Porting the Real-Time Operating System RODOS to SPARC V9

RODOS Porting Guide

Andre Bartke

Würzburg, August 2012

This document briefly describes the interfaces that must be implemented when porting RODOS to a new hardware platform. The related code should be placed in a directory that follows this pattern: `src/bare-metal/<platform>/`. All listed examples in this document belong to the `sparc64` target. Every `<platform>` has to provide, among others, some specific header files that will be required at compile time. Other interfaces require symbols for linkage and some are completely optional or depend on the respective target architecture. In the following sections, optional interfaces will be mentioned as *stubs*, meaning, their symbol must be present for the linker but they are not doing anything.

This section focuses on operating system specifics but we also need to support a few mandatory C++ run-time features and comply with some ABI standards. Besides this section it might be valuable to take a look at some language-related implementation details outlined in section 2 on p. 8.

1.1 Development Environment

Usually a new hardware platform requires a GNU cross compiler. This is necessary since development and boot image creation is not done from the same architecture we are targeting. For the development of this port a cross toolchain consisting of binutils and gcc is sufficient and must be configured to emit code for the target. This is done by building them with the `--target=sparc64-unknown-elf`[i] target triplet. On a common setup the build and host machine (on which the cross compiler is run from) is x86-based. Since C++ support is required, in addition to gcc-core, we also need the gcc-g++ package. To enable this compiler subset it is sufficient to set the configure switch `--enable-languages=c,c++`. Available with the sparc64 port, there is a bundled bash script, capable of building the necessary cross toolchain for a specified target triplet. By changing this triplet it would be easily adaptable to another platform.

[i]Another possibility would be the triplet `sparc64-unknown-linux-gnu`.

As for the linker, instead of specifying a long list of parameters we create a script that defines the relevant settings. A so-called *linker script* is parsed every time we link application software against the RODOS library. It includes the mapping of the output sections, additional symbols, as well as settings to strip off unnecessary symbols from the resulting binary.

1.2 System Configuration

There is a number of RODOS-related configuration parameters that define hardware limits and behavior of the operating system. The name of this header is expected to be `params.h` and must contain a list of preprocessor directives. An exemplary list of some important settings and its values (as set in the sparc64 port), is provided in figure 1. For a complete list of directives you should check with one of the existing ports. In general, most settings are only set preliminarily and may depend completely on the actual applications. Stack sizes for instance have to be evaluated for use cases individually. They not only depend on the applications that will eventually run on the system but also depend on factors like compiler optimization levels and the function call depths.

Figure 1: RODOS Configuration Example (params.h)

```
/** memory for allocations, used for all stacks */
#define XMALLOC_SIZE                    0x200000 /* 2M */

/** default stack size (in bytes) for threads */
#define DEFAULT_STACKSIZE               0x4000 /* 16K */

/** default stack size (in bytes) for the scheduler */
#define SCHEDULER_STACKSIZE             0 /* we continue to use the boot stack */

/** time interval between timer interrupts in microseconds */
#define PARAM_TIMER_INTERVAL            1000

/** time slice to switch between threads with priority conflicts */
#define TIME_SLICE_FOR_SAME_PRIORITY   (100*MILLISECONDS)
...
```

The setting for `XMALLOC_SIZE` defines how much memory is going to be available *all together*. One has to understand that the whole amount of memory RODOS will consume is static and known at compile time. The reserved amount is used by the `xmalloc` function that also allocates every thread's stack defined by `DEFAULT_STACKSIZE`. When setting these values, be aware of your system's resource limits. Given the values above, there would be place for 128 threads each with a 16K stack at most. While at it, it should be mentioned that the number of threads is also limited by a fixed amount of possible termination functions, determined by `MAX_ATEXIT_ENTRIES` which is set as part of the C++ ABI compliance (see section 2.3 on page 10).

