C++ ABI for the ARM® Architecture

Development systems Division

Compiler Tools Group

Document number:	GENC-003540
Date of Issue:	30th October 2003
Author:	
Authorized by:	

Abstract

This document describes the C++ Application Binary Interface for the ARM architecture.

Keywords

C++ ABI, generic C++ ABI, exception handling ABI

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This document has been released publicly. Anticipated changes to this document include:

- Typographical corrections.
- Clarifications.
- Compatible extensions.

1.1.2 Change history

Issue	Date	By	Change
1.0	30th October 2003	Lee Smith	First public release.

1.2 References

This document refers to, or is referred to by, the following documents.

Ref	URL or other reference	Title
AAPCS		Procedure Call Standard for the ARM Architecture
BSABI		ABI for the ARM Architecture (Base Standard)
CPPABI		C++ ABI for the ARM Architecture (*This document*)
EHABI		Exception Handling ABI for the ARM Architecture
EHEGI		Exception handling components, example implementations
GC++ABI	http://www.codesourcery.com/cxx-abi/abi.html	Itanium C++ ABI ($Revision: 1.71 $) (Although called *Itanium C++ ABI*, it is very generic).
ISO C++	ISO/IEC 14882:1998	International Standard ISO/IEC 14882:1998 – Programming languages C++

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
ABI	Application Binary Interface: 1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the ARM Architecture*. 2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the *C++ ABI for the ARM Architecture*, the *Run-time ABI for the ARM Architecture*, the *C Library ABI for the ARM Architecture.*
AEABI	EABI (see below) for the ARM Architecture, *this* [E]ABI.
ARM-based	… based on the ARM architecture …
EABI	An ABI suited to the needs of embedded, and deeply embedded (sometimes called *free standing*), applications.

1.4 About the licence to use this specification

Use of these *ABI for the ARM Architecture* specifications published by ARM is governed by the simple licence agreement shown on the cover page of this document, and on the cover page of each major component document. Without formalities or payment, you are licensed to use any IP rights ARM might hold in these ABI specifications for the purpose of producing products that comply with these ABI specifications.

Because these specifications may be updated by ARM without notice, we prefer that these specifications should not be copied, but that third parties should refer directly to them, in the same way that we refer directly to the specifications underpinning this ABI, such as the specifications of ELF, DWARF, and the generic C++ ABI.

1.5 Acknowledgements

This specification could not have been developed without contributions from, and the active support of, the following organizations. In alphabetical order: ARM, Intel, Metrowerks, Montavista, Nexus Electronics, PalmSource, Symbian, and Wind River.

2 OVERVIEW

The C++ ABI for the ARM architecture (CPPABI) comprises four sub-components.

- The generic C++ ABI, summarized in §2.1, is the referenced base standard for this component.
- The *C++ ABI supplement*, summarized in §3, details ARM-specific deviations from the generic standard.
- The separately documented *Exception Handling ABI for the ARM Architecture* (EHABI), summarized in §2.2, describes the language-independent and C++-specific aspects of exception handling.
- The example implementations of the exception handling components [EHEGI], summarized in §2.3, include:
  - A language independent unwinder.
  - A C++ semantics module.
  - ARM-specific C++ unwinding personality routines.

2.1 The Generic C++ ABI

The generic C++ ABI [GC++ABI] (originally developed for Itanium) specifies:

- The layout of C++ non-POD class types in terms of the layout of POD types (specified for *this* ABI by the *Procedure Call Standard for the ARM Architecture* [AAPCS]).
- How class types requiring copy construction are passed as parameters and results.
- The content of run-time type information (RTTI).
- Necessary APIs for object construction and destruction.
- How names with linkage are mangled (name mangling).

The generic C++ ABI refers to a separate Itanium-specific specification of exception handling. When the generic C++ ABI is used as a component of *this*, corresponding reference must be made to the *Exception Handling ABI for the ARM Architecture* [EHABI] and §2.2.

