Application Note 150

Building Linux Applications Using RVDS 3.0 and the GNU Tools and Libraries

Released on: September, 2006

ARM®

Application Note 150

Building Linux Applications Using RVDS 3.0 and the GNU Tools and Libraries

Release Information

The following changes have been made to this application note.

Table 1 Change history

Date	Issue	Change
April 2006	A	First release for RVDS 3.0
September 2006	B	Second release for RVDS 3.0 SP1

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

1 Introduction

1.1 Legal notices and support status

IMPORTANT LEGAL NOTE

ALL USE OF THE RVDS SOFTWARE IS GOVERNED BY THE TERMS OF THE RVDS END USER LICENSE AGREEMENT. NOTHING IN THIS DOCUMENT VARIES THIS LICENSE OR GRANTS YOU ANY ADDITIONAL RIGHTS. NOTHING IN THIS DOCUMENT CONSTITUTES LEGAL ADVICE. IF YOU HAVE ANY QUESTIONS PLEASE CONSULT YOUR LAWYER.

Example code end-user license agreement (EULA)

The end-user license agreement for the included examples can be found in the accompanying EULA.txt file. Please read this license before using the examples. You must agree to the terms of the license before using the example code.

Disclaimer

Please note that should you choose to use this example code in conjunction with your own or third party proprietary software and the GNU C library or any other open source code, you do so entirely at your own risk. ARM makes no representation or warranty as to the legal or business implications of such use. You should consult your own legal advisors if you have any concerns about this. Also, for the avoidance of any doubt, the example code is not open source software. Nothing in your license for this example code or your license (if any) for the ARM RealView Development Suite or, if applicable, RealView Compilation Tools or other ARM development tools products, allows you to distribute this example code, or the libraries or example code supplied by ARM with RVDS or RVCT, or any derivative or collective work created using such software, under a GNU Public License, or other open source license.

Support status

Please note that ARM does not provide support on the use of GNU tools or Linux. The information provided here is given for your reference only. We can provide limited support for the instructions in this document to customers with a valid RealView tools support contract with us. However, we suggest that in the first instance you discuss your issue in one of the various public forums, such as the comp.sys.arm newsgroup.

Alternatively you may prefer to contact CodeSourcery, who can provide paid support and assistance on the GNU tools or accept defect reports. Further information is available from CodeSourcery's GNU toolchain page at: `http://www.codesourcery.com/gnu_toolchains/arm/` CodeSourcery also provide a mailing list for queries on the ARM GNU toolchain. Details of how to subscribe to this list can be found on the CodeSourcery website.

1.2 Scope of this document

This application note gives an introduction to building a Linux application or library, linked with the GNU C and C++ libraries, using the compilation tools provided as part of the RealView Development Suite (RVDS) 3.0. These are referred to in the rest of this application note as RVCT.

RVCT is primarily aimed at building static images for a standalone embedded system, rather than dynamically-linked images such as Linux applications. However, with RVCT 3.0 it is possible to create dynamic images that can run under Linux using the GNU C library.

This application note covers the command-line options used to build a Linux-compatible executable, and describes how to use header files and libraries from the GNU C library (glibc).

Expected use

This application note is intended to cover most expected use cases. It is specifically aimed at developing Linux applications and libraries in the following situations:

- Building a standalone Linux application with RVCT
- Building static and shared libraries with RVCT, and linking these to an application built with RVCT
- Building a static or shared library with RVCT, and linking this to an application built with the GNU toolchain.

Example code

The following examples are supplied, with complete code and makefiles:

- A "hello world" example, which is suitable as a template for your own application builds
- The Dhrystone benchmark, as an example of a simple yet non-trivial application
- A simple example of a C++ application
- An example demonstrating how to build and use static and dynamic libraries with C and assembly code
- An example demonstrating how to build and use static and dynamic libraries with C++ using a combination of GNU and RVCT tools.

For further details of the examples, please see Chapter 4.

1.3 Limitations

There are several limitations on interoperation between the GNU tools and libraries and RVCT:

The GNU binutils (including ld) from CodeSourcery's 2005-q3-2 release cannot consume RVCT 3.0 objects

For linking with the GNU C library, you should use the 2005-Q3-2 release of the CodeSourcery tools (or a later release). However, due to updates in the ARM ABI ELF specification, the binary utilities (binutils) from this CodeSourcery release cannot consume object files built by RVCT 3.0. Support for the new ELF ABI revision is in the 2006-q1 and later releases.

RVCT cannot be used for compiling the Linux kernel or kernel-based code, such as device drivers or other kernel modules

This is because a significant portion of the kernel code is written in assembly language using the GNU assembler ("`gas`") syntax. This is incompatible with `armasm`, and there is no performance gain to be made from rebuilding such code with a different assembler.

In addition, the function interfaces for the kernel code prior to version 2.6.16 have not been written to comply with the ABI. This means that drivers and other kernel modules cannot be compiled using RVCT as there are no guarantees that calls would be made correctly between the kernel and the driver code. You must use the GNU toolchain and the old GNU ABI when building the kernel and kernel modules.

Only ARM architecture versions 5TE and above are supported

See "Target Requirements" below.

C++ exceptions are not supported

These are currently only compatible at the ABI-specified interface. The implementation details differ such that C++ exceptions from `armcc`-compiled code cannot be handled by the GNU C/C++ library, and likewise exceptions from classes in objects compiled with GNU g++ cannot be handled by the RVCT libraries.

You must take care when using `alloca()`

This is compiler-specific, therefore when calling the function from RVCT-compiled code you must statically link against the implementation in the RVCT libraries. This is contained in the `alloca.o` and `rt_alloca_state.o` C library objects. Note also that the RVCT library implementation of `alloca()` is not thread-safe and using `alloc()` with `setjmp()` and `longjmp()` may lead to memory leaks. You may prefer to write your own `alloc()` implementation around `malloc()`, however this would not be trivial.

