Macintosh C/C++ ABI

Standard Specification

Revision 1.3
Dec. 5, 1996

Authors:
Fred Forsman
Doug Landauer

Fred Forsman
Apple Computer, Inc.
(408) 974-2520
forsman@apple.com

Table Of Contents

Revision History

Revision	Date	Comments
1.0	11/16/95	Initial release of document.
1.1	5/2/96	"Natural" alignment mode; static constructors and destructors; passing objects by value; order of "this" and "hidden" parameters.
1.2	8/1/96	Long long data type; introduced 3 levels of ABI compliance; thunks; type of size_t; layout of class objects and vtables; constructors and destructors; RTTI; exception handling; trigger functions for vtable generation; XCOFF extensions for imports and exports.
1.3	12/5/96	Rewrite of alignment section; proposal for new C++ virtual function dispatch mechanism.

Acknowledgments

We would like to thank the following people who have provided valuable input to this document: Russ Daniels, Erik Eidt, Scott Fraser, Sassan Hazeghi, Andreas Hommel (Metrowerks), Doug Landauer, Alan Lillich, Colin McMaster, Ira Ruben, and Mark Williams.

The mention of anyone's name here does not imply their approval of this document, in fact, much of the best input has come from our critics. As usual, the errors in content and organization are our own.

Cover Letter and Notes on this Revision of the Specification

This document is a draft proposal and your comments are welcome. Please let us know about any ABI issues that we have missed. Please note the various "Open Issues" sections and feel free to comment on any of these or any other ABI issues.

If you are planning on writing up your comments please refer to sections by name rather than by number since the document is still changing and the names are more likely to remain meaningful.

While this version of this document is relatively complete in that it describes all of what we know to be relevant to the C++ ABI, it is still just a draft proposal and many of the sections should be considered to be "straw man" proposals which will serve as a starting point for discussion. Given that the point of a Macintosh C++ ABI is to work toward having C++ compilers for the Macintosh conform to a common set of conventions, this document makes reference to strategies used by various compilers (particularly Apple's MrCpp and Metrowerks' CodeWarrior compiler). In the various "Open Issues" sections in this document we have tried to enumerate ABI-related issues and problems that have been uncovered but not yet resolved. Please feel free to provide pros and cons on these and any other issues in the ABI. We have not had the time to investigate all of the issues that have been raised, so a more detailed analysis of these or any new issues would be greatly appreciated.

The sections on Virtual Base Classes, Virtual Base Class Tables, and Pointers to Members are provisional and have a reasonably high likelihood of changing. The run-time type information (RTTI) mechanism is being investigated and compared with other RTTI implementations. The exception handling mechanism will be redesigned to address the issues which are listed in the "Open Issues - Exceptions" section. The new exception mechanism will be documented in a future revision of this document.

This revision of the ABI also includes a proposal for a new C++ virtual function dispatch convention. Your comments are welcome. Preliminary discussions of this mechanism have already indicated areas for optimization of the current mechanism if the new one is not adopted.

This revision of the ABI document includes a rewrite of the section on alignment. The alignment rules have not changed, however, we hope that the specification of these rules has become much clearer.

The current version of this document is known to have an application-centric perspective. Subsequent revisions will attempt to provide more information on how DLLs affect ABI issues.

1. Introduction: The Macintosh C/C++ ABI

This section outlines the purpose and scope of this document.

1.1 What is an ABI?

An ABI is an “application binary interface” and describes conventions at the binary level which apply to applications targeted to the same system. An ABI establishes conventions for such things as register usage, parameter passing, and layout of data. Typically, these conventions are effected in development tools, particularly compilers. Assembly-level programmers must be aware explicitly of these conventions, especially if they wish to interface their assembly code with code produced by a compiler.

The primary goals of an ABI are:

1. to establish machine-level runtime conventions for a processor family
2. to ensure object file compatibility between compilers

While it is possible for programs to depart from the conventions of an ABI, particularly within isolated sections of a program (such as sections of hand-crafted assembly code), conformance to the ABI is often required to make use of system-level code and code produced by other compilers. To the extent that a program is monolithic and is built with the same set of tools conformance to the ABI is only an issue when the program interfaces with the system. To the extent that a program is made up of (or accesses) components which may have been built with other tools conformance to the ABI is more critical.

For some, the issue of binary compatibility resolves to whether the output of two compilers can be made to be link compatible. But for others, binary compatibility can be seen as the issue of whether two compilers will produce objects with the same link characteristics given the same source. This latter, more constraining definition of binary compatibility is the one we will address in this document since it ensures a greater degree of source portability between conforming compilers.

1.2 The Scope of this Document

The foundation level of the Macintosh ABI is documented in the book *Macintosh Runtime Architectures* (to be available in fall '96) which covers various basic conventions such as register usage and parameter passing conventions. This document covers what we are calling the "Macintosh C/C++ ABI" which includes some special cases of low-level conventions needed for C and C++, conventions for C such as alignment rules (which also apply to C++), and conventions for C++ specific constructs which get exposed at the ABI level.

The C++ language poses special ABI problems. While the C language relies on a few well-established ABI conventions, the object-oriented features of C++ require a new runtime architecture and conventions, such as v-tables and name-mangling. While some of these conventions have a common precedent in their implementation in AT&T’s CFront, the evolution of the language and the proliferation of compilers has resulted in the availability of many compilers whose binaries are not compatible. The purpose of this document is to address that problem by establishing a set of ABI conventions for C++ on the Macintosh.

This document limits itself to issues directly related to the support of the C and C++ languages with the expectation that this restricted focus will facilitate adoption of a standard set of runtime

conventions by Macintosh C++ compiler providers. Other issues such as standards for SOM and Direct-to-SOM are left to other documents. This document currently limits itself to describing these conventions for the PowerPC Macintosh since many of the conventions for the 68K Macintosh have been in place for many years and since significant changes in compilers targeting the 68K are not anticipated.

Several source compatibility issues are mentioned in this document due to their close connection to ABI issues (for example, the use of types other than "int" for bit fields). While not strictly ABI issues, they can affect source portability between compilers.

The ABI attempts to address issues spanning various levels of interoperability--procedural DLL interoperability, C++ DLL interoperability, compiler and linker interoperability, and source interoperability. We have chosen the tack of organizing the ABI by functional areas since any one given functional area may include issues at several levels of interoperability. Your suggestions for other organizations are welcome.

One important issue which is not addressed in this document is the process of transforming a compiler to conform to the ABI and what to do at the inevitable stages where the compiler only conforms partially to the ABI. The issue is made more complicated by the fact that a change in ABI will often require that both an old and new ABI be supported during the transition. A succession of steps towards ABI conformance can present a variety of difficult packaging issues since the different ABIs will often require different libraries.

1.3 The Evolution of this Specification

This specification will continue to evolve for some time for the following reasons:

1. This ABI specification is a working draft.

   The Macintosh C/C++ ABI is a work in progress--progress toward identifying all of the potential ABI issues raised by the C++ language, documenting conventions to address these issues, and then coming to an agreement with Macintosh C/C++ compiler providers about the ABI conventions.

   Given the open nature of this process, we encourage discussion of the conventions documented here. If there are potential problem areas that have yet to be identified please let us know.

2. The C++ language is still evolving.

   While the C++ language is nearing standardization, some areas, such as templates, have continued to change. Some of these changes affect ABI issues (such as name mangling of templates).

3. The ABI specification identifies issues which have not been resolved.

   Many sections of the ABI specification contain a sub-section of open issues. Many of these are questions and suggestions that have not yet been investigated. Please feel free to discuss these issues with us and to identify any other potential problem areas in the specification.

4. The ABI should not be immutable.

If there is something wrong or sub-optimal about the current ABI specification please let us know. Even once the standard is complete and has been widely adopted, potential improvements should be discussed and weighed against the cost of change.

1.4 When it is Permitted to Deviate from the ABI

The ABI may be violated in generating code for and calls to functions which are known to be local to a compilation unit or internal to a DLL. When a function and all calls to it are local to the compilation unit the compiler may choose to relax the rules of the ABI in order to generate more optimal code. Similarly, when a function is known to be internal to a DLL, the compiler may deviate from the ABI as it chooses as long as the same compiler is used to build all of the pieces of the DLL. The compiler may identify these cases by observing which functions are static and which are specified as internal using the CFM pragmas.

Going a step further, the ABI may be freely violated between DLLs that are constrained to be used together.

2. PowerPC Runtime ABI

This section provides an overview of the conventions of the Macintosh runtime ABI; additionally it describes some special case calling conventions needed to support larger data types as parameters and return values.

2.1 PowerPC ABI Overview

The processor-level Macintosh PowerPC ABI is based on the PowerOpen ABI. The details of the Macintosh PowerPC ABI are given in *Macintosh Runtime Architectures* ([2]).

2.2 Special Calling Conventions

This section describes calling conventions which are not covered explicitly in *Macintosh Runtime Architectures* ([2]). See *Macintosh Runtime Architectures* for details on the Macintosh PowerPC parameter passing conventions.

2.2.1 Passing and Returning Long Longs

Long longs are a 64-bit integral data type. They are represented in memory by two adjacent longs, the first containing the high-order 32 bits and the second containing the low-order 32 bits.

Long long parameters occupy two words in the parameter area and will be passing in registers if they fall in the first eight words of the parameter area.

Long long function return values are returned in the R3/R4 register pair.

2.2.2 Passing and Returning Long Doubles

Long doubles are a 128-bit floating point data type. They are represented in memory by two adjacent doubles, the first containing the high-order 64 bits and the second containing the low-order 64 bits.

Long double parameters occupy four words in the parameter area will be passing in floating point registers if they fall in the first eight words of the parameter area.

Long double function return values are returned in the FPR1/FPR2 floating point register pair.

2.3 Open Issues - Calling Conventions

1. When chars or shorts are passed in a register are the upper bytes of the register undefined? Or do we expect the caller or the callee to sign extend the value if appropriate? *Macintosh Runtime Architectures* indicates that the high order bits are undefined.
2. Consider returning small structures by value. Currently structures are returned by reference via a hidden temporary passed to the function, regardless of the size of the structure.

3. Macintosh PowerPC Compiler ABI

The Macintosh compiler ABI covers conventions which apply to compilers targeting the Macintosh architecture. Conventions such as object module formats apply to compilers independent of their target language. Other conventions are language dependent, such as alignment and bit field rules which apply to C and C++ compilers. The C++ language requires quite a few conventions to support its set of language features. A common set of conventions must be followed in order for compilers to produce code that will be link compatible and will interoperate with code produced by other compilers. These conventions are discussed in the following sections.

3.1 Object Module Formats

Object module formats specify the format of the output of compilers which, in turn, is input for linkers and, sometimes, debuggers. An object file includes code, data descriptions and initializations, fixup information for relocating code, and information to support symbolic debugging.

Integrated development environments (IDEs) may have their own internal conventions for object formats, but should have provisions for importing and exporting object files for use with other systems. This section identifies the standard object formats for the Macintosh.

3.1.1 PowerPC Macintosh Object Module Format - XCOFF

The object module format for PowerPC Macintosh architecture is XCOFF. The XCOFF format is documented in "AIX XCOFF Object and Load Module Format for IBM RISC System/6000" [5].

The XCOFF format has been extended for use on the PowerPC Macintosh as indicated in the following section.

3.1.1.1 XCOFF Extension to Identify Imports and Exports

Motivation

The goal of this extension to XCOFF is to allow translators such as compilers and assemblers to provide linkers/binders with more information about the imports and exports of DLLs in order to simplify the build process for such libraries and to enable more efficient code generation. It is also a goal to minimize the impact of such an extension to the XCOFF format so that tools processing XCOFF files will require modification only if they wish to process the new information.

The document "C Compiler Pragmas for Macintosh “CFM” Runtime" [4] describes the pragmas used to identify exports and imports in C and C++ programs. In this section we describe how this information can be passed from the compiler to the linker via the generated XCOFF file.

XCOFF Change

Import and export information will appear in two new XCOFF sections, named “.import” and “.export”. The type (s_flags) of these new sections will be STYP_INFO identifying them as comment sections. As such, they should be ignored by any well-written XCOFF tool which does not know or care about them.

The format of the contents of these sections will be names in STAB string format. The section begins with a four byte length, which is the length of the comment section, not including the 4 length bytes. The contents of the section will then consist of a sequence of strings with two-byte lengths and NULL terminators, where the length is the length of the string including the NULL terminator but not including the two length bytes.