2.2 The Exception handling ABI for the ARM architecture

In common with the Itanium exception handling ABI, the *Exception handling ABI for the ARM architecture* [EHABI] specifies table-based unwinding that separates language-independent unwinding from language specific aspects. The specification describes:

- The *base class* format and meaning of the tables understood by the language independent exception handling system, and their representation in object files. The language independent exception handler only uses fields from the base class.
- A *derived table class* used by ARM tools that efficiently encodes stack-unwinding instructions and compactly represents the data needed for handling C++ exceptions.
- The interface between the language independent exception handling system and the *personality routines* specific to a particular implementation for a particular language. Personality routines interpret the language specific, derived class tables. Conceptually (though not literally, for reasons of implementation convenience and run-time efficiency), personality routines are member functions of the derived class.

- The interfaces between the (C++) language exception handling semantics module and
  - The language independent exception handling system.
  - The personality routines.
  - The (C++) application code (effectively the interface underlying *throw*).

The EHABI contains a significant amount of commentary to aid and support independent implementation of:

- Personality routines.
- The language specific exception handling semantics module.
- Language independent exception handling.

This commentary does not provide, and is not intended to provide, complete specifications of independent implementations, but it does give a rationale for the interfaces to, and among, these components.

2.3 The exception handling components example implementation

The exception handling components example implementation (EHEGI) comprises six files, as follows.

- **cppsemantics.cpp** is a module that implements the semantics of C++ exception handling. It uses the language independent unwinder (unwinder.c), and is used by the ARM-specific personality routines (unwind_pr.[ch]).
- **cxxabi.h** describes the generic C++ ABI (§2.1).
- **unwind_env.h** is a header that describes the build and execution environments of the exception handling components. This header must be edited if the exception handling components are to be built with non-ARM compilers. This header #includes cxxabi.h.
- **unwind_pr.c** implements the three ARM-specific personality routines described in the *Exception Handling ABI for the ARM Architecture*.
- **unwinder.c** is an implementation of the language independent unwinder.
- **unwinder.h** describes the interface to the language independent unwinder, as described in the *Exception Handling ABI for the ARM Architecture*.

3 THE C++ ABI SUPPLEMENT

3.1 Summary of differences from the generic C++ ABI

This section summarizes the differences between the C++ ABI for the ARM architecture and the generic C++ ABI. Section numbers in captions refer to the generic C++ ABI specification. Larger differences are detailed in subsections of §3.2.

GC++ABI §1.2 ***Limits***

The offset of a non-virtual base sub-object in the full object containing it must fit into a 24-bit signed integer (due to RTTI implementation). This implies a practical limit of 2^{23} bytes on the size of a class sub-object.

GC++ABI §2.3 ***Member Pointers***

The pointer to member function representation differs from that used by Itanium. See §3.2.1.

GC++ABI §2.4 **¶ II.1** ***[layout of bit-fields]***

Oversized bit-fields are laid out differently. See §7.1.5.3 of AAPCS.

GC++ABI §2.7 ***Array operator new cookies***

Array cookies, when present, are always 8 bytes long and contain both element size and element count (in that order). See §3.2.2.

GC++ABI §2.8 ***Initialization guard variables***

Static initialization guard variables are 4 bytes long not 8, and there is a different protocol for using them which allows a guard variable to implement a semaphore when used as the target of ARM SWP or LDREX and STREX instructions. See §3.2.3.

GC++ABI §3.1.5 ***Constructor return values***

Constructors (complete, sub-object and allocating) and destructors (complete and sub-object) return the address of the object constructed or destroyed.

GC++ABI §3.3.2 ***One-time construction API***

The type of parameters to __cxa_guard_acquire, __cxa_guard_release and __cxa_guard_abort is 'int*' (not '__int64_t*'), and use of fields in the guard variable differs. See §3.2.3.

GC++ABI §3.3.4 ***Controlling Object Construction Order***

#pragma priority is not supported. See §3.2.4 for details of how global object construction is coordinated.

GC++ABI §3.3.5.3 ***Runtime API***

This ABI defines __aeabi_atexit (§3.2.2.5), for use in preference to __cxa_atexit. In addition it is forbidden for user code to call __cxa_atexit or __aeabi_atexit, or for any call to __aeabi_atexit to be executed more than once (see § 3.2.4 for an explanation of this restriction).