GCC inline assembly code is not compatible with RVCT and vice versa

Likewise, standard assembly language files cannot be built by both `armasm` and the GNU assembler (`gas`) as they use different syntax. The recommended solution is to conditionally use alternative copies of your assembly code with the appropriate syntax for each toolchain.

Limitations on wide character variables (`wchar_t`)

RVCT 3.0 only supports `unsigned short` type wide character variables. However, RVCT 3.0 SP1 adds support for `unsigned int` type for wide character variables, through the use of the `--wchar32` compiler option.

1.4 Requirements

This document assumes that you are familiar with RVDS, the GNU tools and Linux.

Target requirements

Please note that the instructions in this document relate to building Linux applications for ARM architecture v5TE or later targets, such as the ARM926EJ-S or ARM1176JZ-S. This is because the ARM ABI uses architecture v5TE as its reference architecture and earlier architecture versions are not fully covered by the ABI.

You may be able to use these instructions to build Linux applications for ARM architecture v4T cores (such as the ARM720T and ARM920T) with RVCT. However, this is entirely at your own risk and is not supported. In particular, you will not be able to use Thumb code built for architecture v4T. We suggest that you only use the GNU toolchain when building Linux applications for ARM architecture v4T targets.

Your target's filesystem must contain the ABI-compliant library binaries. These are included in the CodeSourcery GNU toolchain releases described below in the build requirements. Finally, the target must be running a Linux kernel with support for NPTL (the Native POSIX Threading Library, the new mechanism for supporting multithreaded code under Linux with the GNU C library) and TLS (thread-local storage). See section 1.4 below for further details on the ABI requirements on your target system.

For the mainstream kernel source, this means that your target must be running version 2.6.12 (or later) of the Linux kernel. However, your Linux distribution may have applied the appropriate patches to its release of an earlier kernel. You should contact your Linux distributor for further details.

Prebuilt binary images of the Linux kernel configured for ARM's own development boards can be found on the ARM website at: `http://www.arm.com/linux/prebuilt_download.html`.

Build requirements

All information in this document relates to the use of RVCT 3.0 and RVCT 3.0 Service Pack 1 (SP1).

Note that CodeSourcery's 2005-Q1 release was the first to allow EABI-compliant interoperation between RVCT and the GNU toolchain. However, several enhancements and fixes have been made since then and the instructions in this document relate only to the CodeSourcery 2005-Q3-2 release and beyond, as it is now simpler and safer to link with the newer release.

Any work that you do using CodeSourcery's 2005-Q1 or older releases is at your own risk and we will not be able to provide any support when using RVCT with these old versions.

At the time of writing the CodeSourcery binary and source packages for the GNU toolchains can be found at: `http://www.codesourcery.com/gnu_toolchains/arm/`

Your ARM Linux distribution may already use the CodeSourcery toolchain or have the appropriate patches applied. You should contact your ARM Linux distributor for further details.

The examples have been tested using RVCT 3.0, RVCT 3.0 SP1 and CodeSourcery releases 2005-Q3-2, 2006-Q1-3 and 2006-Q1-6.The example makefiles have all been developed for and tested with GNU make 3.80 however they should also work correctly on other recent versions of GNU make.

1.5 About the ARM ABI (Application Binary Interface)

The Application Binary Interface (ABI) for the ARM Architecture is a collection of standards, some open and some specific to the ARM architecture. The standards regulate the inter-operation of binary code, development tools, and a spectrum of ARM core-based execution environments from bare metal to platform operating systems such as ARM Linux.

A third-party toolchain such as the GNU tools must comply with the standards given in the ABI for its objects to link and interoperate correctly with those produced by RVCT. The CodeSourcery release of the GNU tools is specifically tailored to fully support the ARM ABI and allow objects produced using both RVCT and the GNU tools to work together successfully.

Further details of the ARM ABI, including the full ABI documents, can be found on the ARM website at: `http://www.arm.com/products/DevTools/ABI.html`.

Interactions between mixed-ABI components

In the short term, it is likely that you will need to build a mixed ABI system. Kernels before version 2.6.16 can only be built using the legacy GNU ABI (use GCC option `-mabi= apcs-gnu` when using the CodeSourcery toolchain). This includes all kernel modules and device drivers.

This can cause problems when your applications or library must interface directly with kernel structures or functions (syscalls), including through the use of a shared header file describing kernel structures. In this case, you must use assembly code or modified descriptions of the structures to translate between the two ABIs when calling kernel functions or manipulating kernel data structures in your applications or libraries.

As of kernel 2.6.16, the Linux kernel can be built using the new ARM EABI. This allows for much simpler integration of applications and libraries to form a completely EABI-compliant system.

2 Command-line options

You must use several command-line options when compiling and linking your source files. These are explained in the following sections. Examples of the full command lines can be found in the example makefiles.

Please note that you should also be careful when using other compiler and linker switches. For example, you cannot use `--apcs /adsabi` as the ABI (application binary interface) used by ADS (the ARM Developer Suite) is incompatible with the more recent ABI used by RVCT.

2.1 Compiler language and code-generation options

There are a number of options required when compiling source files for Linux applications:

`--gnu`
: As the headers accompanying the GNU C libraries must be used when linking against glibc, you must pass this switch to the ARM compiler to enable support for the GNU extensions. See the section GNU extensions to the ARM compiler in the RVCT 3.0 Compiler and Libraries Guide for a full explanation of the GNU extensions supported by RVCT.

`--enum_is_int`
: This causes all enums to be treated as integer types. This is required by the CodeSourcery document "ARM GNU/Linux Application Binary Interface Supplement" for building Linux applications. For more information on this option, see the section Controlling implementation details of the RVCT 3.0 Compiler and Libraries Guide.

`--library_interface=aeabi_glibc`
: This disables certain optimizations that the ARM compiler uses by default when linking with the RVCT libraries.