The section headers for these new sections should specify the following fields: s_name (“.import” or “.export”), s_size (size in bytes of the section), s_scnptr (offset from the beginning of the XCOFF file to the first byte of the section data), and s_flags (STYP_INFO). The remaining section header fields should be zero.

When processing the new sections when creating a statically linked library a linker should concatenate the contents of any “.export” sections to create the “.export” section for the library, and similarly for “.import” sections.

3.2 Compiler Support Libraries

This section covers issues pertaining to language support libraries.

3.2.1 Runtime Routines with Special Calling Conventions

The ABI would benefit from defining a common set of compiler runtime-support routines with special calling conventions which the compiler can call with less overhead than a normal routine. For example, the compiler could know which volatile registers remain unchanged when calling such routines.

<Candidates for such routines need to be identified.>

3.2.2 Names for Compiler Runtime-support Routines

Even if a compiler conforms to the ABI, there may still be the potential for conflicts when linking with the output of other compilers if the compilers make use of runtime-support routines whose names may collide with those provided by other vendors which may not follow the same calling conventions.

Mangling the names of runtime-support routines provides only a partial solution, limiting collisions to routines with the same names and signatures which is still not a guarantee that they implement the same function.

To ensure that there are no collisions, runtime-support routines should have names which contain some unique (vendor-specific) tag tying the routine to the corresponding compiler. In addition, the ABI could specify the names and functions of standard runtime-support routines to be made available to all compilers on the platform.

3.3 C Compiler ABI

This section defines the conventions which apply to compilers supporting the C language.

3.3.1 Alignment and Bit Field Layout

This section defines the rules for alignment of data objects and the layout of bit fields.[1] The compiler's alignment mode can typically be controlled by command line options, environment settings or pragmas. The pragmas controlling alignment modes are described in Appendix 1.

Alignment is involved in five areas of memory allocation: local and globals (free standing variables), parameters, heap variables, and members of structures.

This section defines the alignment rules for members of structures and array elements, as well as for allocation of local and global variables.

The alignment for parameters is set by the calling conventions of this ABI, not by these rules.

Since alignment of local variables is relative to the stack pointer, and there is no hardware or OS enforcement 16 byte alignment of the stack pointer, physical alignment of locals is not assured. Today's compiled code generally insure 8 byte alignment of the stack pointer.

---

1 These rules were derived early in Apple's PowerPC program from alignment rules supported by IBM's xlc compiler and from alignment rules supported by Apple's Macintosh 68K C compiler.

This ABI does not specify the actions of memory allocators for heap variables. It is expected that a general purpose allocator return memory aligned relative to the size of allocation.

Many environments (ABIs) on other platforms support only one alignment mode, a natural alignment mode, sometimes made from a compromise between performance and space requirements. By contrast, this ABI defines and condones multiple alignment strategies. The presence of multiple alignment modes causes situations that single mode environments never have to deal with. This specification attempts to clarify the interactions of the different alignment modes by separating aspects of alignment that most other ABIs don't attempt to and don't need to. For example, allocating a "short" variable is addressed separately from embedding a "short" in a struct.

Each data type has two alignments: a **natural alignment**, which is fixed, and an **embedding alignment**, whose interpretation can change. Embedding alignment is controlled by the **alignment mode**.

The alignments of an enumeration data type and the int data type are the same as the alignments of the basic type (char, short, long) of the same size.

Alignment of elements in aggregates can result in **pad bytes**. Pad bytes in static objects should be initialized with zero.

3.3.1.1 Natural Alignment

The **natural alignment** of the type of is used whenever an instance of that type, a local or global, is allocated to memory or assigned a memory address.

A natural alignment for each basic type is set by this ABI and is defined with processor performance in mind.

Data Type	Natural Alignment
char	1
short	2
long	4
pointer	4
float	4
double	8
long long	8
long double	16

The natural alignment for an aggregate type is the maximum of the natural alignments of its members.

NOTE: It is recommended that compilers use the following formula to choose the alignment for local and global aggregate variables instead of using the natural alignment :

size of 1 byte -> alignment of 1 byte
size of 2-3 bytes -> alignment of 2 bytes
size of 4-7 bytes -> alignment of 4 bytes
size of 8-15 bytes -> alignment of 8 bytes
size of 16 byte or more -> larger of 8 and the embedding alignment of the variable's type

This will result in more alignment than the mandatory minimum natural alignment for variables of certain types, but there are good reasons why this is recommended. First, consider a variable of type struct or array consisting of 256 characters. Though none of the embedding members require alignment greater than 1 byte, an operation such as copying the entire array will almost certainly benefit greatly from 8 byte alignment of the overall variable. Cache performance will probably improve with the additional alignment. The second reason is more practical, that compilers need not maintain an internal value for the natural alignment for each aggregate type (computed from its members), but rather may simply compute it from the total size whenever it is needed.

3.3.1.2 Embedding Alignment and Alignment Mode

The **embedding alignment** of a type is combined with the **current alignment mode** and used when computing the offset for an instance of that type (a member) within a struct.

The embedding alignment of a type is determined when the type is declared. For basic types the embedding alignments under various modes is given in the table below.

The alignment of an array element or structure member is the lesser of the embedding alignment of the type of the element or field and the "aggregate" alignment below, determined by the current alignment mode.

The embedding alignment of an aggregate type is the largest of the resulting alignments of its constituents**. The total size of a struct or union is rounded up to a multiple of its embedding alignment**.

This ABI defines 4 alignment modes: “power”, “mac68k”, “packed” and “natural”.

Data Type	power	mac68k**	packed	natural
char	1	1	1	1
short	2	2	1	2
long	4	2	1	4
pointer	4	2	1	4
float	4	2	1	4
double	4 (8) *	2	1	8
long long	4	2	1	8
long double	4	2	1	16
aggregate	4 (8) *	2	1	16

***** When in power alignment mode, a special exception is made when the first embedding element of a struct or any element of a union has the data type of double -- in this case, the embedding alignment for all (directly included/top level) double members in the aggregate is 8. By the rules of computing the embedding alignment for the struct being declared, this will also cause the embedding alignment of the entire struct or union to be at least 8 as well. The aggregate table entry for power is 8 for such structs.

****** In mac68k alignment mode, all struct and union types have a size which is rounded up to an even byte count. The embedded alignment for all struct and union types is 2.

Example

The alignment mode is set to "power", and a struct, X, containing two longs is declared. It has a size of 8 and an embedding alignment of 4. If the alignment mode is then switched to "packed" and a struct Y is declared as one char followed by an X, the member of type X begins at an offset of 1 and the embedding alignment of Y is 1. If the alignment mode is then set to "natural", and a struct Z is declared as a short followed by a char followed by a Y, the member of type Y begins at an offset of 3 and the embedding alignment of Z is 2.

3.3.1.3 Bit Fields

There are two methods for laying out bit fields--"mac68k" and "power"--depending on the alignment mode. The differences are mainly historical, however the mac68k method can result in fewer bytes allocated for a series of bit field definitions. The "mac68k" bit field rules are in effect in the "mac68k" and "packed" alignment modes, while the "power" bit field rules are in effect in the "power" and "natural" alignment modes.

The **mac68k bit field rules** are as follows:

1. A bit field may be declared with any integral type, including an enumeration type, or “char”, “short”, or “int”, either “plain”, “signed”, or “unsigned”, just as in C++. This type has no effect on how space for the field is allocated. For example, a bit field may be declared with type “char”, yet the given width of the field may be more than the number of bits in a byte. The type is ignored, and the specified number of bits is allocated for the field.
2. The type determines whether the field is signed or unsigned. A bit field declared as plain “int” shall be a signed field.
3. A bit field may be named or unnamed. No reference can be made to an unnamed bit field. Unnamed bit fields of static objects are initialized to zero.
4. A single bit field may have a width of no more than 32-bits. A named bit field may have a width of no less that 1-bit. An unnamed bit field may have a width of 0 bits.
5. A single bit field may not span a 4-byte boundary. Bit padding will be added to cause a field that would otherwise span a 4-byte boundary to start at the next 4-byte boundary.
6. A zero-width bit field takes no space and results in no padding.
7. One or more sequential bit fields are packed into a single unit. This will be referred to here as a **bit field group**.
8. A bit field group will be allocated starting at the next byte boundary, or the current byte boundary if the previous element ended on a byte boundary, except as in rules #5 and #6. A bit field group is allocated in the minimum number of bytes needed to contain all the fields of the group, either 1, 2, 3, or 4 bytes, regardless of the declared type(s) of the bit field(s). Another way to say this is that bit fields are packed into the current 4-byte unit, in the minimum amount of space, immediately following (if possible) whatever element started at the previous 4-byte boundary.
9. A bit field group ends at the next 4-byte boundary, because of rule #5. Thus a bit field group is no more than 4-bytes wide. A single bit field that would otherwise span a 4-byte boundary will start the next bit field group at the next 4-byte boundary.
10. The fields of a bit field group will be packed into the group from high-order to low-order.
11. The preferred alignment of a bit field group is 1-byte.
12. It is NOT legal to perform the “sizeof” operator on a bit field.
13. It is not legal to take the address of a bit field.

The **power bit field rules** are the same as those for the mac68k, except for the following rules:

1. In the strict ANSI mode, a bit field may be declared with only the types “int”, “signed int”, or “unsigned int”. In the relaxed ANSI mode, a bit field may be declared with any integral type, except an enumeration.
2. The type determines whether the field is signed or unsigned. A bit field declared as plain “int” shall be an unsigned field.
4. The given width of a bit field may be no larger than can be contained in the declared type. Thus, a “char” bit field may have a width of no more than 8-bits.
6. A zero-width bit field forces alignment of the next structure element (either bit field or other member) on the next 4-byte boundary, or the current 4-byte boundary if the previous field ended on a 4-byte boundary. Thus, a zero-width bit field may not force padding in all instances.
11. The preferred alignment of a bit field is the same as its declared type.

3.3.1.4 Open Issues - Alignment

1. Alternate schemes for how bit fields should work in the new “packed” alignment mode will be considered. The current proposal is obviously based on the “mac68k” mode treatment of bit fields.

2. There may be interest some day in 64 bit wide bit fields based on long longs.

3.4 C++ ABI

This section describes ABI conventions for C++ compilers for the Macintosh which should be followed in order to produce objects which are link compatible with other Macintosh compilers.

3.4.1 Name Mangling

This section describes how C++ compilers for the Macintosh should implement name mangling in order to be link compatible with other Macintosh compilers.

Name mangling was originally developed to distinguish overloaded functions in C++ by encoding the type signature of functions in their names. Over time name mangling has developed into a means for providing type-safe linking. The presumption of this approach to type-safe linking is that linkers are typically not “smart” or easy to change, so virtually all of the work is done in the compiler.

While the C++ ARM [1] presents “one scheme” for name mangling, which was partially institutionalized in CFront, various compilers have chosen to depart from the original scheme, occasionally in major (e.g., templates) and often in minor ways. Because the ARM’s description of name mangling is “commentary” it is not an official part of the language standard. However, a consistent mangling scheme is necessary for binary interoperability of code produced by different compilers, so a standard for C++ name mangling on the Macintosh is needed.

3.4.1.1 Name Mangling and Interoperability

Interoperability of code produced by different compilers is often undermined by the compilers' use of different mangling schemes. One may encounter this issue when using a library produced by a different compiler. Problems may occur with both statically-linked and dynamically-linked shared libraries.

There are two categories of name mangling interoperability problems: (1) functions which are compatible but which are given different mangled names by different compilers, and (2) functions which are not link compatible but which are given the same mangled name by different compilers. The first problem can be addressed by encouraging all compilers to support the same name mangling standard. The second problem can come about from compilers using different calling conventions for the same function (such as how to pass an object by value), in which case the mangling conventions should provide mechanisms to distinguish the use of different calling conventions.

The C++ ARM has the following to say on this second type of incompatibility:

> If two C++ implementations for the same system use different calling sequences or in other ways are not link compatible it would be unwise to use identical encodings of type signatures. Such implementations might agree on using encodings that differ by a single character where incompatibilities exist (only). [1, §7.2.1c]

This would indicate that compilers should be required to generate incompatible mangled names until they conform to the platform's standard calling conventions.