The scheduler gets its own stack of the size defined by `SCHEDULER_STACKSIZE`. On the sparc64 port we continue to use our initial boot stack and therefore can set this value to zero. Interestingly we can start with a fresh stack every time the scheduler is called since none of its register contents needs preserving. This directly implies that on every call to `__asmSwitchToContext()` the scheduler's stack *dies* which saves us the effort of saving supervisor contents.

Usually the setting `PARAM_TIMER_INTERVAL` defines the interval between two consecutive timer interrupts. As for real-time operating systems, the timer Interrupt Service Routine (ISR) is also the place from which the scheduler is called. This is later explained as *pre-emptive context switching*. In case of a priority conflict, scheduling should be done with the *round-robin* algorithm. For this, the conflicting threads are assigned equal *time slices* that can be defined with `TIME_SLICE_FOR_SAME_PRIORITY`. When executing such threads, the variable `timeToTryAgainToSchedule` is set to the *current time*, plus this *time slice* definition. The ISR is then implemented in a way that it calls the scheduler only if the time at that moment has passed `timeToTryAgainToSchedule`. For the time these threads are active, the scheduler is called less frequently and only in the interval specified by `TIME_SLICE_FOR_SAME_PRIORITY`.

In the most recent RODOS version there have been some interfaces added that acquire information about the host's environment at run-time (see figure 2). These interfaces are very self-explanatory in general. In our case `getHostBasisOS()` returns `"baremetal"` and `getHostCpuArch()` will be `"sparc64"`. SPARC V9 is bi-endian, but usually operates in big-endian mode. Consequently `getIsHostBigEndian()` will return `true`. The one function worth talking about is `getSpeedKiloLoopsPerSecond()`. It should be set to a fixed value according to benchmark results from a development-test that can be found among the tutorials. Since all testing systems greatly differ for sparc64 and there is no mass produced SPARC V9 embedded processor at the moment, we cannot return a reliable value here. To make it clear that this value is currently useless we return `-1` since a negative value represents an impossible benchmark result.

Figure 2: RODOS Host Information Interfaces

```
bool getIsHostBigEndian();
long getSpeedKiloLoopsPerSecond();
const char* getHostCpuArch();
const char* getHostBasisOS();
```

1.3 Startup and Control Routines

Before we can jump to the kernel's main function, static locals and global objects have to be initialized. For every thread `hwInitContext()` is called to set up the context with certain parameters. The first argument to this function is a pointer to the start of the thread's stack and provides the basis to compute an appropriate stack pointer. Stacks are generally very architecture specific and may require special alignments. SPARC requires an alignment of 16-bytes and the top of the stack has to be biased by an offset of -2047. The second argument is the thread's object pointer (`this`) that is passed to the *callback function*.[ii] It is stored in the register that makes up the first argument on a function call. This initial entry point is the callback function `threadStartupWrapper()`. Its address has to be stored within a context structure so it will be loaded to the program counter when switching to the thread. The startup wrapper begins the thread execution by jumping to its `run()` routine.[iii] Upon exit, `hwInitContext()` returns a pointer to the newly created context. This is usually the address of a *C struct* containing space for a number of registers[iv] which is located within the initially allocated stack frame.

When this is done, we can safely enter `main()` that can be found in `src/independent/-main.cpp`. It immediately calls `hwInit()` where hardware-related initializations like FPGAs or devices of the new platform should be located. If we already did everything necessary, this function can be a stub. Like the name suggests, the `hwResetAndReboot()` routine ends the program and initiates a reboot. As it will be described in the C++ section concerning ABI compliance (2.3 on page 10), this would also be the time to call `__cxa_finalize(0)` in order to call all registered termination functions in the reverse order of their registration. The `sp_partition_yield()` function is always a stub when dealing with `bare-metal` targets. It is only used on top of other operating systems such as Linux to allow additional space/time partitions in terms of processes. Even less likely is an implementation for the `startIdleThread()` function. It is only used on processors that support special exceptions for context switching like the ARM Cortex M3 does, otherwise it is also a stub. An overview of the just discussed functions is listed in the figure 3 below.