`-Dlrintf=_ffix_r, -D__align=__alignx`
: These two options define the names used by GNU for two ARM built-in functions.

`--apcs /interwork`
: This instructs the compiler to build all code for interworking, and to set the build attributes in the output objects accordingly. This ensures that your code is built correctly for ARM/Thumb interworking. It also helps to prevent the linker generating additional warnings, as GCC does not add build attributes to an object file that indicates whether it was compiled for ARM/Thumb interworking or not.

`--no_hide_all`
: This instructs the compiler to use dynamic import and export for the symbols in an image. This sets the attributes of references and definitions in the image so that they can be dynamically linked.

`--wchar32`
: This changes type of wchar_t to unsigned int instead of unsigned short. Please note this is only available in RVCT 3.0 SP1 (build 586 and later).

Optional compiler switches

`--signed_bitfields`
: This option makes bitfields signed by default. If you are building code that has previously been targeted at GCC, you may need to use this switch for compatibility.

2.2 Assembler command-line options

When using assembly code in your application or library, two of the above compiler switches must also be given to the assembler:

`--apcs /interwork`

This instructs the assembler to set the build attributes in the object file to indicate that the code is ARM/Thumb interworking-safe.

`--no_hide_all`

This indicates that the assembler must use symbol import and export for all symbols.

2.3 Additional headers from RVCT

Some of the RVCT headers must be used in preference to those from the GNU C library. These headers define some implementation-specific macros. The files are provided in the include directory in the accompanying files, and this directory should be given before the GNU C library include directories in the path list.

Note that these are not the standard versions of the header files included with RVCT. These files have small changes made that allow successful compiling and linking with the GNU header files and libraries.

2.4 Paths to the GNU headers

You should use the glibc header files in order to access Linux-specific functions. You will therefore need to include the paths to these files using the `-J` compiler switch. If CSL_ROOT is the top-level directory of your CodeSourcery installation and GCC_VERSION is the version number of GCC being used, the following paths should be used:

```
$(CSL_ROOT)/lib/gcc/arm-none-linux-gnueabi/$(GCC_VERSION)/include
$(CSL_ROOT)/arm-none-linux-gnueabi/libc/usr/include
$(CSL_ROOT)/include/c++/$(GCC_VERSION)
$(CSL_ROOT)/include/c++/$(GCC_VERSION)/arm-none-linux-gnueabi
```

You may also wish to include these two additional paths:

`$(CSL_ROOT)/arm-none-linux-gnueabi/libc/usr/include/linux`
- for open-source applications that directly include kernel header files and do not specify the "linux/" directory prefix in their sources

`$(CSL_ROOT)/include/c++/$(GCC_VNUM)/backward`
- for legacy applications requiring older C++ header files for backwards compatibility

At the time of writing, the GCC version included with CodeSourcery's toolchain (in the 2006-q1-6 release) is GCC 4.1.0.

2.5 Linking the application

Linker options

The linker must also be passed a number of options to successfully create a Linux executable.

`--sysv` This instructs the linker to use System V library linkage

`--no_startup`

Normally the linker will automatically add a reference to the library entry point function `__main`. However, as we will be using the GNU C library startup code the `__main` function will not be required and we do not need this reference. This switch instructs the linker not to add the reference.

`--no_ref_cpp_init`

This instructs the linker not to add a special reference to the standard C++ initialization function `__cpp_initialize__aeabi_()` when an object file contains global C++ objects whose constructors must be called at startup. We instead use the GNU C library equivalent, `__libc_csu_init()`, to perform this initialization as part of the standard GNU application startup code, and no special linker reference is required.

`--userlibpath=…`

You must specify the paths to the GNU libraries. In a typical CodeSourcery installation, you are likely to need:

```
$(CSL_ROOT)/arm-none-linux-gnueabi/libc/usr/lib
$(CSL_ROOT)/arm-none-linux-gnueabi/libc/lib
$(CSL_ROOT)/arm-none-linux-gnueabi/lib
$(CSL_ROOT)/lib/gcc/arm-none-linux-gnueabi/$(GCC_VNUM)
```

`--no_scanlib`

Prevents the search on the RVCT libraries. See section 3.1.3 for further details.

`--entry _start`

Your image must have its entry point set to the GNU C library initialization code.

`--keep *(.init), --keep *(.fini), --keep *(.init_array), --keep *(.fini_array)`

These options instruct the linker not to remove the C++ global constructor and destructor tables.

`--dynamiclinker=…`

The linker should also be given the path to the dynamic linker that Linux will look for when executing the application. If you do not specify a dynamic linker at build time, or the specified linker is missing, you can manually call a dynamic linker at run-time. For example:

```
> /libeabi/ld-linux.so.3 /opt/bin/eabi/application
```

Optional linker switches

`--rpath=…` This can be used to specify the location of the shared libraries required. Please note this is only available in RVCT 3.0 SP1 (build 586 and later).

Further details of the linker command-line switches specific to building Linux applications can be found in section 6.2 of the RVCT 3.0 Linker and Utilities Guide.

Linking with the library objects

With the correct library paths in place, the GNU libraries should be linked with your application. A full list including a discussion of the functions included is given in section 3.2.

2.6 Building with GNU tools

You may want to build parts of your application with the GNU toolchain. This section describes the steps that you must take.

Use of armlink or GNU ld

We recommend that applications built with RVCT should be linked with `armlink` whenever possible. This is for ease of use when linking with a small number of helper functions which must be taken from the RVCT libraries.

When linking with GNU `ld`, you bear all responsibility for checking that its behavior is as you intend, and does not lead to violation of any license through, for example, the linking of additional libraries.

While it is technically possible to use GNU `ld`, the GNU tools are not under ARM's control and we cannot provide any guarantees about its behavior. This may be of particular concern when linking closed-source applications; ARM can provide no details or guarantees as to whether your use of the GNU linker will pull in IP covered by a non-closed-source license grant without your explicit intention.