3.4.1.2 The Rules of Mangling

The names of all externally-known C++ entities -- methods, functions, v-tables, etc. -- should be mangled except for those that are declared to be ‘extern “C”’.[2]

```
<mangled_name>          ::= <entity_name> "__" [<class_name>] <type>
```

When a name is mangled, the function name is followed by "__" and the encoded type signature. The two underscores ("__") are the signal indicating that the name is mangled. If the function is a member of a class, the class qualification is encoded before the type signature.

```
<entity_name>           ::= <id> | <special_name>
<special_name>          ::= "__ct" | "__dt" |
                            "_vtbl" | "_rttivtbl" | "_vbtbl" |
                            "__rtti" | "__ti" | "___ti" |
                            "__op" <type> | "__" <op>
```

Entity names can be either an <id> (a normal identifier or name as permitted by the language) or a special name in the case of constructors (“__ct”), destructors (“__dt”), vtables (“_vtbl” or “_rttivtbl”), virtual base class tables (“_vbtbl”), run-time type information (“__rtti”, or “__ti”, or “___ti”) or operator functions. Constructors and destructors are identified by their <class_name> and type signature. The "__op" form of <special_name> is used for types as conversion operators. Operator functions are encoded using special operator names (<op>) from the table below.

operator	<op>
"new"	"nw"
"new[]"	"nwa"
"delete"	"dl"

[2] A compiler may choose to mangle static function names in order to deal with the overloading of such functions, but this is not an ABI issue or requirement.

"delete[]"	"dla"
"+"	"pl"
"-"	"mi"
"*"	"ml"
"/"	"dv"
"%"	"md"
"^"	"er"
"/="	"adv"
"&"	"ad"
"|"	"or"
"~"	"co"
"!"	"nt"
"="	"as"
"<"	"lt"
">"	"gt"
"+="	"apl"
"-="	"ami"
"*="	"amu"
"%="	"amd"
"^="	"aer"
"&="	"aad"
"|="	"aor"
"<<"	"ls"
">>"	"rs"
">>="	"ars"
"<<="	"als"
"=="	"eq"
"!="	"ne"
"<="	"le"
">="	"ge"
"&&"	"aa"
"||"	"oo"
"++"	"pp"
"--"	"mm"
"()"	"cl"
"[]"	"vc"
"->"	"rf"
","	"cm"
"->*"	"rm"

If the entity is a member of a class, the <class_name> is encoded immediately following the "__" and before the type signature.

```
<class_name>            ::= <qualified_name>
<qualified_name>        ::= 'Q' <count> <qual_name_list>
<count>                 ::= <terminated_int>
<terminated_int>        ::= <int> '_'
<int>                   ::= <digit> | <int> <digit>
<digit>                 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<qual_name_list>        ::= <qual_name> | <qual_name_list> <qual_name>
<qual_name>             ::= <l_name> |
                            <parameterized_type>
<l_name>                ::= <int> <id>
```


If the entity is a member of a class, the class information is encoded as a <qualified_name> which is a list of names (more than one if there are nested classes). A <qualified_name> is introduced by a 'Q' followed by the <count> of the number of names in the following <qual_name_list>.

The <count> is a <terminated_int> which is an <int> followed by an underbar ('_') to separate the integer from what follows (which may begin with a digit). An <int> is a base 10 representation of a value using decimal digits with no leading zeros.

The <qual_name_list> is a list of <qual_name>'s which may be either an <l_name> (consisting of a length and the characters making up the name) or a <parameterized_type> (discussed further below). If there is more than one name in the qualified name list, the names appear from left to right in outer-to-inner order.

The type of a named entity is represented in one of several different ways depending on whether the entity is an instance of a basic type, a class, an array, or a function.

```
<type>                  ::= <basic_type> |
                            <qualified_type> |
                            <class_name> |
                            <array_type> |
                            <function_type>
```

The syntax for basic types is as follows:

```
<basic_type>            ::= <int_type> |
                            <other_base_type>
<int_type>              ::= [<sign_mod>] <int_base_type>
<sign_mod>              ::= 'S' | 'U'
<int_base_type>         ::= 'b' | 'c' | 's' | 'i' | 'l' | 'x' | 'w'
<other_base_type>       ::= 'f' | 'd' | 'r' | 'v' | 'e'
```

A <basic_type> can be either an <int_type> which can be modified by a <sign_mod> to indicate whether it is signed ('S') or unsigned ('U'). Otherwise a <basic_type> can be an <other_base_type> which includes the floating point types, void, and ellipsis.

Even though <basic_type>s which are declared without an explicit <sign_mod> attribute are considered to be signed, they should be encoded to reflect whether the <sign_mod> attribute was present or not since this distinction is significant for function overloading. Thus something declared as an "int" should be encoded as "i", an "unsigned int" as "Ui", and a "signed int" as "Si".

The encodings for basic types are given in the following table:

type	encoding
bool	'b'
char	'c'
double	'd'
...	'e'
float	'f'
int	'i'
long	'l'
long double	'r'

long long	'x'
short	's'
void	'v'
wchar_t	'w'

Note that a conformant compiler must support all of the above types as basic types in order to produce correct mangled names for entities containing these types in their type signatures. Thus, for example, a compiler in which "bool" is a predefined type will produce a different mangled name for "void f(bool)" than would a compiler for which "bool" must be provided by a typedef.

A type can be qualified using the following syntax:

```
<qualified_type>       ::= [<cv>] [<type_qualifier_list>] <type>
<type_qualifier_list>  ::= <type_qualifier> |
                           <type_qualifier_list> <type_qualifier>
<type_qualifier>       ::= 'R' | 'P' | 'M' <type>
<cv>                   ::= 'C' | 'V' | 'CV'
```

The optional <cv> allows for the specification of whether the type is a const or volatile type. The optional <type_qualifier_list> allows for the specification of a variety of type modifiers. When more than one <type_qualifier> is present they are interpreted from right to left. The meaning of the various qualifiers is given in the following table:

type	syntax	encoding
pointer	"*"	'P'
reference	"&"	'R'
pointer to member	"::*"	'M <type>'

The syntax for encoding types which represent arrays is as follows:

```
<array_type>           ::= <dimension_list> <type>
<dimension_list>       ::= <dimension> | <dimension_list> <dimension>
<dimension>            ::= 'A' <terminated_int>
```

An array type is encoded as a <dimension_list> and a <type> representing the base type of the array. Each <dimension> is encoded with an 'A' followed by a <terminated_int> representing the array dimension in decimal.

The syntax for encoding types which represent functions is as follows:

```
<function_type>        ::= 'F' <param_list> ['_' <return_type>]
<param_list>           ::= <param> |
                           <param_list> <param>
<param>                ::= <type>
<return_type>          ::= <type>
```

An 'F' is used to encode a function type. The signature of the function is given by its <param_list>. The parameter list consists of one or more parameters (<param>). A parameter is specified by its <type>.[3]

[3] Two conventions exist for a more concise encoding of repeated parameters, using the following syntax:

```
<param>      ::= <type> |
                 'T' <index> |
```


Note: A function with no parameters should be encoded as having a “void” parameter, e.g., “foo__Fv”.

Note: A function with no explicit parameters other than ellipsis should be encoded as having a “...” parameter, e.g., “foo__Fe”.

When the entity whose name is being mangled is a function, no <return_type> is specified at the end of its <function_type> signature.[4] On the other hand, if a <function_type> is being used to specify a function typed parameter in the <param_list> of another <function_type> a <return_type> must be specified, even if void.

When the name being mangled represents a class or parameterized type it is encoded using the <class_name> syntax discussed above. The syntax for encoding parameterized types is as follows:

```
<parameterized_type>    ::= <int> <pt>
```

Parameterized types are encoded by an <int> representing the length of the parameterized type encoding followed by <pt> which represents the template name and instance parameters. Parameterized types are encoded according to the following syntax:

```
<pt>                    ::= "__PT" <template_name> <pt_param_list>
<template_name>         ::= <l_name>
```

A parameterized type is introduced by "__PT" and is followed by the <template_name> (an <l_name>) and then the <pt_param_list>, the parameter list of the parameterized type. The parameters are encoded according to the following syntax:

```
<pt_param_list>         ::= <pt_param> | <pt_param_list> <pt_param>
<pt_param>              ::= <type> | <expr>
```

A <pt_param_list> consists of a list of one or more parameters, each of which may be either a <type> or an <expr>. Expressions in the parameter lists of parameterized types are encoded according to the following syntax:

```
<expr>                  ::= 'V' <value>
<value>                 ::= <reference> |
                            <num_value>
```

```
                    'N' <rep_count> <index>
 <rep_count> ::= <digit>
 <index>     ::= <digit>
```

Using this scheme, a parameter may either be specified by its <type> or, if its type has already appeared in the parameter list, by a special form indicating that it has the same type as a previous parameter. The 'T' form indicates that the current parameter has the same type as the parameter indicated by <index> (using 1-based indices). The 'N' form indicates that the next <count> parameters have the same type as the parameter indicated by <index>. We choose not to use these conventions because either there must be explicit rules for when they are to be used or else they introduce incompatible names for link compatible functions.

[4] The rationale for omitting the function’s return type signature given in pages 126-7 of *The Annotated C++ Reference Manual* ([1]) is that this allows linkers to identify when there are two functions with the same name and arguments lists but different return types, something which is not allowed in C++.

An <expr> in a <pt_param_list> is encoded using a 'V' followed by a <value>. A value may be encoded in a number of ways depending on its type, according to the following syntax:

```
<reference>             ::= 'R' <count> <characters>
<num_value>             ::= 'N' <count> <characters>
<characters>            ::= <any_char> | <characters> <any_char>
<any_char>              ::= any ASCII char
```

Values may be expressed by addresses or numeric values. A reference value (representing something beginning with '&') is encoded with an 'R' followed by a <count> and a string of characters. Numeric values (<num_value>) are encoded with an 'N' followed by a <count> and a string of characters representing the value after it has been reduced to an integer value.

3.4.1.3 The Grammar for Mangled Names

The following is the complete grammar for mangled names.

```
<mangled_name>          ::= <entity_name> "__" [<class_name>] <type>
<entity_name>           ::= <id> | <special_name>
<special_name>          ::= "__ct" | "__dt" |
                            "_vtbl" | "_rttivtbl" | "_vbtbl" |
                            "__rtti" | "__ti" | "___ti" |
                            "__op" <type> | "__" <op>
<op>                    :: see table

<class_name>            ::= <qualified_name>
<qualified_name>        ::= 'Q' <count> <qual_name_list>
<count>                 ::= <terminated_int>
<int>                   ::= <digit> | <int> <digit>
<digit>                 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<terminated_int>        ::= <int> '_'
<qual_name_list>        ::= <qual_name> | <qual_name_list> <qual_name>
<qual_name>             ::= <l_name> |
                            <parameterized_type>
<l_name>                ::= <int> <id>
<parameterized_type>    ::= <int> <pt>

<type>                  ::= <basic_type> |
                            <qualified_type> |
                            <class_name> |
                            <array_type> |
                            <function_type>

<basic_type>            ::= <int_type> |
                            <other_base_type>
<int_type>              ::= [<sign_mod>] <int_base_type>
<sign_mod>              ::= 'S' | 'U'
<int_base_type>         ::= 'b' | 'c' | 's' | 'i' | 'l' | 'x' | 'w'
<other_base_type>       ::= 'f' | 'd' | 'r' | 'v' | 'e'

<qualified_type>        ::= [<cv>] [<type_qualifier_list>] <type>
<type_qualifier_list>   ::= <type_qualifier> |
                            <type_qualifier_list> <type_qualifier>
<type_qualifier>        ::= 'R' | 'P' | 'M' <type>
<cv>                    ::= 'C' | 'V' | 'CV'
```



```
<array_type>            ::= <dimension_list> <type>
<dimension_list>        ::= <dimension> | <dimension_list> <dimension>
<dimension>             ::= 'A' <terminated_int>

<function_type>         ::= 'F' <param_list> ['_' <return_type>]
<param_list>            ::= <param> | <param_list> <param>
<param>                 ::= <type>
<return_type>           ::= <type>

<pt>                    ::= "__PT" <template_name> <pt_param_list>
<template_name>         ::= <l_name>
<pt_param_list>         ::= <pt_param> | <pt_param_list> <pt_param>
<pt_param>              ::= <type> | <expr>
<expr>                  ::= 'V' <value>
<value>                 ::= <reference> |
                            <num_value>
<reference>             ::= 'R' <count> <characters>
<num_value>             ::= 'N' <count> <characters>
<characters>            ::= <any_char> | <characters> <any_char>
<any_char>              ::= any ASCII char
```

3.4.1.4 Open Issues - Name Mangling

1. Should the places mangling syntax uses an `<int>` to indicate a number be modified to use a `<terminated_int>`, an integer terminated by an underscore, as is already used elsewhere in the grammar? In these cases an int without an underscore is unambiguous; however, there may be too many forms of encoding counts without any clear tradeoffs motivating the different forms.