GC++ABI §3.4 ***Demangler API***

The demangler is not provided as a library.

GC++ABI §5.2.3 ***Virtual Tables and the key function***

The *key function* is the textually first, non-inline, non-pure, virtual function identified at the end of the **translation unit**. An inline member function is not a key function even if it is first declared inline after the class definition has been completed.

(The Itanium ABI defines the key function to be the textually first, non-inline, non-pure, virtual function identified at the end of the **class definition**).

GC++ABI §5.3 ***Unwind Table Location***

See §4 of *Exception Handling ABI for the ARM Architecture* [EHABI].

(No section in the generic C++ ABI)

Library versions of the following functions *must not* examine their second argument.

```
::operator new(std::size_t, const std::nothrow_t&)
::operator new[](std::size_t, const std::nothrow_t&)
```

(The second argument conveys no useful information other than though its presence or absence, which is manifest in the mangling of the name of the function. This ABI therefore allows code generators to use a potentially invalid second argument – for example, whatever value happens to be in R1 – at a point of call).

(No section in the generic C++ ABI, but would be §2.2 ***POD data types)***

Pointers to extern "C++" functions and pointers to extern "C" functions are interchangeable if the function types are otherwise identical.

In order to be used by the library helper functions described below, implementations of constructor and destructor functions (complete, sub-object, and allocating) must have a type compatible with:

```
extern "C" void* (*)(void* /* , other argument types if any */);
```

Deleting destructors must have a type compatible with:

```
extern "C" void (*)(void*);
```

(No section in the generic C++ ABI, but would be §3.3.4 ***Controlling Object Construction Order)***

Global object construction and destruction are managed in a simplified way under this ABI (see §3.2.4).

3.2 Differences in detail

3.2.1 Representation of pointer to member function

The generic C++ ABI [GC++ABI] specifies that a pointer to member function is a pair of words <*ptr*, *adj*>. The least significant bit of *ptr* discriminates between (0) the address of a non-virtual member function and (1) the offset in the class's virtual table of the address of a virtual function.

This encoding cannot work for the ARM-Thumb instruction set where code addresses use all 32 bits of *ptr*.

This ABI specifies that *adj* contains twice the *this* adjustment, plus 1 if the member function is virtual. The least significant bit of *adj* then makes exactly the same discrimination as the least significant bit of *ptr* does for Itanium.

A pointer to member function is NULL when *ptr* = 0 *and* the least significant bit of *adj* is zero.

3.2.2 Array construction and destruction

3.2.2.1 Array cookies

An array cookie is used for heap-allocated arrays of objects with class type where the class has a destructor or the class's *usual (array) deallocation function* [ISO C++ §3.7.3.2] has two arguments, i.e. `T::operator delete(void*, std::size_t)`. Nonetheless, an array cookie is not used if `::operator new[](std::size_t, void*)` is used for the allocation as the user is then responsible for the deallocation and the associated bookkeeping.

When a cookie is needed this ABI always specifies the same cookie type:

```
struct array_cookie {
    std::size_t element_size; // element_size != 0
    std::size_t element_count;
};
```

This is different than the generic C++ ABI which uses a variable sized cookie depending on the alignment of element type of the array being allocated.

Note Although it's not a particularly useful property, this cookie is usable as a generic C++ cookie when the generic C++ cookie size is 8 bytes.

Both the element size and element count are recorded in the cookie. For example, in the following the element size would be sizeof(S) = 8 and the element count would be 3 * 5 = 15.

```
struct S { int a[2]; };
typedef SA S[3];
S* s = new SA[5];
```

Note The element size can never legally be zero. Finding a zero element size at delete [ ] time indicates heap corruption.

3.2.2.2 Array cookie alignment

The array cookie is allocated at an 8-byte aligned address immediately preceding the user's array. Since the cookie size is 8 bytes the user's array is also 8-byte aligned.