You should seek legal advice regarding any such concerns, and ARM is unable to provide such advice.

Linking an application with GNU ld

You will typically only need to link with GNU `ld` where your own application is to be built and linked using the GNU tools and you are linking with a third-party library that has been built using RVCT. Alternatively, you may wish to link a static library into your program that has been created using RVCT.

The only special requirement when using GNU `ld` to link objects compiled with RVCT is to include an appropriate RVCT compiler helper libraries on the command line, for example `h_t__un.l`. Further details on using the RVCT helper libraries can be found in section 3.1.2.

2.7 Creating and using shared libraries

Often for a Linux application you will want to create a dynamic shared library that can be linked with a variety of applications.

Building a shared library with RVCT

Compiler Options

When building dynamic shared libraries, all of the library code must be compiled and linked to be position-independent. To do this, use the `--apcs /fpic` compiler switch.

Linker Options

`armlink` supports the creation of dynamic shared libraries; however this requires some additional options.

`--shared` This instructs the linker to create a dynamic shared library and not a static library

`--soname <name>`

This specifies the SONAME (shared object name) for the library

`--fpic` This enables you to link position-independent code (compiled with `--apcs /fpic`)

For example, to link libfunc.o and asmfunc.o into a dynamic shared library libdynamic.so, you can use the following linker command line:

```
armlink --sysv --fpic --shared --soname libdynamic.so -o libdynamic.so
libfunc.o asmfunc.o libc.so.6
```

Using shared libraries in your application

Shared libraries can be used with `armlink` in the same way as normal libraries by specifying them on the linker command line. References to the shared library will be added to the image and resolved to the library by the dynamic loader at runtime.

Note that the order in which references are resolved is the order in which libraries are specified on the command line. This is also the order in which the dependencies will be resolved by the dynamic linker. In RVCT 3.0 SP1 you can specify the runtime location of libraries using the `--rpath` linker option.

3 Library use considerations

Reminder: ARM cannot provide legal advice regarding the use of open-source code or the compatibility of different licenses. The following discussion is provided for technical reference only. Should you choose to use these instructions or example code in conjunction with your own or third party proprietary software and the GNU C library or any other open source code, you do so entirely at your own risk. ARM makes no representation or warranty as to the legal or business implications of such use. You should consult your own legal advisors if you have any concerns about this.

3.1 Issues with library linkage

There are a number of issues to consider regarding library linkage:

- When linking with the RVCT libraries, there is the risk of linking semihosted I/O functions into an image
- In particular, when linking with the RVCT division functions, the library code may need to be retargeted to avoid the semihosted signal-handling functions. See section 3.3.1 for details.
- A small number of library functions must always be linked into an application when compiling with `armcc`. These are compiler helper functions specific to `armcc` that do not have a public interface (i.e. they are not covered by the ABI)
- You may need very careful control of which files are linked into your application due to licensing restrictions of open-source code.

Avoiding semihosted RVCT functions

During development work, you may wish to add the following pragma directives to your code:

```
#pragma import __use_no_semihosting
#pragma import __use_no_heap_region
```

These causes `armlink` to generate an error if any semihosting library function is used, or if any of the ARM library heap functions (for example `malloc()`) are used.

Linking with compiler helper libraries

Whenever performing a link step to create an application or dynamic shared library, you must always link with an appropriate compiler helper library. Table 2 on page 14 below summarizes which library you should choose for different circumstances.

When distributing a static library to a third party that has been created using RVCT, you may also need to include the objects from an appropriate helper library in your library in case these functions are not available to the user. Please note when manually selecting the libraries that the library names and contents may change between releases of the tools, and your build scripts or makefiles may need changing in the future to reflect this.

If in doubt, the best choice would be the `h_t__uf.l` library as this covers most typical uses, including linking your code into a shared library and cases where the application uses some Thumb code. You should consult the RVDS End-User License Agreement (EULA) for details of the restrictions on redistribution of the RVCT libraries, including the helper libraries.

Table 2 Compiler helper libraries

ARM/Thumb code in your application or library?	Application or library?	Library to use
ARM code only	Application	`h_a__un.l`
Thumb code only or mixed ARM/Thumb code	Application	`h_t__un.l`
ARM code only	Shared library or Application	`h_a__uf.l`
Thumb code only or mixed ARM/Thumb code	Shared library or Application	`h_t__uf.l`

Controlling the functions that your application links with

Whether you are building a closed or open source application, you may want to ensure that no intellectual property covered by an incompatible license is linked into your application.

The best solution in this respect is to use the `--no_scanlib` linker option. This leaves the library selection under your full and explicit control. If you do not use this switch, the linker will by default search the system library path as given by the `RVCT30LIB` environment variable or the `--libpath` option. As a result, the linker may pull in additional functions that you did not intend to use. The details of the linker's library searching behavior can be found in section 7.2 of the RVCT 3.0 Linker and Utilities Guide.

For additional checks on which library functions are being included by the linker, you should consult the linker's verbose output. You can obtain this using the `--verbose` option, and redirected to a file using `--errors filename.txt`. This will produce full output including details of how the linker is searching for and selecting functions from libraries to resolve references in your object files.

3.2 Libraries used by the example code

The libraries and library objects listed in Table 3 are linked to in the example code and are expected to be used by most applications. This is intended only as a reference and a starting point for the legal checks you should carry out on the compatibility of the various licenses covering the libraries you need to use.

Table 3 Libraries used in the example makefiles

Library	Description
`libc.so.6, libc_nonshared.a`	These constitute the main GNU C library (`glibc`). All of the examples provided link to the dynamic library plus the non-shared portions in `libc_nonshared.a`. In most cases this will be preferred, as you will only require one copy of the library in your target's filesystem. Alternatively, for a standalone, statically-linked application you can link to `libc.a`.
`libm.so.6`	This is the math library accompanying `glibc`.
`crt1.o, crti.o, crtn.o`	These objects contain the GNU application startup code, including the entry point function `_start`
`libgcc_s.so libgcc.a`	These are the compiler helper functions included with GCC. These include ABI functions that are not part of the standard C library, for example the division functions.
`libstdc++.so.6, libsupc++.a`	These are the GCC C++ libraries, including the standard template library and support functions. These are not needed if your application uses only C code.

