get to our runtime support. As before is a BSR and not a BRA, since we are going to need a pointer to the begining of the resource, and pulling it from the stack was as convienient method as any.

Following the BSR are the tables that are generated by MakeSA. Each table is used by the runtime support to build the execution environment.

The first section of the runtime code walks the tables, and stores off pointers to the various tables in local vars.

The runtime then executes a JSR to A5Size to get the size of the global space the executable is expecting. Remember that the routine returns its result in D0.

The size of the handle we need to allocate is calculated by the following formula:

```
(# JT_Entries * 8) + 32 + GlobSize
```

This value is passed to _NewHandle to actually allocate the space, along with all the associated housekeeping.

A5 needs to be set to the transition point between the globals and the jump table, so the global size is added to the pointer we just allocated, and the result is move into A5. The first peice of building our own A5 world is now completed.

The global space now has to be initialized, which is done by the linker generated function A5Init.

Since we now have a global world, the runtime has decalred a few globals of its own. The original value of A5 needs to be stored, so that it can be restored when the stand alone code has completed executing. This original value of A5 is saved off into SAOldA5(A5)

Three more global variables need to be used by the runtime code, and they are initialized here.

SAGlobHDL contains the handle that was allocated for the A5 world. This needs to be saved, since the runtime needs to dispose of it later, when the stand alone terminates.

SASegType is a variable of type OSType, that contains the four character ID that is the type of any multi segment resources.

SADtorPtr contains a pointer to the table of Destructors that need to be called when the stand alone code has finished executing.

SASegPtr contains a pointer to the array of segment handles, so that multiple segments may be loaded automatically.

The next step walks the table of jump table entries in the header, and builds the appropriate entries in the newly constructed A5 world.

This is simply a loop that takes the segment ID, and the offset for each routine, and builds the correct entry for it.

After this, we loop through the constructor table, and execute all of the constructors.

Finally, we save the return address into SARetAddr, and push the address of StopRunTime, and then jump to the main function of the stand alone code.

The rest of the routines in the runtime are fairly self explanatory.

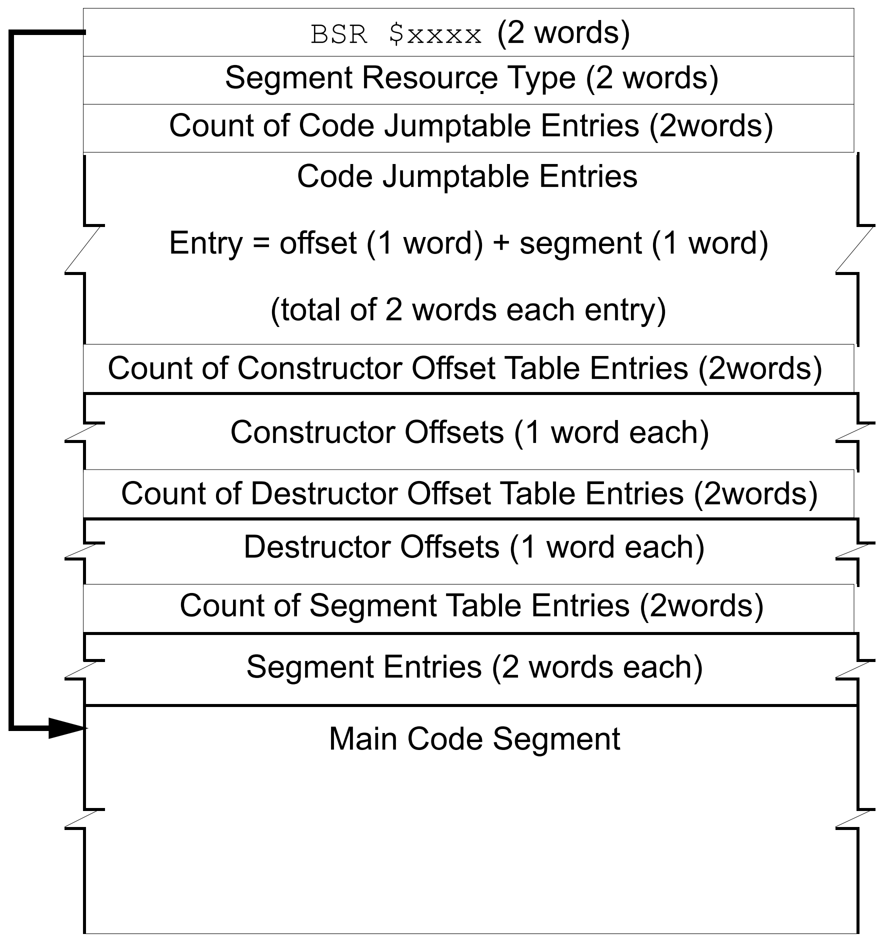
BSR $xxxx (2 words)
Segment Resource Type (2 words)
Count of Code Jumptable Entries (2words)
Code Jumptable Entries
Entry = offset (1 word) + segment (1 word)
(total of 2 words each entry)
Count of Constructor Offset Table Entries (2words)
Constructor Offsets (1 word each)
Count of Destructor Offset Table Entries (2words)
Destructor Offsets (1 word each)
Count of Segment Table Entries (2words)
Segment Entries (2 words each)
Main Code Segment

Figure 3