3.2.2.3 Library helper functions

The generic C++ ABI contains some helper functions for array construction and destruction:

```
__cxa_vec_new          __cxa_vec_new2
__cxa_vec_new3         __cxa_vec_ctor
__cxa_vec_dtor         __cxa_vec_cleanup
__cxa_vec_delete       __cxa_vec_delete2
__cxa_vec_delete3      __cxa_vec_cctor
```

Compilers are not required to use these helper functions but runtime libraries must supply them and they must work with the always 8-byte cookies. These functions take pointers to constructors or destructors. Since constructors and destructors conforming to this ABI return *this* (§3.1, ¶§3.1.5 *Constructor return values*, above) the return types of these parameters are void* instead of void.

The generic C++ ABI gives __cxa_vec_ctor and __cxa_vec_cctor a void return type. This ABI specifies void* instead. The value returned is the same as the first parameter – a pointer to the array being constructed. We do not change the return type for __cxa_vec_dtor because we provide __aeabi_vec_dtor which has the additional advantage of not taking a padding_size parameter.

In addition, we define the following new helpers which can be called more efficiently.

```
__aeabi_vec_ctor_nocookie_nodtor
__aeabi_vec_ctor_cookie_nodtor
__aeabi_vec_cctor_nocookie_nodtor
__aeabi_vec_new_cookie_noctor
__aeabi_vec_new_nocookie
__aeabi_vec_new_cookie_nodtor
__aeabi_vec_new_cookie
__aeabi_vec_dtor
__aeabi_vec_dtor_cookie
__aeabi_vec_delete
__aeabi_vec_delete3
__aeabi_vec_delete3_nodtor
__aeabi_atexit
```

Again, compilers are not required to use these functions but runtime libraries must supply them.

__aeabi_vec_dtor effectively makes __cxa_vec_dtor obsolete.

Compilers are encouraged to use the __aeabi_vec_dtor instead of __cxa_vec_dtor and __aeabi_vec_delete instead of __cxa_vec_delete. Run-time environments are encouraged expect this, perhaps implementing __cxa_vec_delete in terms of __aeabi_vec_delete instead of the other way around.

We provide __aeabi_delete3 but not __aeabi_delete2. Using __aeabi_delete2 is less efficient than using __aeabi_vec_dtor and calling the `T1::operator delete[]` directly. See note 3 on page 15, below.

__cxa_vec_ctor still has uses not covered by __aeabi_vec_ctor_nocookie_nodtor and __aeabi_vec_ctor_cookie_nodtor.

Additional helpers for array construction (i.e. new T[n], __aeabi_vec_new_*) may be added in future releases of this ABI.