Figure 3: RODOS Startup and Control Interfaces

```
void hwInit(); /* may be a stub */
long* hwInitContext(long* stack, void* object);
void hwResetAndReboot();
void sp_partition_yield(); /* stub on bare-metal */
void startIdleThread();  /* stub on most bare-metal targets */
```

These interfaces are usually placed in a file named `hw_specific.cpp` located in the platform's root directory. The corresponding header file `hw_specific.h` can be found in `src/bare-metal-generic/`.

1.4 Timing Interfaces

One of the most fragile parts of a real-time operating system is precise time keeping. The whole concept shatters if an error in this matter regresses over the time of execution and time windows start to be missed. A timer will most certainly be implemented using an interrupt handler

[ii] Explained in section 1.5 on p. 5.
[iii] `Thread::run()` is a pure virtual that is implemented by inherited classes.
[iv] The registers stored in this structure are completely dependent on the architecture.

that sums up the passed nanoseconds since the last interrupt to a global value. The initialization of time and registration of a respective interrupt handler can be done in either one of `hwInitTime()` or `Timer::init()`. The other one may be left as a stub. The `Timer` class also provides `start()` and `stop()` methods that are intended to disable interrupts over the time of a voluntary context switch. If this is handled otherwise it is perfectly fine to also leave these methods as stubs. The timer interval, that is the time between two consecutive timer interrupts, should be settable with the method `setInterval()` in a scale of microseconds. It is called with the argument `PARAM_TIMER_INTERVAL` from the RODOS main function right before `init()`. The latter should start the timer interrupts. Given the nature of the sparc64 timer support, we convert the interval time to an equal number of *ticks* (clock-cycles) at run-time. This is dependent on the respective clock frequency that we can easily acquire through the OpenBoot PROM.[v] As viewed in figure 4, there are two different interfaces to acquire the current time.

Figure 4: RODOS Timing Interfaces

```
void hwInitTime();  /* may be a stub */
long long unsigned int hwGetNanoseconds();
long long unsigned int hwGetAbsoluteNanoseconds();
void Timer::init();
void Timer::start();
void Timer::stop();
void Timer::setInterval(const long long interval);
```

Based on the number of counted *ticks* the function `hwGetNanoseconds()` returns the number of passed nanoseconds since the system was booted up. The resolution is dependent on the frequency at which the timer operates but is scaled to nanoseconds because this is the timing unit RODOS relies on. While time events are mostly based on the value returned by this function, a pseudo random number generator greatly benefits from increased entropy and relies on `hwGetAbsoluteNanoseconds()` which adds the value of an internal real-time clock if present. Otherwise, it just returns the same value as `hwGetNanoseconds()`.

Even if irrelevant for the current sparc64 implementation, it is an important aspect that on other hardware platforms there are certain additional issues to consider. On some platforms the `start()` and `stop()` methods are used to disable timer interrupts over the period of a voluntary context switch. This may lead to a violation of correct timings. If not, otherwise possible correct time keeping must be ensured with an additional timer. For an example of a secondary timer solution you should take a look at the STM32 port.

1.5 Context Switching

In a multitasking environment there has to be a facility to switch the execution context which, on RODOS, implies the switch to another thread. Responsible routines are most certainly implemented in assembly since we need to directly access and modify register contents. But before

[v]The SPARC firmware interface, also known as IEEE 1275 OpenFirmware.

we can do a switch, we have to pass control to a thread in the first place. This is done by the scheduler with a call to the `__asmSwitchToContext()` routine. Here, supervisor mode is left in favor of the user mode, saved registers are restored and finally a jump to the old program counter is made. As mentioned earlier, when the thread is entered the first time it initially jumps to the `threadStartupWrapper()`, that calls the implemented virtual `Thread::run()`.