2. Numeric values in parameterized types need to be represented by canonical strings. Floating point values in particular are problematic due to the need for a standardized, normalized representation.

3. Is there any need to impose limits on lengths of names?

4. Assignment-expressions are now allowed in template definitions to express default values. A mechanism is needed to express this in a mangled name for a template.

5. Name mangling should distinguish between classes that are declared locally in functions since two such classes may generate names (e.g., for v-tables) which conflict at link time.

6. The name mangling rules need to cover how namespaces impact mangling. One simple scheme is let the namespace act like an enclosing class, making use of the existing mechanism for mangling the names of enclosing classes.

7. Categorizing wchar_t as a basic type is a problem since it can change from one implementation to another and can range in size from a char to an unsigned long. Either wchar_t must have a fixed size in the Macintosh API or the mangling must distinguish the possible sizes.

Resolved Issues

1. Should the Pascal attribute on a function be considered to be part of the mangling signature? The Pascal attribute doesn't make any difference in the PowerPC calling conventions; however, it is significant for 68K calling conventions. One suggestion is that functions with

the Pascal attribute should not have their names mangled, but instead the name should be converted to upper case.
Resolution:
The Pascal attribute should be ignored in the mangling of function names for the PowerPC.

3.4.2 Class Object and V-Table Layout

In this section the layout of class objects and vtables are described using a C-like struct syntax.

There are a variety of cases of class object layout depending on whether the class is simple, whether it inherits from one or more other classes, whether it has virtual base classes, and whether it has virtual functions.

3.4.2.1 Simple Classes

```
class X {
      <members>
};

struct {
      <members>
};
```

Members within a class are aligned according to the same rules that govern structure alignment. The alignment will be done according to the alignment mode in effect at the declaration of the class. Consider the following example:

```
#pragma options align=mac68k
class c0 {
      short s;
      long l;
};
#pragma options align=power
class c1 {
      short s;
      long l;
};
```

Given the different alignment modes in effect for each of the declarations, the long in class c0 will be aligned at offset 2, while the long in class c1 will be aligned at offset 4.

3.4.2.2 Single Inheritance Classes

```
class X: <base> {
      <members>
};

struct {
      <base>
      <members>
};
```


A base class is laid out in a derived class as if the first member of the derived class is the struct representing the base class. Subsequent members are laid out following the general rules governing structure alignment with the exception of the case where the base class has no explicit or implicit (e.g., compiler generated table pointers) data members (i.e., it is empty). In this case the subsequent members are aligned where the empty base class begins, rather than following the bytes (depending on the alignment mode in effect) usually allocated for empty structs which are normally not permitted to have zero size.

If the base class or the derived class has virtual functions then the class will begin with a 4-byte entry for a virtual function table pointer, which is discussed in the section on "Virtual Function Table (vtable) Pointers" below.

3.4.2.3 Multiple Inheritance Classes

```
class X: <base1>, ..., <baseN> {
      <members>
};

struct {
      <base1>           // non-virtual direct base classes in lexical order
      ...               //  except that base classes with virtual functions
      <baseN>           //  go first
      <members>
};
```

The base classes are laid out in lexical order, however, all non-virtual bases classes which have virtual functions are moved "to the front" so that sub-objects for them will precede sub-objects for base classes without any virtual functions. This facilitates sharing of the vtable pointer slot. (See section 3.2.6 below.)

3.4.2.4 Virtual Inheritance Classes

```
class X: <base1>, ..., <baseN>, virtual <vbase1>, ..., <vbaseM> {
      <members>
};

struct {
      <vbtable ptr>     // pointer to table of virtual base class offsets
      <word>            // currently unused
      <base1>           // non-virtual direct base classes in lexical order
      ...               //  except that base classes with virtual functions
      <baseN>           //  go first

      <members>         // This class' members

      <vbase1>          // all direct and indirect virtual base class
      ...               //  sub-objects in canonical order
      <vbaseM>
};
```

A derived class with virtual base classes will contain a pointer to a table of virtual base class offsets. Thus, instead of having a pointer for each virtual base class in the object, there is a single

pointer for the virtual base class table.[5] In simple cases, this *vbtable* pointer will be the first member (i.e., its offset will be zero).

The vbtable pointer can have an offset greater than zero if:

- there is a virtual function table, in which case the virtual *function* table pointer slot will precede that of the virtual *base* table pointer, or
- any of the non-virtual direct base classes has a vbtable, in which case the first such base class sub-object's vbtable pointer slot will be used for the derived class's vbtable pointer.

3.4.2.5 Virtual Base Class Offset Tables (vbtables)

Every class with one or more virtual base classes will have a single vbtable object named "_vbtbl__<class_name>", where <class_name> is the mangled name of the owning class.

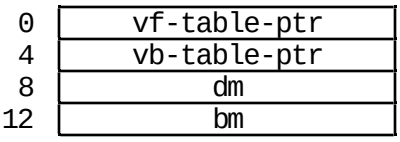
Every vbtable entry is four bytes long. The first entry is always zero, representing the offset of the derived class in itself, and is followed by one entry for each direct or indirect virtual base, containing the virtual base's offset. Each offset is the address of the virtual base object within the most-derived object, relative to the address of the vbtable pointer itself.

Example:

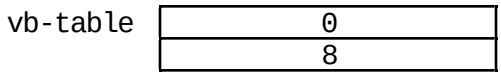
Here we have a simple virtual base class, inherited by a derived class which has a virtual function:

```
struct B { int bm; };
struct D : virtual B {
    virtual void f();
    int dm;
};
```

This will give us a `D` object like this



and a `vb-table` that looks like this:



The **8** means that `D`'s virtual base sub-object `B` is 8 bytes beyond the `D`'s vbtable pointer, or 12 bytes from the start of a `D`.

Another Example:

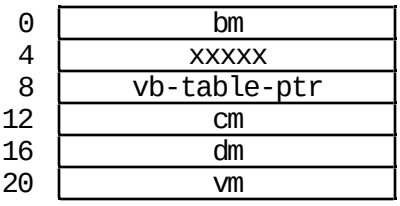
In this case, we have a virtual base class which is inherited indirectly. This allows the derived class to share its vbtable pointer with one of its base classes.

```
struct B { int bm; };
```

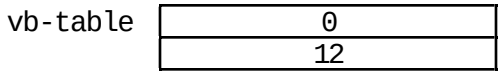
[5] The Metrowerks compiler has a pointer for every virtual base class in the object.


```
struct V { int vm; };

struct C : virtual V { int cm; };

struct D : B, C {
    int dm;
};
```

This `D` object will be laid out as follows:



The `vb-table` for this `D` object will look like this:



The **12** means that `D`'s virtual base sub-object `V` is 12 bytes beyond the `D`'s vbtable pointer, or 20 bytes from the start of a `D`.

3.4.2.6 Virtual Function Table (vtable) Pointers

Every class with any virtual functions will have a virtual function table ("vtable") which is used for indirect dispatch of the virtual functions. Each object of a class with virtual functions will have a virtual function table pointer slot. When present, the vtable pointer slot is allocated at offset zero.

If the first non-virtual base class also has a vtable pointer, the derived class will "share" it by allocating the first base class at offset zero so that its vtable pointer slot will coincide with that of the derived class. If some of the base classes have vtables and others don't, the base classes will be reordered so that the base classes with vtables are first, in order to facilitate the sharing of the vtable pointer. If the derived class has its own vbtable pointer the vtable pointer slot will not be shared. Note that the base classes will be reordered regardless of whether the vtable pointer is actually shared.

The following examples illustrate the various cases of sharing and not sharing vtable pointer slots.

Simple case:

```
class X {
      <members>         // includes one or more virtual functions
};

struct {
      void * <vtblptr>; // pointer to virtual function table
      <members>
};
```


Note that elements that are added implicitly to classes by the compiler (such as vtable pointers) affect the preferred alignment of the class to which they are added. Thus, in the power alignment mode, a class with virtual functions and containing a single char will have a preferred alignment of 4 and a size of 8 (due to the vtable pointer), as opposed to a preferred alignment of 1 and a size of 1 (as would be the case if the class had no virtual functions and no vtable pointer).

Sharing, non-virtual base:

```
class X: <base> {
      <members>          // includes one or more virtual functions
};

struct {
      <base>;            // reuse base's vtable ptr slot
      <members>
};
```

Sharing, reordering of multiple non-virtual bases:

```
class Y {
      <members>          // includes no  virtual functions
};
class Z {
      <members>          // includes one or more virtual functions
};
class X: Y, Z {
      <members>          // includes one or more virtual functions
};

struct {
      <Z>;               // reordered to reuse Z's vtable ptr slot
      <Y>;
      <members>
};
```

Note that if X had virtual base classes, then the vtable pointer should be followed by the vbtable pointer. In this case, X would not share Z's vtable pointer slot because X's vbtable pointer could not immediately follow the vtable pointer since, in order to share Z's vtable pointer, all of Z has to appear at offset zero, conflicting with the required location for X's vbtable pointer.

3.4.2.7 Pointers to Members

Pointers to data members and pointers to member functions require different treatment.

3.4.2.7.1 Pointers to Data Members

Pointers to data members have 3 fields: (1) the data member offset "in its class", (2) the offset to the vbptr, and (3) the virtual base index (always greater than 0 if this is a pointer in a virtual base). Here "in its class" means the offset in the most derived class unless the member comes from a virtual base, in which case the offset is the offset in the virtual base.

```
struct ptdm {
      unsigned int dmoff;     // data member offset "in its class"
```



```
      unsigned int vbix;      // index of vbase in vbtbl (or 0 if no vb)
      unsigned int vbptroff;  // offset to relevant vbtbl ptr, if any
};
```

The algorithm for dereferencing a ptdm is as follows:

```
Deref(ObjectAddress oa, PtrToDataMember ptdm) ==>
      if ptdm.vbix == 0
            // the ptdm is not from a virtual base
            resultAddr = &oa[ptdm.dmoff];
      else
            // the ptdm comes from a virtual base
            vbtpio = oa + ptdm.vbptroff;
            vbt = *vbtpio;
            vbo = vbt[ptdm.vbix];
            vbio = vbtpio + vbo;
            resultAddr = &vbio[ptdm.dmoff];
      fi
```

3.4.2.7.2 Pointers to Member Functions

Pointers to member functions have 4 fields: (1) the address of the function or thunk, (2) the this-ptr-delta, (3) the offset to the vbptr, and (4) the virtual base index (always greater than 0 if this is a pointer in a virtual base).

```
struct ptmf {
      void * address_of_func;
      int this_ptr_delta;     // adjustment to "this" pointer
      unsigned int vbix;      // index of vbase in vbtbl (or 0 if no vb)
      unsigned int vbptroff;  // offset to relevant vbtbl ptr, if any
};
```

The algorithm for calling a ptmf is as follows:

```
CallViaPTMF(ObjectAddress oa, PtrToMemberFunction ptmf, OtherArg...) ==>
      if ptmf .vbix == 0
            // the ptmf is not from a vitual base
            pth = oa;
      else
            // the ptmf comes from a virtual base
            vbtpio = oa + ptmf.vbptroff;
            vbt = *vbtpio;
            vbo = vbt[ptmf.vbix];
            vbio = vbtpio + vbo;
            pth = &vbio[ptmf.dmoff];
      fi
      pth += ptmf.this_ptr_delta;
      CALL(ptmf.address_of_func, pth, OtherArgs...);
```

3.4.2.8 Virtual Function Tables (vtables)

Every polymorphic class will have a single virtual function table named "_vtbl__<class_name>" where <class_name> is the mangled name of the owning class, for example, the vtable for class XXX is "_vtbl__3XXX".