Definitions of the __aeabi_* functions are given below in terms of example implementations. It is not required to implement them this way.

```
#include <cstddef>  // for ::std::size_t
#include <cxxabi.h> // for __cxa_*

namespace __aeabiv1 {
  using ::std::size_t;

  // Note: Only the __aeabi_* names are exported.
  // array_cookie, cookie_size, cookie_of, etc. are presented for exposition only.
  // They are not expected to be available to users, but implementers may find them useful.

  struct array_cookie {
      size_t element_size; // element_size != 0
      size_t element_count;
  };

  // The struct array_cookie fields and the arguments element_size and element_count
  // are ordered for convenient use of LDRD/STRD on architecture 5TE and above.

  const size_t cookie_size = sizeof(array_cookie);

  // cookie_of() takes a pointer to the user array and returns a reference to the cookie.

  inline array_cookie& cookie_of(void* user_array)
  {
    return reinterpret_cast<array_cookie*>(user_array)[-1];
  }

  // element_size_of() takes a pointer to the user array and returns a reference to the
  // element_size field of the cookie.

  inline size_t& element_size_of(void* user_array)
  {
    return cookie_of(user_array).element_size;
  }

  // element_count_of() takes a pointer to the user array and returns a reference to the
  // element_count field of the cookie.

  inline size_t& element_count_of(void* user_array)
  {
    return cookie_of(user_array).element_count;
  }

  // user_array_of() takes a pointer to the cookie and returns a pointer to the user array.

  inline void* user_array_of(array_cookie* cookie_address)
  {
    return cookie_address + 1;
  }

  extern "C" void* __aeabi_vec_ctor_nocookie_nodtor(
          void* user_array,
          void* (*constructor)(void*),
          size_t element_size, size_t  element_count)
  { // The meaning of this function is given by the following model implementation...
    // Note: AEABI mandates that __cxa_vec_ctor return its first argument
    return __cxa_vec_ctor(user_array, element_count, element_size, constructor, NULL);
  }
```

```
// __aeabi_vec_ctor_cookie_nodtor is like __aeabi_vec_ctor_nocookie_nodtor but sets
// cookie fields and returns user_array. The parameters are arranged to make STRD
// usable.  Does nothing and returns NULL if cookie is NULL.

extern "C" void* __aeabi_vec_ctor_cookie_nodtor(
        array_cookie* cookie,
        void*(*constructor)(void*),
        size_t element_size, size_t element_count)
{ // The meaning of this function is given by the following model implementation...
  if (cookie == NULL){ return NULL; }
  else
  {
    cookie->element_size = element_size;  cookie->element_count = element_count;
    return __aeabi_vec_ctor_nocookie_nodtor(
            user_array_of(cookie), constructor, element_count, element_size);
  }
}

extern "C" void* __aeabi_vec_cctor_nocookie_nodtor(
        void* user_array_dest,
        void* user_array_src,
        size_t element_size, size_t element_count,
        void* (*copy_constructor)(void*, void*))
{ // The meaning of this function is given by the following model implementation...
  // Note: AEABI mandates that __cxa_vec_cctor return its first argument
  return __cxa_vec_cctor(user_array_dest, user_array_src,
          element_count, element_size, copy_constructor, NULL);
}

extern "C" void* __aeabi_vec_new_cookie_noctor(size_t element_size, size_t element_count)
{ // The meaning of this function is given by the following model implementation...
  array_cookie* cookie =
      reinterpret_cast<array_cookie*>
          (::operator new[](element_count * element_size + cookie_size));
  cookie->element_size = element_size; cookie->element_count = element_count;
  return user_array_of(cookie);
}

extern "C" void* __aeabi_vec_new_nocookie(
        size_t  element_size, size_t  element_count,
        void* (*constructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  return __cxa_vec_new(element_count, element_size, 0, constructor, NULL);
}

extern "C" void* __aeabi_vec_new_cookie_nodtor(
        size_t  element_size, size_t  element_count,
        void* (*constructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  return __cxa_vec_new(element_count, element_size, cookie_size, constructor, NULL);
}

extern "C" void* __aeabi_vec_new_cookie(
        size_t  element_size, size_t  element_count,
        void* (*constructor)(void*),
        void* (*destructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  return __cxa_vec_new(element_count, element_size, cookie_size, constructor, destructor);
}
```

```
// __aeabi_vec_dtor is like __cxa_vec_dtor but has its parameters reordered and returns
// a pointer to the cookie (assuming user_array has one).
// Unlike __cxa_vec_dtor, destructor must not be NULL.
// user_array must not be NULL.

extern "C" void* __aeabi_vec_dtor(
        void* user_array,
        void* (*destructor)(void*),
        size_t element_size, size_t element_count)
{ // The meaning of this function is given by the following model implementation...
  __cxa_vec_dtor(user_array, element_count, element_size, destructor);
  return &cookie_of(user_array);
}

// __aeabi_vec_dtor_cookie is only used on arrays that have cookies.
// __aeabi_vec_dtor is like __cxa_vec_dtor but returns a pointer to the cookie.
// That is, it takes a pointer to the user array, calls the given destructor on
// each element (from highest index down to zero) and returns a pointer to the cookie.
// Does nothing and returns NULL if cookie is NULL.
// Unlike __cxa_vec_dtor, destructor must not be NULL.
//  Exceptions are handled as in __cxa_vec_dtor.
// __aeabi_vec_dtor_cookie must not change the element count in the cookie.
// (But it may corrupt the element size if desired.)

extern "C" void* __aeabi_vec_dtor_cookie(void* user_array, void* (*destructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  // like:
  //   __cxa_vec_dtor(user_array, element_count_of(user_array),
  //                  element_size_of(user_array), destructor);
  return user_array == NULL ? NULL :
          __aeabi_vec_dtor(user_array, destructor,
                           element_size_of(user_array), element_count_of(user_array));
}

extern "C" void __aeabi_vec_delete(void* user_array, void* (*destructor)(void*))
{ // The meaning of this function is given by the following model implementation...
  // like:  __cxa_vec_delete(user_array, element_size_of(user_array),
  //                         cookie_size, destructor);
  ::operator delete[](__aeabi_vec_dtor_cookie(user_array, destructor));
}

extern "C" void __aeabi_vec_delete3(
        void* user_array, void* (*destructor)(void*), void (*dealloc)(void*, size_t))
{ // The meaning of this function is given by the following model implementation...
  // like:  __cxa_vec_delete3(user_array, element_size_of(user_array),
  //                          cookie_size, destructor, decalloc);
  if (user_array != NULL) {
      size_t size =
          element_size_of(user_array) * element_count_of(user_array) + cookie_size;
    (*dealloc)(__aeabi_vec_dtor_cookie(user_array, destructor), size);
  }
}
```

```
  extern "C" void __aeabi_vec_delete3_nodtor(
          void* user_array, void (*dealloc)(void*, size_t))
  { // The meaning of this function is given by the following model implementation...
    // like:  __cxa_vec_delete3(user_array, element_size_of(user_array),
    //                          cookie_size, 0, decalloc);
    if (user_array != NULL) {
        size_t size =
            element_size_of(user_array) * element_count_of(user_array) + cookie_size;
      (*dealloc)(&cookie_of(user_array), size);
    }
  }

  extern "C" int  __aeabi_atexit(void* object, void (*destroyer)(void*), void* dso_handle)
  { // The meaning of this function is given by the following model implementation...
    return __cxa_atexit(destroyer, object, dso_handle);
  }

} // namespace __aeabiv1
```