3.3 Library linkage under particular licensing conditions

This section provides a very broad overview of the mechanisms that you can use when linking a closed-source application or one containing material covered by one or more open-source licenses. This is a purely technical overview and for all legal concerns you should consult your own legal advisors.

Linking closed-source applications

In the closed-source case, you may use additional features provided by the RVCT libraries provided that your use of the libraries is permitted by the RVDS End-User License Agreement. There are two options:

- Use `--no_scanlib` and explicitly specify one or more RVCT libraries or their member functions on the linker command line.

  This provides full control over exactly which functions are used. However, you must be careful which libraries you use for different circumstances. In particular, you must use the position independent variant of a library whenever linking its objects into a dynamic shared library. You should also use the Thumb variant of a library whenever any Thumb code is present in your application. Details of library naming can be found in section 5.16 of the RVCT 3.0 Compiler and Libraries Guide.

- Allow the linker to search the RVCT libraries as it would by default.

You should ensure that all input and output routines are provided by additional libraries such as the GNU C library (`glibc`), which are given on the linker command line. Also take additional care that no other semihosting functions or heap-using functions are linked into your image. See "Avoiding semihosted RVCT functions" in section 3.1.1 above for further details.

Also note that the ARM libraries provide a complete signal-handling interface that can be linked into an application for a number of reasons, such as to handle division by zero. These signal-handling functions are required in a bare-metal environment. However, they may not be needed in a Linux application and in such cases should be retargeted to use Linux system calls. A detailed discussion of such retargeting is beyond the scope of this Application Note. The same ABI function from the GNU libraries will be functionally equivalent, without the need for any additional work.

As with any other application, you are responsible for ensuring that the objects and libraries you provide to the linker are covered by compatible licenses. If you are concerned about linking with code from an incompatible license, it is recommended that you use the `--no_scanlib` option as above and select the library functions used in your image very carefully. In such circumstances, we again suggest that you speak to your legal advisors about such licensing concerns.

Linking open-source applications

With open-source applications, it is important to understand the licensing terms your code is covered by and the implications of those terms. You must be happy that everything you link with is covered by compatible license terms. Please note that the RVCT C and C++ libraries are not available under an open-source license.

The only safe way of doing this is to use the `--no_scanlib` linker option and explicitly specify the libraries that you wish to link with.

Interoperation of toolchains

When using a combination of the GNU tools and RVCT to build an application or library, additional care must be taken.

Limitations and restrictions on builds shared between toolchains

There are a number of differences between GCC and the RVCT compiler that may affect your ability to build the code with either toolchain. In particular, note that:

- GCC inline assembly code is not compatible with RVCT and vice versa
- Likewise, standard assembly language files cannot be built by both `armasm` and the GNU assembler (`gas`) as they use different syntax
- RVCT 3.0 supports the C90 standard with some elements of C99, rather than the full C99 standard. You may find that you need to make some changes to your source code to take account of this, if it uses C99 features
- Certain language extensions supported by GCC are not implemented by RVCT; this includes any implementation-specific `#pragma directives`

In some circumstances, you may be able to conditionally define around such code to allow building with both toolchains where this is important.

Limitations on builds using objects generated by both toolchains

Other limitations restrict how you can use objects compiled with both toolchain in the same application or library:

- Linking C++ objects compiled with both RVCT and GCC is particularly difficult from a legal perspective and is not recommended by ARM. The interfaces between C++ objects are complex in nature at both the binary and source levels. The simplest solution is to link an entire C++ application using only one toolchain.
- "When linking RVCT objects using GNU `ld`, the RVCT objects must have been compiled using `--dwarf2`. By default RVCT uses DWARF3 debug data, however GNU `ld` is unable to read DWARF3 debug data. Therefore the objects must be compiled with DWARF2 debug data.
- For C applications and C++ objects whose interfaces are all C (i.e. all functions and data are declared extern "C"), there should be no barriers. However, you are responsible for ensuring that all of the intellectual property in your application is used in a way that does not violate their license terms.

4 Example code

Included with this Application Note are several examples, demonstrating the build scripts needed to create a Linux application.

4.1 Makefile templates and standard build rules

The examples each include a makefile and link to the necessary retargeting code. The makefiles are all based around a series of templates that will allow you to rapidly adapt the examples to your own builds.

Configuration

Before the the supplied makefiles can be used they need be configured for the location and version of the CodeSourcery and RVCT installed. This can be done either automatically or manually.

Automatic Configuration

To configure the makefile automatically enter "make configure" from within the examples directory. This should detect the different versions of the tools installed on your computer and create `common/an150_config.mk` to store this information. This can be updated at a later time using "make reconfigure".

For the automatic configuration to be successful you must have the make utility, RVCT and the CodeSourcery tools on your PATH environment variable.

Manaul Configuration

If the automatic configuration is unsuccessful you can manually configure the make files. This is done by creating the file `an150_config.mk` in the common directory. Within this file you will need to declare the variables detailed below in Table 4.

