



## Contents

### 1. Multiarch paths and toolchain implications

#### 1. Overview and goals

#### 2. Loading/running an executable

##### 1. ELF interpreter

##### 2. Dynamic libraries

#### 3. Building an executable from source

##### 1. Directory prefixes

##### 2. Multilib suffixes

##### 3. Detailed directory search rules

#### 4. Multiarch impact on the toolchain

##### 1. ELF interpreter

##### 2. Shared library search paths

##### 3. GCC and toolchain directory paths

##### 4. Multiarch and cross-compilation

##### 5. Multiarch and multilib

#### 5. Summary

# Multiarch paths and toolchain implications

## Overview and goals

Binary files in packages are usually platform-specific, that is they work only on the architecture they were built for. Therefore, the packaging system provides platform-specific versions for them. Currently, these versions will install platform-specific files to the same file system locations, which implies that only one of them can be installed into a system at the same time.


The goal of the "multiarch" effort is to lift this limitation, and allow multiple platform-specific versions of the same package to be installed into the same file system at the same time. In addition, each package should install to the same file system locations no matter on which host architecture the installation is performed (that is, no rewriting of path names during installation).

This approach could solve a number of existing situations that are not handled well by today's packaging mechanisms:

- Systems able to run binaries of more than one ISA natively.

- Support for multiple incompatible ABI variants on the same ISA.
- Support for processor-optimized ABI-compatible library variants.
- NFS file system images exported to hosts of different architectures.
- Target file systems for ISA emulators etc.
- Development packages for cross-compilation.

In order to support this, platform-specific versions of a multiarch package must have the property that for each file, it is either 100% identical across platforms, or else it must be installed to separate locations in the file system.

The latter is the case at least for executable files, shared libraries, static libraries and object files, and to some extent maybe header files. This means that in a multiarch world, such files must move to different locations in the file system than they are now. This causes a variety of issues to be solved; in particular, most of the existing locations are defined by the  [FHS](#) and/or are assumed to have well-known values by various system tools.

In this document, I want to focus on the impact of file system hierarchy changes to two tasks in particular:

- loading/running an executable
- building an executable from source

In the following two sections, I'll provide details on file system paths are currently handled in these two areas. In the final section, I'll discuss suggestions how to extend the current behavior to support multiarch paths.

## Loading/running an executable

Running a new executable starts with the `execve()` system call. The Linux kernel supports execution of a variety of executable types; most commonly used are

- native ELF executable
- ELF executable for a secondary native ISA (32-bit on 64-bit)
- `#!` scripts
- user-defined execution handlers (via `binfmt_misc`)

The binary itself is passed via full (or relative) pathname to the `execve` call; the kernel does not make file system hierarchy assumptions. By convention, callers

of `execve` usually search well-known path locations (via the `PATH` environment variable) when locating executables. How to adapt these conventions for multiarch is beyond the scope of this document.

With `#!` scripts and `binfmt_misc` handlers, the kernel will involve a user-space helper to start execution. The location of these handlers themselves and secondary files they in turn may require is provided by user space (e.g. in the `#!` line, or in the parameters installed into the `binfmt_misc` file system). Again, adapting these path names is beyond the scope of this document.

## ELF interpreter

For native ELF executables, there are two additional classes of files involved in the initial load process: the ELF interpreter (dynamic loader), and shared libraries required by the executable.

The ELF interpreter name is provided in the `PT_INTERP` program header of the ELF executable to be loaded; the kernel makes no file name assumptions here. This program header is generated by the linker when performing final link of a dynamically linked executable; it uses the file name passed via the `-dynamic-linker` argument. (Note that while the linker will fall back to some hard-coded path if that argument is missing, on many Linux platforms this default is in fact incorrect and does not correspond to a ELF interpreter actually installed in the file system in current distributions. Passing a correct `-dynamic-linker` argument is therefore mandatory.)