If RTTI is enabled then name of the vtable is "_rttivtbl__<class_name>", and the table has an additional pointer at its start which points to a type_info object. The object's constructor will initialize the vtable pointer by adding 4 to the _rttivtbl__<class_name> symbol, so as to skip the RTTI pointer. This makes the rest of the vtable similar to the non-RTTI case. (For more details see section 3.5 on Runtime Type Information.)

The rest of the vtable (the whole vtable when RTTI is disabled) contains pointers to member functions (or thunks which in turn dispatch to member functions).

A class's vtable is a lexical-order concatenation of the vtables of its direct base classes (not including the RTTI information), followed by the virtual function pointers unique to the class. The virtual function pointers are allocated in lexical order. New virtual function pointers are appended at the end of the vtable. Virtual function pointers for overridden functions reuse the corresponding slot in the area representing the base class vtable.[6] Base classes that have no virtual functions have no vtables and therefore do not have any effect on the vtable of the derived class.

Multiple non-virtual inherited vtables:

```
class X: <base1>, ..., <baseN> {
      <members>          // includes M virtual functions
};

struct {
      <vtable base1>;    // vtables from inherited base classes
      <...>
      <vtable baseN>;
      <vfptr1>;          // X's virtual function pointers
      <...>
      <vfptrM>;
};
```

Virtual inherited vtables:

```
class X: <base1>, ..., <baseN>, virtual <vbase1>, ..., <vbaseM> {
      <members>          // includes L virtual functions
};

struct {
      <vtable base1>;    // vtables from direct base classes
      <...>
      <vtable baseN>;
      <vfptr1>;          // X's virtual function pointers
      <...>
      <vfptrL>;
      <vtable vbase1>;   // all direct & indirect vbtables
      <...>
      <vtable vbaseM>;
};
```

6 The Metrowerks compiler only reuses the slot for an overriden virtual function if it is in a base class with a zero offset. In all other cases a new virtual function pointer is appended to the end of the derived class's vtable.

3.4.2.9 Open Issues - Class Object and Vtable Layout

1. Should a class with nothing in it occupy any space when it is a base case of some other derived class? The base class by itself typically will be assigned a non-zero size. The pertinent rules in the language standard are as follows: (1) two different entities cannot have the same address; (2) however, the first data member of a class/struct can have the same address as the class; and (3) a base class can have the same starting address as the derived class (but perhaps only one base class may do so in the case of multiple inheritance).

2. Is the space savings of having a vbtable worth the additional time overhead of the added indirection to access virtual bases? Do virtual bases occur frequently enough to make the space issue dominate?

3.4.3 Constructors and Destructors

The following sections describe conventions for constructor and destructor functions.

3.4.3.1 Constructors

Constructors take an implicit first argument which is the "this" pointer. They have an implicit return value which is also the "this" pointer. Constructors for classes with virtual base classes have a second implicit argument ("vbasearg") which will be 1 in the most derived constructor call. Base constructor calls (from inside another constructor) will always pass a 0. This allows virtual base classes to be constructed only once.

Compilers implicitly generate constructor calls for base classes and members which are classes. The constructor also sets up the vtable pointers in the object being constructed.

The following examples illustrate the two basic forms of constructors.

Constructor for class with no virtual bases:

```
C::C(<args>) { <user written body> }

C* __ct__<class_name> (C* const this, <args>)
{
      // construct direct bases
      // construct members (as needed)
      // set up vtable pointers (if any)

      <user written body>

      return this;
};
```

Constructor for class with virtual bases:

```
C::C(<args>) { <user written body> }

C* __ct__<class_name> (C* const this, int vbasearg, <args>)
{
      if (vbasearg != 0) {
```



```
            // set up virtual base pointers
            // construct all direct & indirect virtual bases
      }
      // construct direct non-virtual bases
      // construct members (as needed)
      // set up vtable pointers (if any)

      <user written body>

      return this;
};
```

3.4.3.2 Destructors

Destructors take an implicit first argument which is the "this" pointer. They have an implicit return value which is also the "this" pointer.[7]

Compilers implicitly generate destructor calls for base classes and members which are classes.

The following examples illustrate the basic form of destructors.

```
C::~C() { <user written body> }

C* __dt__<class_name> (C* const this)
{
      if (this != NULL) {
            // set up vtable pointers (if any)

            <user written body>

            // destroy members in reverse order
            // destroy direct base classes in reverse order
      }

      return this;
};
```

3.4.3.3 Static Constructors and Destructors

Static objects (those declared at the file level) require a special mechanism to enable their construction and destruction. This mechanism is based on a set of conventions recognized and supported by compilers and linkers.

By convention, compilers will create routines with names prefixed by "__sinit__" and "__sterm__" to identify code that provides for static construction and destruction. Linkers will recognize these names and create an array called "_cdtors". The "_cdtors" array is used by the application's or shared library's startup and termination routines to call the appropriate static ctor and dtor routines.

The "_cdtors" array is made up of pairs of 4-byte pointers. The first element of each pair is the "__sinit__" function pointer for the compilation unit or zero if an initialization function does not

[7] The Metrowerks compiler passes an extra argument to the destructor which tells the destructor to destroy and delete, to destroy without deleting, or to destroy without also destroying the virtual bases.

exist, and the second element of the pair is the "__sterm__" function pointer or zero if a termination function does not exist. The array of pairs is terminated by a pair of 4-byte zeros.

3.4.3.4 Open Issues - Constructors and Destructors

1. The Metrowerks compiler declares the `vbasearg` argument for constructor for classes with virtual bases to be a signed short for compatibility with their 4-byte and 2-byte int models for the 68K. This small change could remove an obstacle to compatibility.

2. The Metrowerks compiler adds an additional signed short argument to destructors which gets interpreted as follows:

   0 destroy non-virtual bases
   -1 destroy all bases
   1 destroy all bases and delete object

   So if the value of this additional argument is 1, the destructor will call `delete` of `this`. This results in space savings in PowerPlant and MacApp applications. This also simplified Metrowerks' exception handling model because all destructors are called in the same way and the exception handler does not have to distinguish between classes with and without virtual bases.

3.4.4 Runtime Type Information (RTTI)

This section describes the representation of Run-Time Type Information (RTTI).

The C++ language definition requires a user-visible form of RTTI. The Macintosh C++ ABI uses the same underlying mechanism for both the user-visible RTTI and the implied RTTI which is required in order to support exception handling.

3.4.4.1 RTTI in C++

The user-visible RTTI functionality consists of `typeinfo, typeid`, and `dynamic_cast`. Implementors usually implement the three other “new-style” casts along with RTTI, because they share the odd syntax that `dynamic_cast` uses. These are `const_cast`, `static_cast`, and `reinterpret_cast`.

3.4.4.1.1 RTTI and Exception Handling

When an exception is thrown, the runtime must be able to determine whether a given handler is capable of handling that exception’s type. Since the normal C++ assignment-compatibility rules are followed, the runtime must be able to tell whether the handler’s type is an ancestor of the thrown exception’s type. This is done by comparing RTTI entries for the two.

There are two levels of RTTI — the low-level RTTI used for exception matching and the user-visible RTTI, i.e., the class `type_info`, defined in `typeinfo.h`.

3.4.4.1.2 RTTI Features

This section contains a brief description of the user-visible aspects of C++ RTTI features.

RTTI is available for any polymorphic type (i.e., any class which has a virtual function table) and for any other type which is used in a `typeid` expression or `dynamic_cast` expression.

`typeid(typename)`
`typeid(expression)`
: These expressions yield a value of type `const type_info &`. The `type_info` class is defined in `typeinfo.h`.

`dynamic_cast<new-type>(` *expression* `)`
: This expression casts the expression (usually a pointer to a base class) to the new type (typically a pointer to a derived class) if the object "really is" of the desired type. If not, the expression returns zero. RTTI is used to determine this.

`class type_info`
: This is the type returned by `typeid` expressions. The only operations allowed on this type are comparison (==, !=), and the member functions `before` and `name`. In particular, the copy constructor and assignment operators are declared to be `private`, so objects of this type cannot be copied or passed to other functions.

3.4.4.1.3 Related Non-RTTI Features (the Other "new-style" Casts)

Following are very brief descriptions of the other new kinds of casts in C++. For more details, see sections 5.2.8, 5.2.9, and 5.2.10 of the C++ committee's draft working papers [3].

`static_cast<new-type>(` *expression* `)`
: This cast is used to render implicit conversions explicit, and is also used for a few other kinds of conversions — to `void`, `int` to `enum`, et al.

`reinterpret_cast<new-type>(` *expression* `)`
: This is the least restrictive cast — in most cases, on most implementations, it will produce a representation unchanged from the representation of the original value. However, this is not mandated: the mapping is implementation-defined.

`const_cast<new-type>(` *expression* `)`
: This cast is most often used to "cast away" `const`; it can also add a `const`, or add or remove a `volatile`.

3.4.4.2 RTTI Data Structures

RTTI is implemented at two levels. The compiler generates a low-level kind of RTTI (sometimes called "pdata") for any type seen in a `throw` or `catch`. For types seen in `typeid` expressions, the compiler will also generate the (user-visible) `type_info` structure, which contains only a pointer to the low-level RTTI for that type.

In addition to the RTTI structures themselves, there are changes in the virtual function tables (vtables) generated for polymorphic types. If the `-rtti` flag is on when the compiler generates the vtable, then that vtable will contain a pointer to the `typeinfo` for that type, and the vtable is given a

name of the form "_rttivtbl__<class>".[8] (Normally, the vtable's name is "_vtbl__<class>".) This is done to ensure that a normal, non-RTTI vtable will not link with code expecting to find that RTTI pointer in the vtable. An RTTI-aware constructor will adjust the class's this pointer to point just past that RTTI pointer.

3.4.4.2.1 High-level — typeinfo structure

The high-level data for the type_info structure consists of a simple pointer which points to the low-level type information for the type in question. The structure is generated as a file-level static variable whose name consists of "__rtti" followed by the mangled name of the type which this structure describes.

3.4.4.2.2 Low-level — pdata

The low-level data (pdata) is also generated as file-level static variables whose names consist of "__ti" followed by the mangled name of the type described.

The pdata consists of a pointer, a tag character, and then other items depending on which tag character is there. Every kind of pdata ends with a string containing the human-readable (non-mangled) name of the described type. The human-readable form of the name should be as it appeared in the declaration of the type with all sequences of "white space" replaced with a single blank.

The pointer points to an unused "common" data byte (i.e., a unique address, of a single byte, used only by the runtime type-matching routine, to compare addresses). That "common" data byte is given a name beginning with three underscores: "___ti<type>". Following that pointer will be a tag character, which then tells what else is in the pdata:

Tag characters

0	simple type (int, char, float, ...), enumerations, and function types
1	pointer or reference
2	class, struct, or union type

The data representations for each of these categories are described in the following sections.

3.4.4.2.2.1 Simple Types and Enumerations

The pdata for a simple type consists of the "common" pointer and the zero tag character, followed immediately by the name of the type. Thus, for instance, the pdata for the unsigned short type will be a variable named __tiUs, and will contain

```
[ pointer to common ___tiUs ], [ 0 tag byte ], [ "unsigned short\0" ]
```

[8] The latest versions of MrCpp generate only the "_rttivtbl" form of vtable, so there is always a slot in the vtable for rtti information. The "_rttivtbl" name is used to prevent inadverntent problems with older code not compiled with new versions of the compiler.

Enumerations are treated as if they were simple types, and the name string that they end with includes the underlying representation type (since the size of an enum type may depend on how many enumerators it contains). So:

```
enum anEnumerationType { anE, anotherE, yaE, lastE };
```

will yield a pdata like this:

```
[ ptr-to-___ti17anEnumerationType ], [ 0 tag byte ],
    [ "unsigned char anEnumerationType\0" ]
```

Function types are also treated as if they were simple types. Example:

```
int func( int, float );
```

yields

```
[ ptr-to-___tiFif ], [ 0 tag ], [ "int \"C++\"(int ,float)\0" ]
```

Note that the "C++" in the function type name indicates C++ linkage.