It should be explained at this point why we use top-level wrapper functions as intermediaries in order to interact with C++ members. As context switching is most likely implemented in assembly, it is not possible to use a non-static C++ member function. Those are meaningless without an instance to call them on. Therefore a callback function should be `extern "C"`[vi] to get correct calling conventions through C linkage. This is also the reason why `hwInitContext()` is passed a `'void *'` object pointer that in turn is passed to the callback function. It provides the `Thread` object instance that is necessary to call the member method we are actually aiming for.[1, p 67]

Context switches can be voluntary, initiated by a call to `Thread::yield()` from the currently executed thread. This method invokes a direct call to a routine usually implemented in assembly called `__asmSaveContextAndCallScheduler()`. In order to pass control to the scheduler one usually has to switch back to privileged mode which is why this routine issues a system call on sparc64. In case of a voluntary context switch we arrive in the appropriate routines similarly to a normal function call. Therefore only those registers need to be saved that needs preserving across function calls. After everything is stored to a respective context structure, we place the context pointer from the thread we are switching from as the first argument and jump to the `schedulerWrapper()`.

These two routines, summarized in figure 5, are absolutely essential and have to be implemented for every new hardware platform. The routines might be invoked very frequently which makes it a good idea to optimize them for a minimal number of instructions.

Figure 5: RODOS Context Switching Interfaces

```
    void __asmSwitchToContext(long* context);
    void __asmSaveContextAndCallScheduler();
```

Involuntary context switching, known as pre-emption, describes a process where control is withdrawn from a thread without further notice. This is a core concept of *real-time*, that it is not possible for a thread to occupy the system *indefinitely*. After a previously defined interval, the context is forcibly switched. The procedure is highly architecture dependent as the complete register file, including the current processor state, needs to be preserved. Even if unimportant for the world outside of the bare-metal target, for consistency reasons the internal entry point for this routine is often named `__asmSaveContext`. Upon exit, the function will also jump to the `schedulerWrapper()` in supervisor mode. It is important that we only want to call the

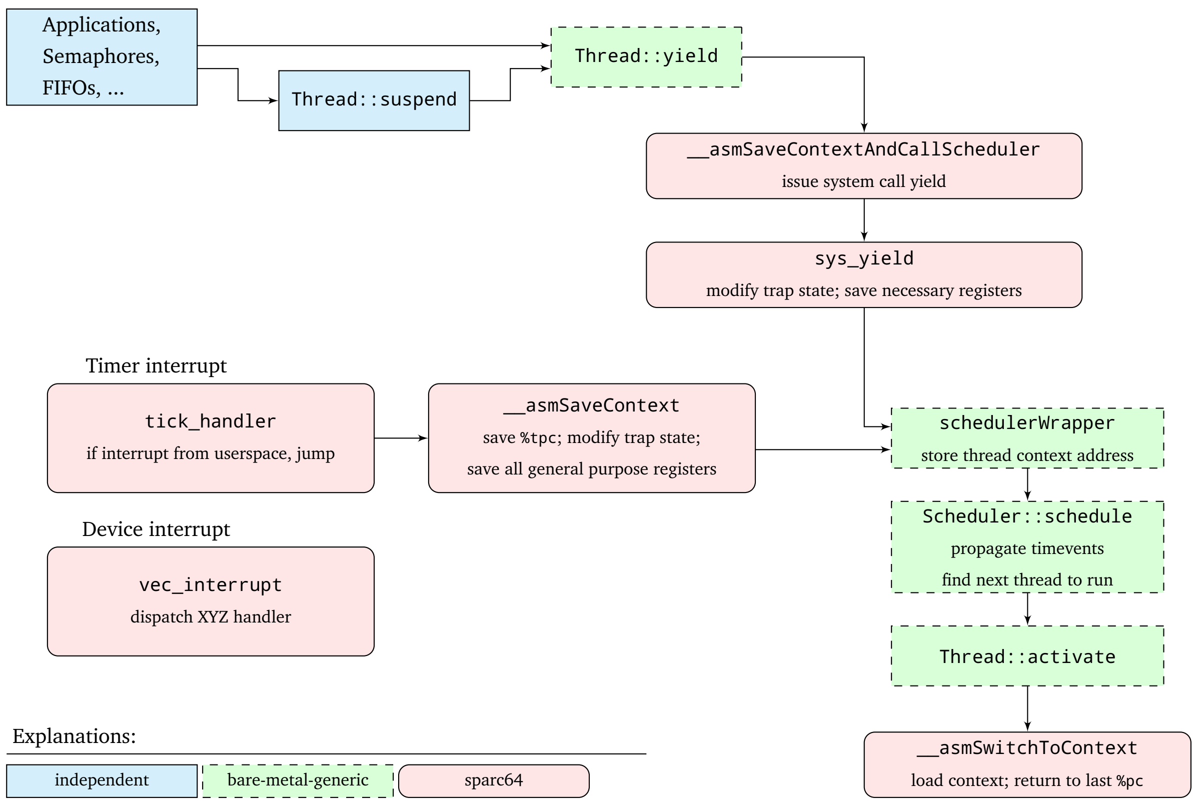
[vi] `static` member methods do not need an object to be called on and might be equally possible.