In normal builds, the `-dynamic-linker` switch is passed to the linker by the GCC compiler driver. This in turn gets the proper argument to be used on the target platform from the specs file; the (correct) default value is hard-coded into the GCC platform back-end sources. On bi-arch platforms, GCC will automatically choose the correct variant depending on compile options like `-m32` or `-m64`. Again, the logic to do so is hard-coded into the back-end. Unfortunately, various bi-arch platforms use different schemes today:

	32-bit	64-bit
amd64	<code>/lib/ld-linux.so.2</code>	<code>/lib64/ld-linux-x86-64.so.2</code>
ia64	<code>/lib/ld-linux.so.2</code>	<code>/lib/ld-linux-ia64.so.2</code>
mips	<code>/lib/ld.so.1</code>	<code>/lib64/ld.so.1</code>
ppc	<code>/lib/ld.so.1</code>	<code>/lib64/ld64.so.1</code>
s390	<code>/lib/ld.so.1</code>	<code>/lib/ld64.so.1</code>

sparc      /lib/ld-linux.so.2      /lib64/ld-linux.so.2

glibc upstream has a [reference](#) of all ELF interpreter paths and there is a list of [dynamic linker path collisions](#) available.

## Dynamic libraries

Once the ELF interpreter is loaded, it will go on and load dynamic libraries required by the executable. For this discussion, we will consider only the case where the interpreter is ld.so as provided by glibc.

As opposed to the kernel, glibc does in fact *\*search\** for libraries, and makes a variety of path name assumptions while doing so. It will consider paths encoded via `-rpath`, the `LD_LIBRARY_PATH` environment variable, and knows of certain hard-coded system library directories. It also provides a mechanism to automatically choose the best out of a number of libraries available on the system, depending on which capabilities the hardware / OS provides.

Specifically, glibc determines a list of search directory prefixes, and a list of capability suffixes. The directory prefixes are:

- any directory named in the (deprecated) `DT_RPATH` dynamic tag of the requesting object, or, recursively, any parent object (note that `DT_RPATH` is ignored if `DT_RUNPATH` is also present)
- any directory listed in the `LD_LIBRARY_PATH` environment variable
- any directory named in the `DT_RUNPATH` dynamic tag of the requesting object (only)
- the system directories, which are on Linux hard-coded to

```
/lib$(biarch_suffix)
/usr/lib$(biarch_suffix)
```

where `$(biarch_suffix)` may be "64" on 64-bit bi-arch platforms.

The capability suffixes are determined from the following list of capabilities:

- For each hardware capability that is present on the hardware as indicated by a bit set in the `AT_HWCAP` auxillary vector entry, and is considered "important" according to glibc's hard-coded list of important hwcaps (platform-dependent), a well-known string provided by glibc's platform back-end.

- For each "extra" hardware capability present on the hardware as indicated by a GNU NOTE section in the vDSO provided by the kernel, a string provided in that same note.
- A string identifying the platform as a whole, as provided by the kernel via the AT\_PLATFORM auxillary vector entry.
- For each software capability supported by glibc and the kernel, a well-known string. The only such capability supported today is "tls", indicating support for thread-local storage.

The full list of capability suffixes is created from the list of supported capabilities by forming every sub-sequence. For example, if the platform is "i686", supports the important hwcap "sse2" and TLS, the list of suffixes is:

```
sse2/i686/tls
sse2/i686
sse2
i686/tls
i686
tls
<empty>
```

The total list of directories to be searched is then formed by concatenating every directory prefix with every capability suffix. Various caching mechanisms are employed to reduce the run-time overhead of this large search space.

Note: This method of searching capability suffixes is employed only by glibc at run time; it is unknown to the toolchain at compile time. This implies that an executable will have been linked against the "base" version of a library, and the "capability-specific" version of the library is only substituted at run time. Therefore, all capability-specific versions must be ABI-compatible to the base version, in particular they must provide the same soname and symbol versions, and they must use compatible function calling conventions.

## Building an executable from source

For this discussion, we only consider GCC and the GNU toolchain, installed into the usual locations as system toolchain, and in the absence of any special-purpose options (-B) or environment variables (GCC\_EXEC\_PREFIX, COMPILER\_PATH, LIBRARY\_PATH ...).