3.2.2.4 Code examples for the delete expression

Section 5.3.5 of the ISO C++ standard discusses the delete expression.

The code needed to implement **`delete [] p`** is tabulated in Table 1, below. It depends on:

- The static element type of p (referred to as T below),
- Which `operator delete []` is being used for this deallocation: either `::operator delete(void*)` or `T1::operator delete(void*)`, where T1 is T or a base class of T.
- Whether a cookie is needed for arrays of T (see §3.2.2.1).
- *Has dtor*, which means T is a class type with a non-trivial destructor [ISO C++ §12.4]. In cases where there is no cookie there must be no dtor.

Table 1, Implementation of delete [] p

`operator delete []`	**Needs cookie**	***Has dtor***	**Implementation of** `delete [] p` / `::delete [] p`	**Note**
::operator delete[](void*)	N	–	::operator delete[](p)	
::operator delete[](void*)	Y	N	::operator delete[](&cookie_of(p))	2
		Y	__aeabi_vec_delete(p, &T::~T{D1})	
T1::operator delete[] (void*)	Y	N	T1::operator delete[](&cookie_of(p))	
		Y	T1::operator delete[] (__aeabi_vec_dtor_cookie(p, &T::~T{D1}))	
T1::operator delete[] (void*, std::size_t)	Y	N	__aeabi_vec_delete3_nodtor(p, &T1::operator delete[])	3
		Y	__aeabi_vec_delete3(p, &T::~T{D1}, &T1::operator delete[])	4

Notes

1. Other operator delete[]s, e.g. operator delete[](void*, const std::nothrow&) or operator delete[](void*, void*) can only be called from new array expressions with exceptions.

2. This is an unusual case that can only be reached by using `::delete[]`, for example:

```
struct T { static void operator delete(void*, std::size_t); } *p;
::delete[] p;
```

3. `__aeabi_vec_delete3_nodtor(p, &T1::operator delete[])` could also be done this way:

```
T1::operator delete[](&cookie_of(p), sizeof(T)*element_count_of(p))
```

4. `__aeabi_vec_delete3(p, &T::~T{D1}, &T1::operator delete[])` could also be done this way:

```
T1::operator delete[]
    (__aeabi_vec_dtor_cookie(p, &T::~T{D1}), sizeof(T)*element_count_of(p))
```