[1] Dewhurst, S. C., *C++ Common Knowledge: Essential Intermediate Programming*, 1st ed.; Addison-Wesley: July 2005.

scheduler from the timer ISR when the variable `isSchedulingEnabled` is set and the current time is greater than `timeToTryAgainToSchedule`.

The possibilities for a task switch are visualized in figure 6. In the sparc64 implementation the two functions `__asmSaveContext` and `sys_yield` are structured very similarly since both are trap handlers that return to the `schedulerWrapper()` in supervisor mode. In principle, the main difference to the previous voluntary switch is the amount of saved registers. It is noteworthy that this might be done with a single routine on a different platform.

Figure 6: RODOS Visualized Taskswitch on sparc64



1.6 Low Level I/O

However minimal necessary interfaces are laid out, we need at least some possibility to provide feedback (even if only readable with a serial connector). The high level function `xprintf` from the RODOS API for example, translates to `putchar()` on a lower level. In turn, `putchar()` may translate to the `write()` system call which is the way it is realized on the sparc64 port. Optionally, `puts()` can be provided in a similar fashion, the prototypes are shown in figure 7 below. The header `hw_stdlib.h` from the bare-metal target will be included in such cases and must specify these symbols. We also need to provide the facility for *variadic functions* here, meaning, those that allow a variable number of arguments.[2, p 154] In a freestanding environment

[2]Stroustrup, B., *C++ Programming Language*, 3rd ed.; Addison-Wesley: June 1997.

these are provided by the `stdarg.h` header file which is provided by the cross compiler.

Figure 7: RODOS Console Interfaces (hw_stdlib.h)

```
    int putchar(int character);
    int puts(const char *str);
```

I/O device drivers are expected to follow the naming convention `hw_<DEVICE>.cpp` and to be placed in the bare-metal's root directory. However, in any case there should be `hw_uart.cpp` that specifies at least a stub for `FFLUSH()` as well as `hw_udp.h` where the classes `UDPReceiver` and `UDPTransmitter` are required at least as a dummy interfaces.

For a completely independent C-library like Red Hat's *newlib*[vii] there are a number of system calls that need to be implemented. For newlib, a list (including possible *stubs*) can be found online.[viii] In order to be able to link against this library, newlib would have to be built for the respective target. Note that this is completely optional and not an integral part of any RODOS port.