However, we do consider the three typical modes of operation:

- native compilation
- cross-compilation using a "traditional" toolchain install
- cross-compilation using a sysroot

## Directory prefixes

During the build process, the toolchain performs a number of searches for files. In particular, it looks for (executable) components of the toolchain itself; for include files; for startup object files; and for static and dynamic libraries.

In doing so, the GNU toolchain considers locations derived from any of the following "roots":

- Private GCC installation directories

`/usr/lib/gcc`

`/usr/libexec/gcc`

These hold both GCC components and target-specific headers and libraries provided by GCC itself.

- GNU toolchain installation directories

`/usr/$(target)`

These directories hold files used across multiple components of the GNU toolchain, including the compiler and binutils. In addition, they may also hold target libraries and headers; in particular for libraries traditionally considered part of the toolchain, like newlib for embedded systems. In fact, in the "traditional" installation of a GNU cross-toolchain, *\*all\** default target libraries and headers are found here. However, the toolchain directories are always consulted for native compilation as well (if present)!

- The install "prefix" This is what is specified via the `--prefix` configure option. The toolchain will look for headers and libraries under that root, allowing for building and installing of multiple software packages that depend on each other into a shared prefix. (This is really only relevant when the toolchain is *\*not\** installed as system toolchain, e.g. in a setup where you provide a GNU toolchain + separately built GNU packages on a non-GNU system in some distinct non-system directory.)
- System directories

```
/lib$(biarch_suffix)
/usr/lib$(biarch_suffix)
/usr/local/lib$(biarch_suffix)
/usr/include
/usr/local/include
```

These are default locations defined by the OS and hard-coded into the toolchain sources. The examples above are those used for Linux. On bi-arch platforms, `$(biarch_suffix)` may be the suffix "64". These directories are used only for native compilation; however in a "sysroot" cross-compiler, they are used for cross-compilation as well, prefixed by the sysroot directory.

## Multilib suffixes

In addition to the base directory paths referred to above, the GNU toolchain supports the so-called "multilib" mechanism. This is intended to provide support for multiple incompatible ABIs on a single platform. This is implemented by the GCC back-end having hard-coded information about which compiler option causes an incompatible ABI change, and a hard-coded "multilib" directory suffix name corresponding to that option. For example, on PowerPC the `-msoft-float` option is associated with the multilib suffix "nof", which means libraries using the soft-float ABI (passing floating point values in integer registers) can be provided in directories like:

```
/usr/lib/gcc/powerpc-linux/4.4.4/nof
/usr/lib/nof
```

The multilib suffix is appended to all directories searched for libraries by GCC and passed via `-L` options to the linker. The linker itself does not have any particular knowledge of multilibs, and will continue to consult its default search directories if a library is not found in the `-L` paths. If multiple orthogonal ABI-changing options are used in a single compilation, multiple multilib suffixes can be used in series.

As a special consideration, some compiler options may correspond to multiple incompatible ABIs that are already supported by the OS, but using directory names differently from what GCC would use internally. As the typical example, on bi-arch systems the OS will normally provide the default 64-bit libraries in `/usr/lib64`, while also providing 32-bit libraries in `/usr/lib`. For GCC on the other side, 64-bit is the default (meaning no multilib suffix), while the `-m32` option is associated with the multilib suffix "32".

To solve this problem, the GCC back-end may provide a secondary OS multilib suffix which is used in place of the primary multilib suffix for all library directories derived from `*system*` paths as opposed to GCC paths. For example, in the typical bi-arch setup, the `-m32` option is associated with the OS multilib suffix `"../lib"`. Given the that primary system library directory is `/usr/lib64` on such systems, this has the effect of causing the toolchain to search

<code>&lt;default&gt;</code>	<code>-m32</code>
<code>/usr/lib64/gcc/powerpc64-</code>	<code>/usr/lib64/gcc/powerpc64-</code>
<code>linux/4.4.4</code>	<code>linux/4.4.4/32</code>
<code>/usr/lib64</code>	<code>/usr/lib64/../lib (i.e. /usr/lib)</code>

## Detailed directory search rules

The following rules specify in detail which directories are searched at which phase of compilation. The following parameters are used:

- `$(target)`  
GNU target triple (as specified at configure time)
- `$(version)`  
GCC version number
- `$(prefix)`  
Determined at configure time, usually `/usr` or `/usr/local`
- `$(libdir)`  
Determined at configure time, usually `$(prefix)/lib`
- `$(libexecdir)`  
Determined at configure time, usually `$(prefix)/libexec`
- `$(tooldir)`  
GNU toolchain directory, usually `$(prefix)/$(target)`
- `$(gcc_gxx_include_dir)`  
Location of C++ header files. Determined at configure time:
  - If `--with-gxx-include-dir` is given, the specified directory
  - Otherwise, if `--enable-version-specific-runtime-libs` is given:  
`$(libdir)/gcc/$(target)/$(version)/include/c++`
  - Otherwise for all cross-compilers (including with `sysroot`!):  
`$(tooldir)/include/c++/$(version)`
  - Otherwise for native compilers:  
`$(prefix)/include/c++/$(version)`



- `$(multi)`  
Multilib suffix appended for GCC include and library directories
- `$(multi_os)`  
OS multilib suffix appended for system library directories
- `$(sysroot)`  
Sysroot directory (empty for native compilers)

Directories searched by the compiler driver for executables (cc1, as, ...):

GCC directories	<code>\$(libexecdir)/gcc/\$(target)/\$(version)</code>
	<code>\$(libexecdir)/gcc/\$(target)</code>
	<code>\$(libdir)/gcc/\$(target)/\$(version)</code>
	<code>\$(libdir)/gcc/\$(target)</code>
Toolchain directories	<code>\$(tooldir)/bin/\$(target)/\$(version)</code>
	<code>\$(tooldir)/bin</code>

Directories searched by the compiler for include files:

G++ directories (when compiling C++ code only)		<code>\$(gcc_gxx_include_dir)</code>
		<code>\$(gcc_gxx_include_dir)/\$(target)</code> <code>[/\$(multi)]</code>
		<code>\$(gcc_gxx_include_dir)/backward</code>
Prefix directories (if distinct from system directories)	[native only]	<code>\$(prefix)/include</code>
GCC directories		<code>\$(libdir)/gcc/\$(target)/\$(version)/include</code>
		<code>\$(libdir)/gcc/\$(target)/\$(version)/include-fixed</code>
Toolchain directories	[cross only]	<code>\$(tooldir)/sys-include</code>
		<code>\$(tooldir)/include</code>
System directories	[native/sysroot]	<code>\$(sysroot)/usr/local/include</code>
	[native/sysroot]	<code>\$(sysroot)/usr/include</code>

Directories searched by the compiler driver for startup files (crt\*.o):

GCC		\$(libdir)/gcc/\${target}/\${version}
directories		[\$(multi)]
Toolchain		\$(tooldir)/lib/\${target}/\${version}
directories		[\$(multi_os)]
		\$(tooldir)/lib[\$(multi_os)]
Prefix	[native only]	\$(libdir)/\${target}/\${version}
directories		[\$(multi_os)]
		[native only] \$(libdir)[\$(multi_os)]
System	[native/sysroot]	\$(sysroot)/lib/\${target}/\${version}
directories		[\$(multi_os)]
	[native/sysroot]	\$(sysroot)/lib[\$(multi_os)]
	[native/sysroot]	\$(sysroot)/usr/lib/\${target}/\${version}
		[\$(multi_os)]
	[native/sysroot]	\$(sysroot)/usr/lib[\$(multi_os)]

Directories searched by the linker for libraries:

Prefix	[native only]	\$(libdir)\$(biarch_suffix)
directories (if distinct from system directories)		
Toolchain	[native/cross]	\$(tooldir)/lib\$(biarch_suffix)
directories		
System	[native/sysroot]	\$(sysroot)/usr/local/lib\$(biarch_suffix)
directories	[native/sysroot]	\$(sysroot)/usr/lib\$(biarch_suffix)
	[native/sysroot]	\$(sysroot)/lib\$(biarch_suffix)
Non-biarch	[native only]	\$(libdir)
directories (if	[native/cross]	\$(tooldir)/lib
distinct from	[native/sysroot]	\$(sysroot)/usr/local/lib
the above)	[native/sysroot]	\$(sysroot)/usr/lib
	[native/sysroot]	\$(sysroot)/lib