3.2.2.5 Code example for __aeabi_atexit

Because constructors conforming to this ABI return *this*, static construction can be done as:

```
__aeabi_atexit(T::T{C1}(&t), &T::~T{D1}, &__dso_handle);
```

This saves an instruction compared with:

```
T::T{C1}(&t);   __cxa_atexit(&T::~T{D1}, &t, &__dso_handle);
```

3.2.3 Guard variables and the one-time construction API

3.2.3.1 Guard variables

To support the potential use of initialization guard variables as semaphores that are the target of ARM SWP and LDREX/STREX synchronizing instructions we define a static initialization guard variable to be a 4-byte aligned, 4-byte word with the following inline access protocol.

```
#define INITIALIZED 1

// inline guard test…
if ((obj_guard & INITIALIZED)!= INITIALIZED) {
    // TST obj_guard, #1; BNE already_initialized
    if (__cxa_guard_acquire(&obj_guard)) {
        ...
}
```

A guard variable should be allocated in the same data section as the object whose construction it guards.

3.2.3.2 One-time construction API

```
extern "C" int __cxa_guard_acquire(int *guard_object);
```

If the guarded object has not yet been initialized, this function returns 1. Otherwise it returns 0.

If it returns 1, a semaphore might have been claimed and associated with *guard_object*, and either __cxa_guard_release or __cxa_guard_abort must be called with the same argument to release the semaphore.

```
extern "C" void __cxa_guard_release(int *guard_object);
```

This function is called on completing the initialization of the guarded object. It sets the least significant bit of *guard_object* (allowing subsequent inline checks to succeed) and releases any semaphore associated with it.

```
extern "C" void __cxa_guard_abort(int *guard_object);
```

This function is called if any part of the initialization of the guarded object terminates by throwing an exception. It releases any semaphore associated with *guard_object*.

3.2.4 Static object construction and destruction

Top-level static object object construction

The compiler is responsible for sequencing the construction of top-level static objects defined in a translation unit in accordance with the requirements of the C++ standard. The run-time environment (helper-function library) sequences the initialization of one translation unit after another. The global *constructor vector* provides the interface between these agents as follows.

- Each translation unit provides a fragment of the constructor vector in a read-only ELF section called .init_array of type SHT_INIT_ARRAY (=0xE).
- Each element of the vector contains the address of a function of type extern "C" void (*)(void) that, when called, performs part or all of the global object construction for the translation unit.

  The appropriate entry for an element referring to, say, __sti_*file* that constructs the global static objects in *file*cpp, is 0 relocated by R_ARM_RELABS32(__sti_*file*). Usually, R_ARM_RELABS32 is interpreted by a static linker as R_ARM_ABS32 (for details, see the Note below).
- Run-time support code iterates through the global constructor vector in increasing address order calling each identified initialization function in order. This ABI does not specify a way to control the order in which translation units are initialized.

Note In some execution environments, constructor vector entries contain self-relative references, which cost an additional ADD in the library code that traverses the vector, but save dynamic relocations, giving a smaller executable size and faster start-up when an executable must be dynamically linked and relocated. In these environments, a static linker interprets R_ARM_RELABS32 as R_ARM_REL32 rather than as R_ARM_ABS32.

Static object destruction

The sequencing of static object destruction in C++ is a complicated business that, in general, requires destructors to be registered dynamically in the order of object construction. This ABI requires static object destructors to be registered by calling __cxa_atexit or __aeabi_atexit (§3.2.2.5).

Usually, elements of the list of static objects to be destroyed will be allocated dynamically by __cxa_atexit.

Some environments require static allocation of space for the list.

To support static allocation, compilers must ensure that:

- Destructions are registered using __aeabi_atexit, not __cxa_atexit.
- Each call to __aeabi_atexit registers the destruction of a unique data object.
  (Thus each static call will be executed at most once, and table-driven registration of several destructions by a single static call to __aeabi_atexit is forbidden).

The maximum number of destructions that can be registered by a relocatable file is then the number of sites calling __cxa_atexit. A smart linker can count the number of sites and allocate space accordingly.

It is entirely quality of implementation whether a static linker and its matching run-time library can, in fact, allocate the required space statically.