2 C++ Runtime Support

In contrast to C, there are a few more things that have to be taken into account when writing an operating system in C++. There are a lot of language features that require no special attention like classes or virtual functions. Other features like exceptions can be disabled by compiler flags if not needed. Some parts of the language however are mandatory and assume a special run-time support to be present, as we will see in the following sections. For full C++ run-time support, the libraries `libgcc` and `libsupc++` would be necessary on the target platform. We will however only provide a small, essential subset.

2.1 Freestanding Environment

In a hosted environment `g++` is linking against `crt1.o`[ix] short for *C run-time*. Basically it contains the startup and shutdown code that has to run before and after the `main()` function is called.[3, pp 500,501] This code also includes the symbol `_start` that becomes the entry point of the executable. When a kernel then loads this binary, the `.text`, `.data` and `.bss` sections are set up and arguments are pushed to the stack. The entry point `_start` is called and a C-library takes care about the environment before calling the actual `main()` function. Since we operate in kernel land, ergo a *freestanding environment*, nothing of this will magically happen in the background, but we have to take care of it ourselves. The first thing to do is to disable this default behavior with the `-nostartfiles` compiler flag.

[vii] See http://sourceware.org/newlib/ visited on July 11, 2012.
[viii] See http://sourceware.org/newlib/libc.html#Syscalls visited on July 11, 2012.
[ix] It may also be `crt0.o` if the system does not support constructors and destructors.
[3] Prinz, P.; Crawford, T., *C in a Nutshell*, 1st ed.; O'Reilly Media, Inc.: Dec. 2005.

There is another C++ feature called Run-Time Type Information (RTTI) which adds an additional variable with the object's type to the vtable.[x] The idea is to have information about the type at run-time as opposed to at compile time to simplify resource management. This allows to call `typeid()` on an object to determine the corresponding class at run-time. Another example would be `dynamic_cast<>`. It is used in situations where we are unsure about the type of a pointer. It can *dynamically* convert a base-class pointer to a derived-class pointer if we want to call a derived-class member. In our kernel we have no need for any of this so it is another feature we want to disable. The compiler flag we are using is `-fno-rtti`.

We also do not intend to provide run-time support for exception handling. The compiler flag to disable exceptions is `-fno-exceptions`.

2.2 Object Construction

Since we have already established the fact that there is no C run-time that handles the function *prologue* for us, we also have to consider that local static and global objects are not getting instantiated on their own. For every global object in our kernel we have to call its constructor before we can use respective member functions. GCC places a list of these pointers in a section called `.ctors` at link time. We have to collect this information at run-time and make calls to these constructor function pointers. The gnu-ld manuals[4] specify a sequence for the linker script in order to emit this list in a certain structure. The adapted version is shown in figure 8 and is placed in the `.rodata` (read-only data) section.

Figure 8: Linker script, C++ Static & Global Constructors

```
    __CTOR_LIST__ = .;
    QUAD((__CTOR_END__ - __CTOR_LIST__) / 8 - 2)
    *(.ctors)
    QUAD(0)
    __CTOR_END__ = .;
```

We are operating in a 64-bit environment where pointers are 8-bytes in size. The examples from the manuals are intended for a 32-bit environment. We had to replace all 4-byte `LONG()` words with 8-byte `QUAD()` (extended) words. The same applies to the division by eight. The first element of the list contains the number of constructors, then follows a list of the function pointers and the last item is a zero. In order to count the number of constructors between the `__CTOR_LIST__` and `__CTOR_END__` symbols, we have to divide the number of bytes in between by eight and subtract two for the first and last item of the list.