Note: In addition to these directories built-in to the linker, if the linker is invoked via the compiler driver, it will also search the same list of directories specified above for startup files, because those are implicitly passed in via -L options by the driver. Also, when searching for dependencies of shared libraries, the linker will attempt to mimic the search order used by the dynamic linker, including DT\_RPATH/DT\_RUNPATH and LD\_LIBRARY\_PATH lookups.

## Multiarch impact on the toolchain

The current multiarch proposal is to move the system library directories to a new path including the GNU target triplet, that is, instead of using

`/lib`

`/usr/lib`

the system library directories are now called

`/lib/${multiarch}`

`/usr/lib/${multiarch}`

At this point, there is no provision for multiarch executable or header file installation.

What are the effects of this renaming on the toolchain, following the discussion above?

### ELF interpreter

The ELF interpreter would now reside in a new location, e.g.

`/lib/${multiarch}/ld-linux.so.2`

This allows interpreters for different architectures to be installed simultaneously, and removes the need for the various bi-arch hacks. Change would imply modification of the GCC back-end, and possibly the binutils `ld` default as well (even though that's currently not normally used), to build new executables using the new ELF interpreter install path.

Caveats:

- Any executable built with the new ELF interpreter will absolutely not run on a system that does not provide the multiarch install location of the interpreter. (This is probably OK.)
- Executables built with the old ELF interpreter will not run on a system that *\*only\** provides the multiarch install location. This is clearly *\*not\** OK. To provide backwards compatibility, even a multiarch-capable system will need to install ELF interpreters at the old locations as well, possibly via symlinks.

(Note that any given system can only be compatible in this way with *\*one\** architecture, except for lucky circumstances.)

- As the multiarch string `$(multiarch)` is now embedded into each and every executable file, it becomes invariant part of the platform ABI, and needs to be strictly standardized. GNU target triplets as used today in general seem to provide too much flexibility and underspecified components to serve in such a role, at least without some additional requirements.

## Shared library search paths

According to the primary multiarch assumption, the system library search paths are modified to include the multiarch target string:

```
/lib/$(multiarch)
/usr/lib/$(multiarch)
```

This requires modifications to glibc's ld.so loader (can possibly be provided via platform back-end changes). Backwards compatibility most likely requires that both the new multiarch location and the old location are searched.

Open questions:

- How are `-rpath` encoded paths to be handled?  
 Option A: They are used by ld.so as-is. This implies that in a fully multiarch system, every *\*user\** of `-rpath` needs to update their paths to include the multiarch target. Also, multiarch targets are once again directly embedded into executables.  
 Option B: ld.so automatically appends the multiarch target string to the path as encoded in `DT_RPATH/DT_RUNPATH`. This may break backwards compatibility.  
 Option C: ld.so searches both the `-rpath` as is, and also with multiarch target string appended.
- How is `LD_LIBRARY_PATH` to be handled?  
 The options here are basically analogous to the `-rpath` case.
- What is the interaction between the multiarch string and capability suffixes (hwcaps etc.) supported by ld.so?  
 The most straightforward option seems to be to just leave the capability mechanism unchanged, that is, ld.so would continue to append the capability suffixes to all directory search paths (which potentially already include a multiarch suffix). This implies that different ABI-compatible but

capability-optimized versions of a library share the same multiarch prefix, but use different capability suffixes.

## GCC and toolchain directory paths

The core multiarch spec only refers to system directories. What about directories provided by GCC and the toolchain? Note that for a cross-development setup, we may have various combinations of host and target architectures. In this case  $\$(\text{host-multiarch})$  refers to the multiarch identifier for the host architecture, while  $\$(\text{target-multiarch})$  refers to the one for the target architecture. In addition  $\$(\text{target})$  refers to the GNU target triplet as currently used by GCC paths (which may or may not be equal to  $\$(\text{target-multiarch})$ , depending on how the latter will end up being standardized).