3.4.4.2.2.2 Pointer and Reference Types

The pdata for pointer types consists of the pointer-to-common, a tag byte (1), a flags byte, a copy of the "pointer-to-common" for the target type, and finally the string describing the pointer type. The flags byte was intended originally to give const/volatile info, but changes in the language definition have made that information irrelevant here, so the flags are currently set to 0x0F always.[9] Example:

```
int *xp;
```

yields

```
[ ptr-to-___tiPi ], [ tag byte 1 ], [ 0x0F flags byte ],
    [ ptr-to-___tii ], [ "int *\0" ]
```

3.4.4.2.2.3 Struct/Class/Union Types

The pdata for structured types (classes, structs, and unions) consists of the pointer-to-common, a tag byte (2), an alignment byte (currently unused, always zero), and a short containing the number of base classes. That is followed by a pair of four-byte words per base class: an offset-from-this (32 bits), and a pointer to the pdata for the base class. (Note: the pointer is to the pdata, i.e., the data with the name with two underscores, not the pointer-to-common for it, which is the pointer RTTI pdatas use.) After all of the base classes, the name of the current class is given, as a null-terminated string.

The list of base classes lists all of the direct base classes (both virtual and non-virtual), in lexical order, followed by the virtual-base sub-objects. There is no easy way to tell exactly how many

[9] The flags byte appears to be a candidate for elision.

items there are in the list. The “short containing the number of base classes” is actually the number of direct bases plus the number of indirect virtual bases. This could probably be improved.

Example:

```
struct A { int a1, a2; };
struct B { int b1, b2; };
struct V { int v1, v2; };
struct W { int w1, w2; };

struct D : A, B, virtual V, virtual W { int d1, d2; };

struct X : virtual V { int x1, x2; };
struct E : A, B, X, virtual W { int e1, e2; };
```

yields the following pdatas for D and E:

```
__ti1D: [ ptr-to-___ti1D ], [ tag byte 2 ], [ unused 0 byte ],
    [ 4 bases ],
    [  8, ptr-to-__ti1A ],
    [ 16, ptr-to-__ti1B ],
    [  0, ptr-to-__ti1V ],
    [  0, ptr-to-__ti1W ],
    [ 32, ptr-to-__ti1V ],
    [ 40, ptr-to-__ti1W ], [ “D\0\0\0” ]

__ti1E: [ ptr-to-___ti1E ], [ tag byte 2 ], [ unused 0 byte ],
    [ 5 bases ],
    [  0, ptr-to-__ti1A ],
    [  8, ptr-to-__ti1B ],
    [ 16, ptr-to-__ti1X ],
    [  0, ptr-to-__ti1W ],
    [ 40, ptr-to-__ti1V ],
    [ 48, ptr-to-__ti1W ], [ “E\0\0\0” ]
```

3.4.4.3 Open Issues - RTTI

1. Does the RTTI mechanism described address cross-DLL issues? How do you throw and "int" exception and catch it as an "int" in a different DLL?

2. How do multiple type info structure actually share a common byte? Is this mechanism necessary or can the address of the low-level data be used?

3. The name strings should be in a read-only section.

3.4.5 Exception Handling

The following sections document the ABI for C++ exception handling.

3.4.5.1 Roles of Compilers, Linkers, and Runtime Libraries

The compiler identifies the life spans of `try` and `catch` blocks, of `new` calls, and of any relevant local variables. These potentially nested ranges are converted into a flattened, disjoint set of code ranges, which are represented in an exception table for each function. Some code ranges have no

actions associated with them and become simple “skip” entries in the range’s table entry. Each “actionable” code range entry gets a list of trys, catches, and destructors associated with it. The tables are generated into separate code sections called “`DB csects`”.

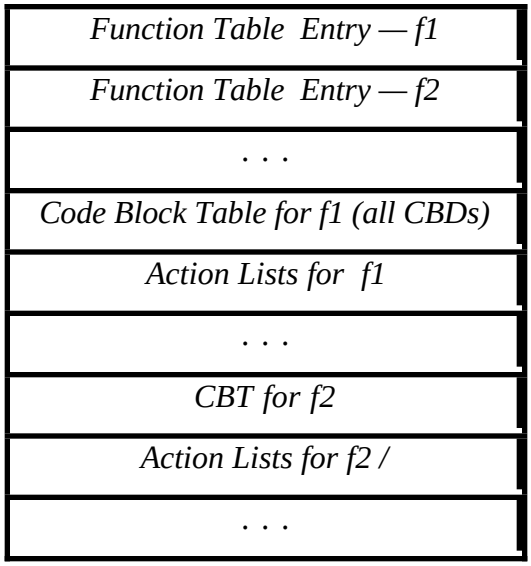
The Linker combines the exception sections and strips the original copies, and sorts the resulting tables in PC address order.

The runtime library contains routines which actually perform the `throw` and the subsequent type-matching, destructor invocation, and stack unwinding.

3.4.5.2 Exception Tables

There are three levels of tables involved: Function tables, Code Block tables, and various kinds of third-level tables. The Function and Code Block tables are language independent; most of the types of third-level tables are C++-specific. All of the tables live in the exception (“`.except`”) section of the PEF container which contains the relevant code. The exception section starts with the first-level table. There is one restriction on the ordering of the other table levels: the first function table entry with a non-zero second-level pointer must point to the second-level entry which appears immediately following the function table. This allows the runtime to find out how many function table entries there are.

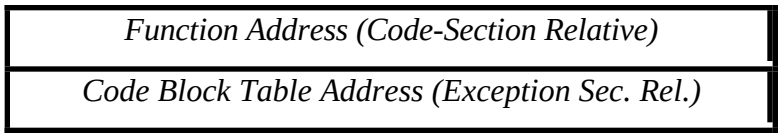
Here is an overview of the table layout:



3.4.5.2.1 Function Tables

The first-level tables are called the *EH Function Tables*. There is one entry per function. The number of entries (i.e., the number of functions) is not specified directly. Instead, the runtime relies on the restriction mentioned above — that the first non-zero CBT address points to the physically first CBT entry, and that CBT entry follows the function table immediately. The runtime can then use that address to determine the number of function table entries.

The linker will ensure that every function in a code fragment has a corresponding entry. Each entry has two 32-bit words:



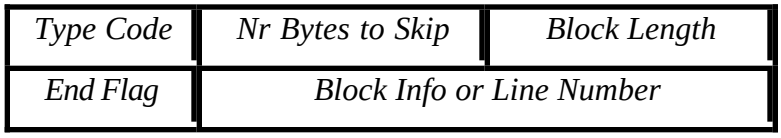

```
struct EH_functionTableEntry {
    address functionStart;
    address CBT_addr;
};
```

The function's address is relative to the beginning of the code section which contains it.

The Code Block Table is the second-level table; the address given here is relative to the beginning of the exception section. If a function has no CBT information, this field is set to zero.

3.4.5.2.2 Code Block Tables

The second-level tables are the *Code Block Tables*. Entries in the CBTs are called *Code Block Descriptors* (CBDs), and are 64 bits long. The entries span the function — i.e., they "flatten" the nested scopes of the variables and `try` blocks which they describe. They are in PC order and must be processed linearly by the exception handling runtime support code. The format of a CBD is:



The one-byte *Type Code* combines with a possible 3rd-level type code, to describe the nature of the third-level table (if any).

The *Number of Bytes to Skip* is a 12-bit field telling how many "unactionable" bytes precede the code range. This allows for a compact table representation of code for which the runtime does not need to perform any actions.

The *Block Length* is a 12-bit field containing the length in bytes of this code range. The start of each code range must be calculated by starting with the function's first CBT entry.

The *End Flag* is a one-byte field; it is set to 5 (`kStop`) or 6 (`kContinue`) to indicate whether this is the last code block in the function.

The 24-bit *Block Info* is a table-relative offset (i.e., offset from the start of the CBT for this function) which usually points to a third level of table entry. The third level table entry formats differ, depending on the values of both the second-level type code and the third- level type code.

Currently, four type codes are defined in the CBD:

0	`kEpilog`	A function epilog — the block info is an unwind descriptor.
1	`kProlog`	The function prolog. Block Info is an unwind descriptor.
3	`kOther`	Other — these kinds of entries have a more complex descriptor.

A function has one prolog, and may have many epilogs. All of the prolog and epilog entries for a given function point to a single *unwind descriptor*.

The other kinds of entries (for `try` blocks, variables to be destroyed, memory to be reclaimed, and exception specifications) are differentiated by their own type codes within the third-level table entries. These are kept distinct from the second-level type code in order to protect the sanity of the implementors. For reference, those third-level type codes are listed here:

4	`kSimple`	An unwind descriptor
8	`kDestruct`	A destructor whose address is locally known
9	`kDestrInd`	An imported destructor
10	`kTry`	A “try” descriptor
11	`kCatch`	A “catch” descriptor
12	`kDelete`	A delete function, locally known
13	`kDelInd`	An imported delete function
14	`kExcSpec`	An exception specification
16	`kClean`	Cleanup, with function to call
17	`kClnInd`	Cleanup, indirect

3.4.5.2.3 Third Level Tables

The third level tables include unwind descriptors which could be language-independent, and a variety of C++-specific tables.

The third-level entries for `try` blocks, for destructor calls, and for `new/delete` calls are collected as *Action Lists*. There is one Action List per code block; it lists all of the actions which are necessary at this particular code range, and all of the `try` blocks enclosing this code range. The ordering of actions on the Action List reflects the nesting of the try blocks and of the scopes of the variables being destroyed. Once the unwinder knows which range applies, it can simply go down

the Action List, performing the relevant destructor and/or delete calls, and then transfer control to the relevant `catch` block. The Action List ends with a `kTry` entry with a zero catch-list offset.

3.4.5.2.3.1 Unwind Descriptors

The *Unwind Descriptors* tell which registers are saved by this function. This allows for the possibility of recovery in the case of asynchronous exceptions. A single type of unwind descriptor is currently defined; its type code is `kSimple`; it is 32 bits long. (Future compiler versions might choose more aggressive optimization of prologs and epilogs, which may require more complex unwind descriptors.)

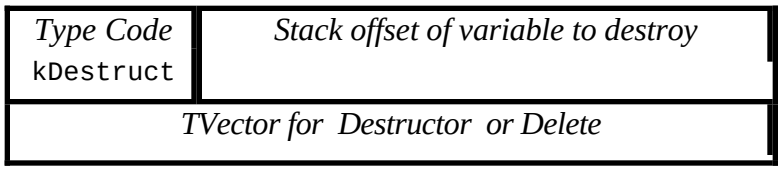
The *Type Code* is `kSimple` (4). This field is one byte wide.

There are 5 unused bits, and then one bit each to tell whether each of the condition register (*CR*), the link register (*LR*), and the stack pointer (*SP*) is saved in this frame.

The final two bytes of the unwind descriptor tell how many floating point and general registers are saved in the frame. In each case, PowerPC conventions determine which registers are saved first, so the simple count of them should be sufficient.

3.4.5.2.3.2 Destructor (TVector) Descriptors

The third-level entries for "normal" destructor calls are 64-bit entries:



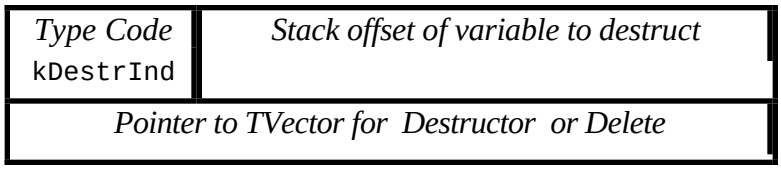
The *Type Code* is `kDestruct` (8) for a locally-known destructor. (That is, a destructor for which we know the address of its transfer vector.) If the destructor is imported from a DLL, it will have type `kDestrInd` (see the following section).

The *stack offset* is a 24-bit field which indicates the address of the variable to be destroyed.

The function to be called is indicated by the *TVector* field, which contains the data section offset for the transfer vector for the destructor.

3.4.5.2.3.3 Destructor (Imported) Descriptors

The third-level entries for destructor calls for destructors imported from other DLLs are 64-bit entries similar to those described in the previous section :



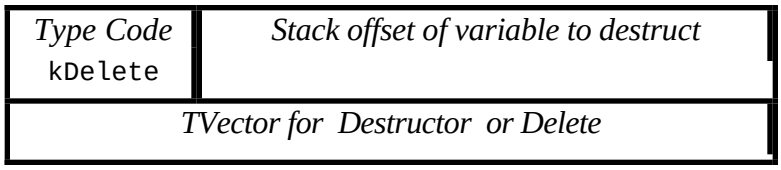
The *Type Code* is `kDestrInd` (10) for a destructor function we are importing.