If the order in which these constructors are executed is of importance, one could use the GNU C++ initialization priorities[5] and put `*(SORT_BY_NAME(.ctors))` instead of `*(.ctors)` in the

[x]Table of addresses that allows dynamic dispatching to implementations of virtual functions.
[4]Documentation for binutils 2.22, Output Section Keywords.,
URL: http://sourceware.org/binutils/docs/ld/Output-Section-Keywords.html accessed June 7, 2012.
[5]GCC online documentation, Extensions to the C++ Language.,
URL: http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html accessed July 17, 2012.

linker script.[4]

To actually call the constructors we need to acquire the stored information from this structure. The number of entries can be found through the __CTOR_LIST__ symbol, which when incremented, has the address of the function pointer of the first constructor. We then consecutively call constructor functions until the just acquired number is met. An example is shown in figure 9 below, where __CTOR_LIST__ is defined as an external function pointer.

Figure 9: Call Constructors for Local Static and Global Objects

```
void (**c)(void) = &__CTOR_LIST__;     // get a pointer to the list
uint64_t nentries = *(uint64_t *)c++; // first entry is the number of ctors
while(nentries--)
  (*c++)();                            // call constructors
```

2.3 Application Binary Interface

An ABI defines all the low-level rules that are expected of executables to follow. Parameter passing, calling conventions and alignments may be some of the things that are specified by it. This becomes particularly important when it comes to dynamic linking or compilers itself. In general, the software follows the System V ABI as described in the compliance definition.[6] This describes low-level interfaces to ensure binary compatibility between the operating system and applications. Since GCC 3.0, g++ follows the Itanium C++ ABI[xi] that also has to be taken into account as a language standard.

One could assume the previously explained process for constructors similarly applies to destructors in a .dtors section, but this is no longer true. The ABI dictates that every constructor has to register a destructor function with `__cxa_atexit()`. This behavior should be default, but can be enforced with the -fuse-cxa-atexit compiler option. The ABI further states that a list of termination functions has to be maintained which, upon exit, is processed in reverse order.[7] It shall contain the termination function pointer, an optional function argument and a dynamic shared object handle named `__dso_handle`. The *dso handle* is a unique value that identifies the shared library of the entry, or in our case, the kernel. Upon termination of a program, `__cxa_finalize()` is called with the address of a function as the parameter. It then calls the corresponding destructor and removes the object from the list. The ABI specifies that when the parameter is 0, all termination functions in the list are called in the reverse order of their registration. This is the expected behavior when the kernel exits.

It is also required to define a function `__cxa_pure_virtual` that terminates the program if a *pure virtual* function is called. A function is called a 'pure virtual' if its vtable address is a NULL

[6] INTERNATIONAL, S., *SPARC Compliance Definition 2.4.1*; SPARC International: July 1999.
[xi] Despite what the name suggests, it is compatible with many architectures.
[7] Itanium C++ ABI, (Revision: 1.83)., Ref: #dso-dtor, #once-ctor,
URL: http://refspecs.linuxbase.org/cxxabi-1.83.html accessed June 8, 2012.

pointer. This should never happen since it is impossible to instantiate a class without defining all its virtual functions. A summary of the discussed interfaces is shown below in figure 10.

Figure 10: C++ ABI Functions

```
void *__dso_handle; /* dynamic shared object identifier */
int __cxa_atexit(void (*func)(void *), void *arg, void *dso_handle);
void __cxa_finalize(void *func);
/* in case a pure virtual is called */
void __cxa_pure_virtual();
```

The ABI further defines run-time support concerning local static variables.[7] As discussed, global scope static objects have their own constructors that are part of the start-up code. For static locals the compiler acquires a *guard* to ensure the initialization code is executed by one thread only. This is necessary due to a concept called *thread-safety*. To support thread-safe initialization, we have to implement the three API functions shown in figure 11. Alternatively, this behavior can be disabled by the compiler option `-fno-threadsafe-statics`, but this would violate the ABI.