Table 4 an150_config.mk configuration variables

Makefile variable	Description

`CONFIG_CSL_ROOT`	Indicates the root of your CodeSourcery installation. On Linux this will be the directory in which you uncompressed the toolchain binaries package; on Windows this will be the directory chosen in the installer.
	For example:
	CONFIG_CSL_ROOT=/opt/codesourcery (Linux)
	CONFIG_CSL_ROOT=c:\codesourcery (Windows)

`CONFIG_SP1`	This is used to indicate whether RVCT 3.0 (build 441) or RVCT 3.0 Service Pack 1 (build 586) is installed. RVCT 3.0 SP1 : CONFIG_SP1=1 RVCT 3.0 : CONFIG_SP1=0

Table 4 an150_config.mk configuration variables (continued)

Makefile variable	Description
`CONFIG_CSL_REL`	Configure the make files for the CodeSourcery release being used. For example CONFIG_CSL_REL=2006q1-6
`CONFIG_CSL_GCC_VNUM`	Specifies the GCC version number installed For Example CONFIG_CSL_GCC_VNUM =4.1.0
`CONFIG_CSL_CRT1_PATH`	If you are using the 2006-Q1-3 CodeSourcery release, you must manually split the `crt1.o` object file. You must then use this to point to the strip version. If you are not using the 2006-Q1-3 CodeSourcery release this should point to the standard `crt1.o` object file. For Example CONFIG_CSL_CRT1_PATH =$(CONFIG_CSL_ROOT)/arm-none-linux-gnueabi/libc/usr/lib/crt1.o Please see section 5.1 Frequently Asked Questions for further details on `crt1.o`.

Application variables

The build rules in the template makefiles are controlled by a number of variables in the application makefile, as given below in Table 5.

Table 5 an150_config.mk configuration variables

Makefile variable	Description
`CFLAGS`	Allows application-specific options to be given to the compiler. Most of the necessary options are provided by the makefile build rules, therefore the only options typically needed here are those required by your application. For example, you may use `-D` options to pass macro definitions to the compiler, optimization levels, or `-I` to specify your application's include paths.
`LDFLAGS`	Allows application-specific options to be given to the linker. Most of the necessary options are provided by the makefile build rules, therefore the only options typically needed here are user library paths and switches that generate information about the application image, such as `--info` or `--verbose`.
`OBJS`	Lists the objects that must be compiled for the application
`LIBS`	Lists any additional libraries that your application links to
`IMAGE`	Specifies the output filename for the image
`ENABLE_DEBUG`	If this is defined, debug information is left in the image. When this is not defined, debugging information is removed at the link step and `fromelf` is used on the final image to remove additional debug-related sections.

Makefile build rules

A number of functions are provided by the template makefiles for different build steps:

Table 6 Makefile build rules

Makefile function name	Parameters	Description
`compile_app`	source file, object file	Compiles a source file using `armcc`. For example: `$(call compile_app,foo.c,foo.o)` compiles `foo.c` into the object file `foo.o`.
`compile_lib`	source file, object file	Compiles a source file using `armcc`, built for linking into a shared library.
`Gcc_app`	source file, object file	This rule is provided for convenience, to compile a source file using CodeSourcery's GCC.
`Gcc_lib`	source file, object file	Compiles a source file using CodeSourcery's GCC, built for linking into a shared library.
`assemble_app`	source file, object file	Uses `armasm` to assemble a source file.
`assemble_lib`	source file, object file	Uses `armasm` to assemble a source file, and marks the code as position-independent and therefore suitable for linking into a shared library.
`link_app`	image file	Links the objects specified by `$(OBJS)` into the image file
`link_lib`	library file, soname	Links the objects specified by `$(OBJS)` into the dynamic library and gives it the specified shared object name. For example: `$(call link_lib,libfoo.so.1.2,libfoo.so.1)` links the library `libfoo.so.1.2` but gives it the shared object name `libfoo.so.1`. This is commonly used to maintain different minor versions of a library in a filesystem.
`gld_app`	image file	Links the objects specified by `$(OBJS)` into the image file using GNU `ld`
`gld_lib`	image file, soname	Links the objects specified by `$(OBJS)` into the dynamic library using GNU `ld`, and gives it the specified shared object name
`fromelf_image`	input file, output file	Runs `fromelf` to strip debug data from an image
`strip_image`	image file	Runs GNU strip on an image to remove debug data. This rule strips the file in place and does not create a separate, stripped copy.
`armar_lib`	archive file	Creates (or updates) a static library using `armar` with the objects specified by the `$(OBJS)` variable
`clean_files`	file list	This rule is provided to delete the specified files

Additional compiler and linker options

The template makefiles use some additional compiler and linker options that you may find useful when creating your own build scripts. These are described in Table 7.

Table 7 Additional compiler and linker options

Switch	Description
`--diag_suppress 6318,6319,6765`	The linker will typically generate a few warnings when building Linux applications, as described below. This switch suppresses these warnings. 6318 - These warnings are generated because GCC uses linker relocations for references internal to each object, whereas all such references are resolved at compile-time by `armcc`. The targets of these relocations may not have appropriate mapping symbols that allow the linker to determine whether the target is code or data, so ordinarily a warning will be generated 6319 - The standard makefiles use `--keep` to retain any .init, .fini, .init_array and .fini_array sections in your objects. This suppresses the warning that the linker is ignoring the `--keep` option when these are not present. 6765 - This is due to no build attributes being present in the GNU entry point code found in `crt1.o`, so the linker cannot determine which state (ARM or Thumb) the code will run in. It is safe to ignore this as the application entry code is ARM code.
`--bss_threshold=0`	This switch instructs the compiler not to move small pieces of ZI (BSS) data into the read/write data section of the image. This reduces the size of the image.

4.2 Simple example code

"Hello world"

This example shows how to build a very trivial application with RVCT and link with the GNU C library. It can be used as a template for compiling your own standalone applications with RVCT.

Dhrystone

The Dhrystone benchmark is included as an example of a small but non-trivial application which makes use of further C library and operating system functions.

You may again wish to use this example as the basis for your application. The makefile variables and layout are the same as the "hello world" example.

Trivial C++ example

This is the first example that demonstrates the use of C++, this the same as the example included with RVDS 3.0.

4.3 Library examples

Two library examples are provided, with complete makefiles.