The *stack offset* is a 24-bit field which indicates the address of the variable to be destroyed.

The function to be called is indicated by the *Pointer to TVector* field, which contains the data section offset for a pointer to the transfer vector for the destructor.

3.4.5.2.3.4 Delete (TVector) Descriptors

The third-level entries for `delete` calls are 64-bit entries, very similar to the corresponding `kDestruct` and `kDestrInd` descriptors.



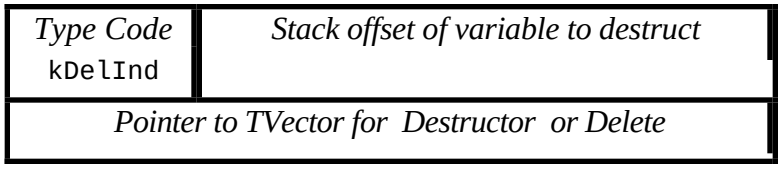
The *Type Code* is `kDelete` (12) for a locally-known delete function. (That is, an `operator delete()` function for which we know the address of its transfer vector.) If the delete function is imported from a DLL, it will have type `kDelInd` (see the following section).

The *stack offset* is a 24-bit field which indicates the address of the variable to be destroyed. (Note that even if the newly-allocated pointer is ultimately assigned only to a global variable, the compiler will also have created a stack-local variable to hold it.)

The delete function is indicated by the *TVector* field, which contains the data section offset for the transfer vector for the `operator delete` to be called.

3.4.5.2.3.5 Delete (Imported) Descriptors

The third-level entries for `delete` functions imported from other DLLs are 64-bit entries similar to those described in the previous section :



The *Type Code* is `kDelInd` (13) for a delete function we are importing.

The *stack offset* is a 24-bit field which indicates the address of the variable to be destroyed.

The delete function is indicated by the *Pointer to TVector* field, which contains the data section offset for a pointer to the transfer vector for the `operator delete` to be called.

3.4.5.2.3.6 Try Descriptors

The third-level entries for `try` blocks are simple 32-bit entries:

Type Code `kTry`	*Catch list offset (CBT-relative)*

The *Type Code* is `kTry` (10).

The 24-bit *catch-list offset* is a CBT-relative offset, pointing to the list of `catch` blocks associated with this `try` block. The `try` descriptors, along with some Destructor/Delete descriptors and `catch` descriptors, will comprise the Action List for a code block.

A `try` descriptor with a catch-list offset of zero indicates the end of the Action List for a code block.

3.4.5.2.3.7 Catch Descriptors

The third-level entries for `catch` blocks are 96-bit (3 “word”) entries:

Type Code `kCatch`	*Function-relative PC*
Run-Time Type Info (RTTI) for type caught	
End Flag	*Catch Variable (stack offset)*

The *Type Code* is `kCatch` (11).

The 24-bit *Function-relative PC* is a pointer to the start of the `catch` block — this is where we jump to after we’ve unwound the stack and initialized the catch variable.

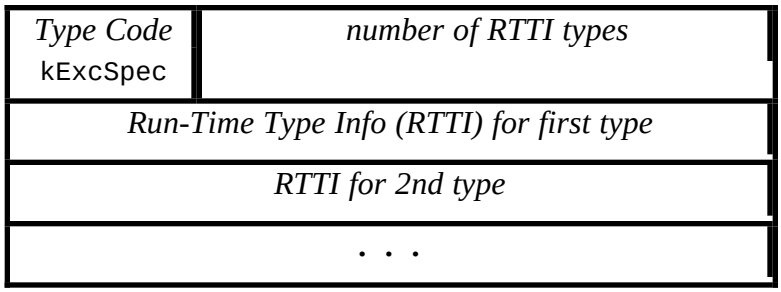
The *RTTI* is a data-section offset to the `typeinfo` structure describing the type that this `catch` block is interested in catching.

The *End Flag* tells whether this `try` block has more catches to check (`kContinue`) or whether this is the last `catch` for this `try` block (`kStop`).

The *Catch Variable* is the stack offset of the variable which we will initialize with the thrown

3.4.5.2.3.8 Exception Specifications

The third-level entry for an exception specification is a variable-length entry. It is considered to be a part of an action list.



The *Type Code* is `kExcSpec` (14).

The *number of RTTI types* tells how many types are in the exception spec for this function.

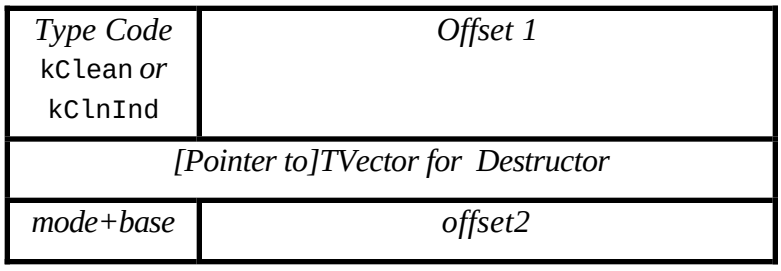
The *RTTI for nth type* is a data section offset of the low-level RTTI structure for this type.

The special case of a "throw nothing" exception specification (i.e., a function which is not permitted to throw any exception, as in `void f() throw() {...}`) is encoded with a count of 1, and an RTTI pointer of zero.

Currently, the exception specification entry happens to be associated with the first code block for a function, but there is no requirement that this remain the case. Since it's considered to be part of an action list, the exception spec entry is usually followed by the end-of-action-list indicator (a `try` descriptor with a catch-list offset of zero).

3.4.5.2.3.9 Cleanup Descriptors

The purpose of a cleanup descriptor is to handle the situation where an exception occurs during the construction of a derived-class object, after some of its base classes are already fully constructed. The bases which have been constructed must be destroyed. This kind of descriptor is more complex than the normal Destructor Descriptor because the address of the base class sub-object might not be at the start of the variable, and might be virtually inherited. These descriptors are actually a bit more general than they currently need to be — basically, they simply tell the exception runtime to call a given function, with a given parameter, with some possible offsets and indirections. The general design should make it possible to keep the same table definitions and library functions, even if our virtual base class strategy changes.



The *Type Code* is `kClean` (16) or `kClnInd` (17) depending on whether the function to be called is known locally. (This is parallel to the difference between `kDestruct` and `kDestrInd` entries.)

The *offset1* entry is used in conjunction with the *mode+base* entry; think of it as the main stack-offset of the parameter to the function which will be called.

The function to be called is indicated by the *TVector* field, which contains the data section offset for the transfer vector for the destructor. (For `kClnInd` descriptors, this field contains the data section offset for a pointer to that TVector.)

The *mode+base* field is 3 bits of addressing-mode information and 5 bits indicating the base register. (Currently, the base register is always the stack pointer.) The possible addressing modes are listed in the following table:

`amBD`	0	Base + unsigned Displacement
`amBS`	2	Base + signed Displacement
`amBDID`	4	Base + unsigned Disp. Indirect, + unsigned Disp.
`amBDIS`	5	Base + unsigned Disp. Indirect, + signed Disp.
`amBSID`	6	Base + signed Disp. Indirect, + unsigned Disp.
`amBSIS`	7	Base + signed Disp. Indirect, + signed Disp.

The first two addressing modes make no use of the *offset2* field. In order to form the argument with which to call the given function, they'll take the base register, and add the *offset1* field, as either a signed or an unsigned value, to the base. The other four will do that, then load a word from the resulting address, and add the *offset2* field (either signed or unsigned) to it.

3.4.5.3 Algorithms

In this initial implementation, we handle only synchronous exceptions. For this case, the initial point of entry into the exception handling mechanism is always a `throw` expression. The compiler implements this as a call to an assembly-language routine called `__eh_throw` (the full signature is "`__eh_throw(const char*, int (*)(), unsigned int, ...)`"). This routine saves all of the registers and then calls `__eh_throw`. A re-throw expression ("`throw;`") is converted by the compiler into a call to `__eh_rethrow__Fv`, which sets the size of the thrown object to a `-1`, and then shares the rest of its implementation with the assembly-language `throw` mentioned above. `__eh_throw` and its support functions are written in C and are defined in the file `mrc_except.c`.

Here's a list of the major routines involved:

```
__eh_throw
__eh_new
FindCatcher

(the table locator functions)

ehDebuggerHook
CallDestructors
```

In the algorithm outlines given below, the type "`addr`" is used to indicate addresses upon which we might have to do address arithmetic, and "`func`" indicates function address.

3.4.5.3.1 `__eh_throw`

Calling sequence:

```
void __eh_throw( rtti_t *rtti, func destructor, int tsize,
        void *tobject, gen_regs, fp_regs, frame, sp );
```


Arguments:

`rtti`	address of the low-level runtime type information string for the type of the object being thrown.
`destructor`	address of the destructor for the object being thrown.
`tsize`	size of the object being thrown (or zero if small object).
`tobject`	pointer to the thrown object (object itself if `tsize` is zero).
`gen_regs`	pointer to the 32 saved general registers
`fp_regs`	pointer to the 32 saved floating-point registers
`frame`	pointer to the frame
`sp`	stack pointer at throw time

Here's a pseudocode summary of what `__eh_throw` does:

```
void __eh_throw ( ... )
{
    peh = __eh_new( allocate new EH stack entry );

    if it's a rethrow {
        if there is no current exception
            terminate();
        set up peh as copy of current exception
    }
    else {
        set up peh with newly thrown exception
    }

    Find the TOC at throw point.
    Call FindCatcher to figure out who will catch this exception.
    if no catcher,
        terminate();

    CallDestructors (for each frame between the throw and the catch).

    FindOwner (ask CFM to locate the container associated with the
        throw point's PC).

    // This loop performs the unwind:
    loop through each frame between thrower and catcher {
        FindCodeBlockTable for that frame
        FindUnwindInfo for the frame

        reload the registers saved in this frame

        find caller
        FindOwner
        if the caller's TOC is different, reset the TOC
    }
    reset sp and linkage register, so that return from this
        function will actually "return" to the catcher.
}
```

3.4.5.3.2 `FindCatcher`

Calling sequence:

```
void FindCatcher( struct frameMarker *catcher,
```



```
                    PCOwnerInfo *ownerText, *ownerData,
                    void *rttiPtr );
```

Arguments:

`catcher`	address of a stack-frame structure, initially set to indicate the frame of the routine that contains the `throw`.
`ownerText`	a CFM structure which `FindCatcher` will set to the container which "owns" the text segment
`ownerData`	who owns the data segment
`rttiPtr`	pointer to the RTTI for the thrown object

Returns:

`FindCatcher` sets the `catcher` structure to the frame info for the routine which will catch this exception. `catcher->reserved1` will be the stack offset of the catch variable; it will be zero if no handler is found.

```
void FindCatcher ( catcher, ownerText, ownerData, rttiPtr )
{
    FindOwner( PC, ownerText );
    FindOwner( TOC, ownerData );

    loop until we find a frame with no unwind info {
        cbbase = FindCodeBlockTable( ownerText, PC, &funcPC );
        actionList = GetActionList( cbbase, PC - funcPC );

        loop through this action list {
            if this is a TRY, and its PC is within range,
                loop through its catches {
                    if this catch matches {
                        set up the catcher struct
                        return;
                    }
                }
        }
        move up to next frame
    }
    set catcher->reserved1 = 0;  (couldn't find a catcher)
}
```

3.4.5.3.3 Table locators

These functions (`FindOwner`, `GetFunctionTable`, `FindCodeBlockTable`, `FindUnwindInfo`, `FindDestrInfo`, and `GetActionList`) locate the tables which are relevant to the current exception and program state.

`FindOwner` — given an address, call the CFM routine `FragFindOwnerOfPC` to locate the CFM container associated with that code (or data) fragment.

`GetFunctionTable` — look through the sections of a CFM container, to locate its (unique) exception-tables section.

`FindCodeBlockTable` — search the function table for a given code section, looking for the code block table associated with a given PC. (Currently, this is a linear search.)

`FindUnwindInfo` — given a code block table pointer, find a `kProlog` entry, which will give us the "how-to-unwind" info for the function.

`FindDestrInfo` — given an action list, return the next destructor entry in the list. Update the given action list pointer.

`GetActionList` — given a PC offset and a code block table, search for an action list that's relevant to the given PC. Return zero if there is no such action list.