Figure 11: Thread-safe Initialization of Local Statics

```
int __cxa_guard_acquire(__int64_t *g);
void __cxa_guard_release(__int64_t *g);
void __cxa_guard_abort(__int64_t *g);
```

Before entering the initialization code, g++ adds in the `__cxa_guard_acquire()` function. If the object in question is not instantiated its status is set from *not initialized* to *pending*[xii] and the function returns 1. After the initialization code has run, a call to `__cxa_guard_release` is issued which sets the state to *done*. If a second thread also wants to initialize this resource, even at *nearly* the same time, it can only encounter an initialization *pending* or - *done* state, upon which it returns 0. In case of a *pending* initialization the thread is waiting for it to finish. This concept guarantees safe execution in an environment with multiple threads since access to shared resources is limited to one thread at a time.

2.4 Vague Linkage

Normally, when linking, sections with the same name are combined in a single section. We are also used to the fact that the linker will complain if there are multiple definitions of the same symbol. The concept of vague linkage allows duplicate occurrences of such constructs across multiple object files by defining these sections with the *link once* type.[8] This is necessary since features like templates may instantiate the same symbol more than once. In order to allow multiple definitions, the symbols are defined as `weak`. For any template expansion, these are then placed in individual `.gnu.linkonce.*` sections. At link time, if there are sections with the same name

[xii]Optimally with an atomic compare and swap instruction.
[8]GCC online documentation, Vague Linkage., Free Software Foundation, Inc.,
URL: http://gcc.gnu.org/onlinedocs/gcc/Vague-Linkage.html accessed June 8, 2012.

that are starting with `.gnu.linkonce.*`, all but one of the definitions are discarded.[9, pp 99–102] For this to work, we need to specify special sections in the linker script as shown in table 1. A similar process can be found with virtual functions. The compiler will define `weak` symbols for all objects that include the class header file, but later on, only generates a vtable in those that define these functions.

Table 1: GNU Linker, *link once* sections

`.text`	`*(.gnu.linkonce.t*)`
`.rodata`	`*(.gnu.linkonce.r*)`
`.data`	`*(.gnu.linkonce.d*)`
`.bss`	`*(.gnu.linkonce.b*)`

Generally speaking, it is a concept to remove duplicate code that results from various C++ features. It is called *vague* because there is no standard of doing this sort of thing. The *linkonce* implementation of the GNU linker is just a variation of this concept. Microsoft introduced a similar implementation where the object's section is called 'COMDAT'.[9, p 101]

3 References

[1] Dewhurst, S. C., *C++ Common Knowledge: Essential Intermediate Programming*, 1st ed.; Addison-Wesley: July 2005 (cit. on p. 6).

[2] Stroustrup, B., *C++ Programming Language*, 3rd ed.; Addison-Wesley: June 1997 (cit. on p. 7).

[3] Prinz, P.; Crawford, T., *C in a Nutshell*, 1st ed.; O'Reilly Media, Inc.: Dec. 2005 (cit. on p. 8).

[4] Documentation for binutils 2.22, Output Section Keywords., URL: http://sourceware.org/binutils/docs/ld/Output-Section-Keywords.html accessed June 7, 2012. (Cit. on pp. 9, 10).

[5] GCC online documentation, Extensions to the C++ Language., URL: http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html accessed July 17, 2012. (Cit. on p. 9).

[6] International, S., *SPARC Compliance Definition 2.4.1*; SPARC International: July 1999 (cit. on p. 10).

[7] Itanium C++ ABI, (Revision: 1.83)., Ref: #dso-dtor, #once-ctor, URL: http://refspecs.linuxbase.org/cxxabi-1.83.html accessed June 8, 2012. (Cit. on pp. 10, 11).

[8] GCC online documentation, Vague Linkage., Free Software Foundation, Inc., URL: http://gcc.gnu.org/onlinedocs/gcc/Vague-Linkage.html accessed June 8, 2012. (Cit. on p. 11).

[9] Levine, J. R., *Linkers and Loaders*, 1st ed.; Morgan Kaufmann: Oct. 1999 (cit. on p. 12).

[9] Levine, J. R., *Linkers and Loaders*, 1st ed.; Morgan Kaufmann: Oct. 1999.