Simple C and assembly code library (example_library directory)

The first library example demonstrates the tool options required to build both static and dynamic libraries with RVCT. The makefile for this example contains six main targets:

libsorts_static.a

This is a static library containing the example code, created using `armar`. It may be possible to create libraries with GNU `ar` and `ranlib`, however there are some known implementation-defined differences between the library symbol tables generated by the GNU tools. Where possible, you should therefore use `armar` provided with RVCT.

libsorts_dynamic.so

This is a dynamic library containing the same code as the `libsorts_static.a` target.

In particular for a dynamic library, the `--apcs /fpic` switch is given to the compiler, and `--fpic` is given to the linker. This indicates that you wish to generate position-independent code that can be located at any arbitrary virtual address.

sorts_static_rvct.axf, sorts_dynamic_rvct.axf

These are the application executables, linked against the respective libraries using `armlink`.

sorts_static_gld.axf, sorts_dynamic_gld.axf

These are similar to the above targets, however they are linked with GNU `ld`.

C++ example library (cpp_library directory)

This shows a simple C++ example that demonstrates the use of C++ class member functions in a library, based on the standalone C++ example. The library also contains definitions of static objects whose constructors are called during application startup.

The main difference in the startup procedure relates to how constructors for static C++ objects are called. For Linux applications there are two mechanisms used.

Firstly, constructors listed in the dynamic segment of a library (dynamic shared object, or DSO) are called by the dynamic loader. The dynamic loader uses other information in the dynamic segment to determine the dependencies of each library or application and initialize those first. Once this is complete, control is transferred to the application's initialization code.

Secondly, further constructors that are local to the application may be called if necessary. In a bare-metal application built using the RVCT libraries, the `__cpp_initialize__aeabi_()` library function calls these constructors. In a Linux application, the RVCT linker will resolve the necessary references from the corresponding GNU initialization functions to allow this code to be used instead.

Notes on constructors for static C++ objects

When developing C++ applications, you should be careful not to rely on the order in which static objects' constructors are called. Note that the ARM linker will not guarantee the order in which they are called at runtime, as the C++ standard does not guarantee the order in which the constructors are called.

The main consequence of this will be that the same code in a static library or a dynamic library may have its constructors called in a different order. The `cpp_library` example demonstrates this. When statically linked as part of the application executable, the C++ objects constructed in the library will have a different number than when the library is dynamic.

This is only of importance when you must rely on the ordering of constructors, for example where there are interdependencies between the objects. You can use the following points as a guide:

- All dynamic library objects will be initialized after their dependencies and before those from static libraries and your main application. However, objects within the same dynamic library cannot have the order of their constructors guaranteed
- Libraries that are statically-linked into your application will have their constructors called with the application's constructors during the C library initialization
- All statically-linked library and application constructors will be called together, and the order in which they are called cannot be guaranteed.

5 Troubleshooting and Frequently Asked Questions

This section provides additional information and answers to common questions.

5.1 Frequently Asked Questions

Where can I find further information?

The recommended starting point for further information is the CodeSourcery toolchain FAQ at: `http://www.codesourcery.com/gnu_toolchains/arm/faq.html`

You may also wish to look at ARM and Linux forums and newsgroups, or looking at mailing list archives. `http://www.arm.linux.org.uk/` provides resources relating specifically to ARM Embedded Linux.

Note that ARM does not provide support on the use of the GNU tools. See section 1.1 for details.

How do I build an EABI-compliant Linux kernel?

Prior to version 2.6.16 there is no such thing as an EABI-compliant kernel. This is due to the large amount of assembly code in the kernel that would require significant time and effort to rewrite. However, this is not important as the EABI-compliant GNU C library translates calls appropriately from EABI-compliant applications to the non-EABI compliant kernel system calls.

From kernel version 2.6.16, it is possible to build an EABI kernel. However, this is only an issue for applications and libraries which directly access kernel structures or functions.

Which kernel version should I use?

The CodeSourcery toolchain as provided in binary form is built to use NPTL (Native POSIX Thread Library) and it expects to have TLS (thread-local storage) support in the kernel. According to CodeSourcery, the `glibc` initialization code requires kernel version 2.6.16 or later. Alternatively, your Linux distributor may have already applied the appropriate patches to their kernel build. You should contact your Linux distributor for more information.

Why do you need to pass so many options to the compiler?

The ARM compiler and linker are primarily used to create standalone, bare-metal applications. The options described in section 2 and shown in the example makefiles make necessary changes to the application image, for correct operation in an ARM Linux environment.

Can you use EABI-compliant and non-EABI-compliant applications together?

Yes. You should place the libraries and the dynamic linker in a different directory to the normal libraries. We recommend that you use `/libeabi` for the EABI-compliant libraries, and leave the original, non-EABI compliant libraries in `/lib`.

You must then set the library search path for EABI applications using the environment variable `LD_LIBRARY_PATH=/libeabi` or by using the `--rpath` linker option. We recommend that you rebuild all applications to use the EABI in your final system as the extra libraries take up a significant amount of space in the file system.

The compiler generates errors like "Identifier va_list is undefined" when building GNU-style code.

This is because of slightly differing methods of implementing variadic functions between RVCT and GCC. The solution is to use the compiler option `--preinclude stdarg.h` to include the definitions of these types before the start of the application code.

I receive warning "L6765W: Shared object entry point must be ARM state with v4T"

The GNU C library startup code does not contain all of the build attributes that the linker uses to determine its execution state. As a result the linker is unable to confirm that the entry point is in ARM code. As the GNU C library startup code is ARM code, you can safely ignore this warning, or suppress this as in the examples with `--diag_suppress 6765`.

The GNU tools report "ERROR: Source object … has EABI version 5, but target … has EABI version 4" when used on objects generated by RVCT 3.0

RVCT 3.0 generates ELF files conforming to the latest revision of the ABI. Specifically, they implement revision 5 of the AAELF (ARM ABI ELF) specification. However, CodeSourcery's 2005-Q3-2 release only supports revision 4 of the AAELF specification and will not consume objects produced by RVCT 3.0 tools. Support for the new ABI revision is included in the 2006-Q1-3 and later releases of the CodeSourcery toolchain.

armlink reports "Fatal error: L6033U: Symbol in crt1.o is defined relative to an invalid section."