3.4.5.3.4 Other Support Routines

`ehDebuggerHook`— at certain points during exception handling, the library will call this routine, passing it a reason code. The debugger can set a breakpoint in this routine, and can sometimes examine the extra parameters (two frame pointers and an RTTI pointer) in order to implement debugger commands like "stop on throw" and "who will catch?".

`CallDestructors`— this function walks the stack between the `throw` and the relevant catcher-frame, and calls the destructors for each relevant action list in each frame between the two endpoints.

3.4.5.4 Future Directions

These table formats are not yet fully stabilized. We hope to support exceptions in other languages, and some limited forms of asynchronous exceptions. More details on these areas will be in a future revision of this document.

3.4.5.5 Open Issues - Exceptions

1. Is the exception handling model flexible enough to handle other languages? Is there a provision for handling machine exceptions? If RTTI structures are used in the location of handlers, what is the RTTI description of an access fault?

2. Is the exception handling model thread-safe? Is `__eh_new` a memory allocation function and does this present a problem?

3. Is the exception handling model really a zero-overhead model? All exception action descriptors seem to expect variables in memory, so the `this` pointer in a constructor cannot be a register variable (or it will have to be copied into memory) if it has base classes so that a partially constructed object can be destroyed.

4. Is there a provision for exception actions for conditional temporary destructions? For example, the temporaries that have to be constructed in the right part of an "a && b" expression are only constructed and destroyed if "a" is true.

5. The exception tables need to be redesigned with scalability issues in mind, so that very large programs/DLLs may be handled while providing space-efficient representations for smaller code fragments.

6. The topmost table needs a header containing version information, overall sizes, etc.

7. The exception mechanism has no support for multiple code and data sections within a DLL.

3.4.6 Special C++ Calling Conventions

There are several constructs which require special calling conventions in the Macintosh C++ ABI. In particular, these are cases where there are implicit parameters or parameters that must treated specially.

3.4.6.1 Passing Objects By Value

In the simplest cases, parameters which are objects (class instances) are passed by value as if they were simple structures. However, if the object is an instance of a class for which there is an explicit copy constructor or a destructor then a temporary copy of the object must be made and passed by reference. If there is a copy constructor it will be used to construct the temporary, and if there is a destructor it will be used to destruct the temporary after the evaluation of the expression containing the call is complete.

The requirements above are dictated by the C++ Standard, Section 12.2, which indicates:

> When an implementation introduces a temporary object of a class that has a non-trivial constructor, it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary object with a non-trivial destructor. Temporary objects are destroyed as the last step in evaluating the full expression that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception.
>
> ... A temporary bound to a reference parameter in a function call persists until the completion of the full expression containing the call.

The name of the called routine will be mangled as usual, even if it requires that a temporary be created and passed by reference.

3.4.6.2 Order of "this" and "Hidden" Parameters

Functions which return structures require a "hidden" parameter which is a pointer to the temporary memory into which the function return value will be placed. Such hidden parameters are always passed as the first parameter to the function. A hidden parameter is always used when the function return value is a structure, regardless of its size.[10]

C++ member functions require an implicit parameter for the value of the object's "this" variable. The implicit "this" parameter is passed as the first parameter unless there is a "hidden" parameter for a structured return value in which case the "this" parameter is passed as the second parameter.

---

10 A possible optimization of the ABI would be to pass back structured return values in registers when the structure is small enough to fit in a register. Such optimizations are not allowed because they violate the ABI; the ABI must change to permit them, in which case they would be required.

3.4.6.3 Thunks

In virtual function dispatch, a vtable entry may be a pointer to a thunk rather than to a virtual member function. A thunk is a small piece of code which performs a minor adjustment (such as modification of the "this" pointer value or saving of the current TOC) and then branches to the "real" intended function.

There must be agreement between compilers about the circumstances under which thunks are generated and what function(s) they perform in order for C++ object files produced by different compilers to be link compatible.

The naming convention for thunks should not be a problem since they should be emitted at the time that the vtable is generated. The vtable will have pointers to the thunks which in turn will have branches to the appropriate routines. Since the vtable and thunks for a class are generated in a simple compilation the names will be consistent regardless of the convention used.

Modification of the "this" pointer takes place when a base class pointer is used for a virtual function call that does not correspond to the address of the derived class (in other words, the base class is not the first base class in the derived class). On the PowerPC such a thunk might look like the following code:

```
subi  r3, r3, <delta>   // change the "this" pointer argument
b     <vfunc>           // branch to the intended function
```

3.4.6.4 Proposal - C++ Virtual Function Dispatch

This is a proposal to change the calling conventions for C++ virtual functions. The C++ virtual function calling conventions are open to change because they can be distinguished from normal calls by the compiler, and because such calls depend on other compiler-dependent conventions such as vtable layouts. We are soliciting input on the merits of this proposal and hope to gather some data on its effects on some large code samples.

In this scheme R2 becomes a nonvolatile register, saved and restored on the callee side. The callee accesses its R2 via the R12 pointer to its TVector. This is fine for virtual functions since they always use indirect calls (see local monomorphic optimization below).

Proposed model

The code below summarizes the R2 switching today and as proposed.

```
Today                                   Proposal

caller   (TVPtr in R12)                 caller   (TVPtr in R12)

   bl            ProcPtrGlue               bl           VirtPtrGlue
   lwz           R2, 20(SP)

ProcPtrGlue                             VirtPtrGlue

   lwz           R0, 0(R12)                lwz          R0, 0(R12)
   stw           R2, 20(SP)                mtctr        R0
   mtctr         R0                        bctr
   lwz           R2, 4(R12)
   bctr
```



```
callee with globals                    callee with globals

   <nothing extra>                        stw          R2, 20(SP)
                                          lwz          R2, 4(R12)
                                          ...
                                          lwz          R2, 20(SP)

callee without globals                 callee without globals

   <nothing extra>                        <nothing extra>
```

Note that direct calls to virtual functions (e.g., cases where the base class is expressed explicitly in the call) will have to either make an indirect call or perform the equivalent operations.

Possible optimizations

The save of the old R2 and load of the new R2 can be freely scheduled, as long as an exception dispatcher and debugger know enough to unwind the stack properly. The same holds for the restore of the old R2.

Recognizing whether the callee uses R2 or not is an obvious optimization opportunity. This may very well apply to a large fraction of virtual functions. Note that this really is whether or not this one routine uses globals. In particular, it may very well make calls to other virtual functions without having to load its own R2.

The shorter virtual function glue is more amenable to inlining. For this purpose the TVector contents should be viewed as nonvolatile, at least within the current function.

A possible, but unlikely optimization can be made for monomorphic methods that are known to be in the same DLL. Instead of doing the "bl VirtPtrGlue" you do "bl .LocalMonomorph".

Effect on space

The proposed model saves 4 bytes per call site. It costs 12 bytes per virtual function that uses globals. The glue savings of 8 bytes per DLL are not worth counting.

If every virtual function uses globals the break-even point is at an average of 3 calls per function. I.e. if you make 3 outgoing calls you've paid for your access to globals. If you make 5 outgoing calls you've paid for your global access and that of another function that only makes 1 call.

If no virtual functions use globals then we are ahead by 4 times the number of call sites.

If 1/3 of the virtual functions use globals the break-even point is at an average of 1 call per function.

Effect on time

The space savings is probably the most important time savings too, since space is time. The R2 switching code in the ProcPtrGlue falls into delay slots, so essentially has no cost other than cache effects. The same can also be said of properly scheduled switching code in the callee. Saving the R2 reload on return is constant, but probably not significant overall. Note though that a well scheduled reload in the callee will significantly mitigate cache misses compared to the caller side reload.

Another significant opportunity to save time can come from use of inlined glue for inner loops.

3.4.6.5 Open Issues - C++ Calling Conventions

1. Consider defining vtable formats and thunks for various special cases, just as SOM does. Anyone want to provide some specific proposals?

2. Can anything be gained by adding extra fields to the TVector, such as data for thunks?

3.4.7 Miscellaneous C++ ABI Issues

There are a some implementation-dependent language features in C++ which will affect whether the outputs of different C++ compilers are link compatible.

3.4.7.1 The Type of size_t

There must be agreement between compilers about the type of size_t (i.e., is it an unsigned int or an unsigned long) in order for C++ object files produced by different compilers to be link compatible. Currently MrC expects it to be an unsigned int while the Metrowerks compiler expects it to be an unsigned long. Given that "int" may represent something smaller than 4 bytes in some compilers, it may be preferable to have size_t be an unsigned long.

3.4.7.2 Trigger Members for Vtable Generation

Most compilers have a strategy based on "trigger members" or "trigger functions" for when to generate the data definition of the vtable for a class so that vtable definitions are not generated in every compilation unit that uses the class. One data member or function in the class is identified as the trigger member, and, when the definition of that member is encountered in a compilation unit the compiler will emit the definition for the vtable. This (usually) ensures that only one definition of the vtable is emitted.

The Macintosh C++ ABI identifies the first declared function which is not inlined in the class to be the trigger function.[11]

3.4.8 Open Issues - C++ ABI

1. Enum sizes.
2. Static constructor/destructor ordering--are DLLs a problem case?

---

[11] The Metrowerks compiler uses different rules to trigger the generation of vtables. The situation in the Metrowerks environment is somewhat different in that the Metrowerks linker knows to merge multiple vtable definitions within an application; thus their rules for the generation of vtables are more "libera;" and emit vtables in more cases. The Metrowerks compiler identifies the "trigger member" to be the lexically first static data or function member not defined within the class, or the lexically first virtual function member not defined within the class. If there are no static or virtual members, vtables are generated if the vtable is referenced. Also if a virtual or static trigger function is defined as inline outside of the class definition a vtable definition will be emitted.

4. References

[1] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.

[2] *Macintosh Runtime Architectures*, Apple Computer. Inc., 1996 (forthcoming).

[3] "Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++", American National Standards Institute (ANSI), Oct. 1996.

[4] Erik Eidt, Alan Lillich, "C Compiler Pragmas for Macintosh “CFM” Runtime", Apple Computer, 1994.

[5] "AIX XCOFF Object and Load Module Format for IBM RISC System/6000", IBM, 1992.

A1. Appendix 1: Compiler Pragmas Affecting the ABI

The pragmas identified in this appendix affect generated code at the ABI level.

For the purpose of compatibility at the language (or API) level, it is desirable to have all compilers to be consistent in their implementation of these pragmas.

A1.1 Alignment Mode Pragma

The alignment mode is changed by a compiler command line option or by the use of the following pragma:

```
#pragma options align=<alignment_specifier>
```

where

```
<alignment_specifier>[12] ::= power | mac68k | packed | natural | reset
```

The “power” alignment specifier establishes the “powerpc” alignment mode. The "mac68k" specifier establishes the 68000 Macintosh alignment conventions used by much of the original Macintosh Toolbox. The "packed" specifier establishes an alignment mode in which no padding is used and fields are packed into the minimum space possible. The "natural" specifier establishes a variant of the "powerpc" alignment mode in which all data types including doubles and long doubles are aligned according to their "natural" alignment.

The “reset” specifier changes the alignment mode back to the alignment mode in effect when the current alignment mode was set, or to the default mode if no other was specified. Thus, “reset” does a “pop” of the alignment mode, while the other options do a “push”. An arbitrary nesting of modes is allowed.

The alignment mode used for an aggregate type definition is the mode that is in effect at the beginning of the defining declaration of the type. If there is a different alignment mode in effect for an incomplete declaration, then that mode has no effect on the eventual mode used to complete the

[12] At one time a “native” `<alignment_specifier>` was supported as a synonym for “powerpc”; however, since the term is ambiguous for different host and target machine configurations, its use has been deprecated and discontinued.

aggregate definition. Changing the alignment mode in the middle of a type definition does not change the alignment mode to be used for the type.[13]

A1.2 CFM Pragmas

The CFM pragmas are documented in "C Compiler Pragmas for Macintosh “CFM” Runtime" [4].

A1.3 Open Issues - Compiler Pragmas Affecting the ABI

1. It may be desirable to have an additional alignment pragma of the following form:

   #pragma alignment <alignment_specifier>

   This pragma would have restrictions on its placement within source code, so that it could not appear inside a definition.

[13] Changing alignment modes in the middle of a type definition is not a recommended programming practice. This is an area where compilers are known to have problems conforming to the alignment rules.