In the 2006-Q1-3 release of the CodeSourcery toolchain the `crt1.o` object file has not been correctly stripped. The supplied example scripts ensure that `crt1.o` object is correctly stripped before linking. This has been fixed in the 2006-Q1-6 CodeSourcery release. Alternatively you can strip the `crt1.o` object yourself.

Intermixing RVCT hardware VFP objects and GNU libraries

The GNU libraries as CodeSourcery supply in their ready-built binary packages are not built with hardware VFP support and return their results in the ARM core's registers. However RVCT object files that are built for hardware VFP will expect the result to be returned in the VFP co-processor registers and not in the ARM core's registers.

The recommended solution is compile with `--fpu SoftVFP+VFP`. However, you can also workaround this by adding `#pragma softfp_linkage` before your list of `#include` lines, and `#pragma no_softfp_linkage` afterwards. For example:

```
#pragma softfp_linkage
#include <stdlib.h>
#pragma no_softfp_linkage
```

This will instruct `armcc` to expect the results to be placed in the ARM core's registers when calling these library functions. However you will not get the full benefit of the hardware VFP.

In a large application armlink reports "L6016U: Symbol table missing/corrupt in object/library <object>."

This is because the GNU `ar` command can generate incompatible information. Replace `ar` with `armar` and use the same command line arguments. Alternatively, the error is recoverable by using "`armar  -s`" to rebuild the symbol table.

Common problems with running your application

Some common problems with running your application are described in Table 8.

Table 8 Common problems with running applications

Problem	Solution
Cannot find the application	• Check that you have set the executable flag for the program (use `chmod +x` program) • Check that the application is on the path, or you are running it with ./program in the current directory • The dynamic loader may not be the same as specified at link time. In this case, use /path-to-linker/dynamic-loader program-path/program For example: `/libeabi/ld-linux.so.3 /opt/bin/eabi/hello`
"GLIBC_2.4 not found" error "unable to find library XXX.so.X"	This is the dynamic linker reporting that it cannot use the libraries found on its default path. You can use the `LD_LIBRARY_PATH` environment variable to access the correct libraries. For example: `LD_LIBRARY_PATH=/libeabi ./helloworld` Alternatively you can use the `--rpath` linker option.
"Illegal instruction" error before `main()`	This indicates that the image has been built for the incorrect architecture (e.g. v6 code running on a v5TE core), or the kernel has not been built with NPTL support. Check that you have built the image for the correct ARM architecture and check that you are using either a 2.6.12 (or later) Linux kernel or one with the appropriate patches applied as part of your distribution. Once at main this is likely to be an actual undefined instruction in the application.
Various Dynamic linker errors	The pre-built libraries supplied in the 2006-Q1-3 release of the CodeSourcery toolchain are built with an ABI tag that require Linux kernel 2.6.16 or later. The dynamic linker will generate various error messages when run on older kernel revisions. To avoid this you will need to update to the 2.6.16 or later kernel or rebuild the libraries to support an older kernel revision.

Segmentation faults

There are a variety of possible causes of segmentation faults. They may be caused by problems with your application. You should also ensure that:

- You have linked against the GNU C library and used the `--no_scanlib` linker switch. Even for a dynamic library, if you do not link against glibc, `armlink` may statically link the semihosted I/O functions from the RVCT libraries into your application or dynamic library
- When creating a dynamic library, you have compiled and linked as position-independent code (use `--apcs /fpic` for the compiler and `--fpic` for the linker)
- When creating an application, you have used the linker switches `--nostartup` and `--entry _start`

5.2 Image sizes and stripping debug data

Both the GNU and ARM toolchains add a significant amount of information to an image that is generally only of use for debugging.

For production embedded Linux systems, it is likely that you would want to strip the debugging data from your applications and shared libraries. With RVCT, this can be removed using the `--no_debug` switch either at the link stage or by running `fromelf` on the linked image. In addition, you can use `fromelf` on the linked image to remove debug information and the "`.comment`" sections from the file. For example:

```
fromelf --no_debug --no_comment_section --elf -o stripped.axf image.axf
```

The example makefiles perform the above `fromelf` step by default. If you wish to retain the debug tables and the "`.comment`" section you can define the `ENABLE_DEBUG` variable.

In addition, the data sizes in RVCT images can be slightly larger than those in GNU images. This is typically because some ZI data (BSS) is moved into the RW data area for performance reasons on bare-metal systems. You can move this data to ZI (also called BSS) sections using the compiler switch `--bss_threshold=0`.

6 References and further information

For further information on the GNU toolchain supplied by CodeSourcery, please see:
`http://www.codesourcery.com/gnu_toolchains/arm/`

In particular, the FAQ for the ARM GNU toolchain can be found at:
`http://www.codesourcery.com/gnu_toolchains/arm/faq.html`

The full documentation of the ABI for the ARM Architecture can be found at:
`http://www.arm.com/products/DevTools/ABI.html`

The ARM GNU/Linux ABI Supplement can also be found at:
`http://www.codesourcery.com/gnu_toolchains/arm/arm_gnu_linux_abi.pdf`

Prebuilt kernel binaries and an example file system image for ARM's development boards are available from the ARM website at:
`http://www.arm.com/linux/`

Additional information on the ARM tools can be found in the relevant RVCT 3.0 documentation:
RVCT 3.0 Compiler and Libraries Guide (ARM DUI 0205G)
RVCT 3.0 Linker and Utilities Guide (ARM DUI 0206G)
RVCT 3.0 Developer Guide (ARM DUI 0203G)

General information on ARM Linux can be found from the open-source community. Useful starting points are:

- The comp.sys.arm newsgroup
- The ARM Linux project website: `http://www.arm.linux.org.uk/`