- GCC private directories

The GCC default installation already distinguishes files that are independent of the host architecture (in `/usr/lib/gcc`) from those that are dependent on the host architecture (in `/usr/libexec/gcc`). In both cases, the target architecture is already explicitly encoded in the path names. Thus it would appear that we'd simply have to move the libexec paths to include the multiarch string in a straightforward manner in order:

```
/usr/lib/gcc/ $\$(\text{target})$ / $\$(\text{version})$ /...
/usr/libexec/ $\$(\text{host-multiarch})$ /gcc/ $\$(\text{target})$ / $\$(\text{version})$ /...
```

This assumes that two cross-compilers to the same target running on different hosts can share `/usr/lib/gcc`, which may not be fully possible (because two cross-compilers may build slightly different versions of target libraries due to optimization differences). In this case, the whole of `/usr/lib/gcc` could be moved to multiarch as well:

```
/usr/lib/ $\$(\text{host-multiarch})$ /gcc/ $\$(\text{target})$ / $\$(\text{version})$ /...
/usr/libexec/ $\$(\text{host-multiarch})$ /gcc/ $\$(\text{target})$ / $\$(\text{version})$ /...
```

The alternative would be to package them separately, with the host independent files always coming from the same package (presumably itself produced on a native system, or else one "master" host system). Note that if -in the first stage of multiarch- we do not support parallel installation of *\*binaries\**, we may not need to do anything for the GCC directories.

- Toolchain directories

The `/usr/ $\$(\text{target})$`  directory used by various toolchain components is somewhat of a mixture of target vs. host dependent files:

<code>/usr/\$(target)/bin</code>	executable files in host architecture, targeting target architecture (e.g. cross-assembler, cross-linker binaries)
<code>/usr/\$(target)/sys-include</code>	target headers (only for cross-builds)
<code>/usr/\$(target)/include</code>	native toolchain header files for target (like <code>bfd.h</code> )
<code>/usr/\$(target)/lib</code>	target libraries + native toolchain libraries for target (like <code>libbfd.a</code> )
<code>/usr/\$(target)/lib/ldscripts</code>	linker scripts used by the cross-linker

In a full multiarch setup, the only directory that would require the multiarch suffix is probably `bin`:

```
/usr/$(target)/bin/$(host-multiarch)
```

As discussed above, if we want to support different versions of target libraries for the same target, as compiled with differently hosted cross-compilers, we might also have to multiarch the `lib` directory:

```
/usr/$(target)/lib/$(host-multiarch)
```

Yet another option might be to require multiarch systems to always use the `sysroot` cross-compile option, and not support the toolchain directory for target libraries in the first place. [Or at least, have no distribution package ever install any target library into the toolchain `lib` directory ...]

- System directories

For these, the primary multiarch rules would apply. In a `sysroot` configuration, the `sysroot` prefix is applied as usual (after the multiarch paths have been determined as for a native system). Note that we need to apply the `*target*` multiarch string here; in case this is different from the target triplet, the toolchain would have to have explicit knowledge of those names:

```
/lib/$(target-multiarch)
/usr/lib/$(target-multiarch)
```

- Prefix directories

For completeness' sake, we need to define how prefix directories are handled in a GCC build that is both multiarch enabled *and* not installed into the system prefix. The most straightforward solution would be to apply multiarch suffixes to the prefix directories as well:

```
$(prefix)/lib/$(target-multiarch)
$(prefix)/include/$(target-multiarch)    [ if include is multiarch'ed ... ]
```

## Multiarch and cross-compilation

Using the paths as discussed in the previous section, we have some options how to perform cross-compilation on a multiarch system. The most obvious option is to build a cross-compiler with sysroot equal to "/". This means that the compiler will use target libraries and header files as installed by unmodified distribution multiarch packages for the target architecture. This should ideally be the default cross-compilation setup on a multi-arch system. In addition, it is still possible to build cross-compilers with a different sysroot, which may be useful if you want to install target libraries you build yourself into a non-system directory (and do not want to require root access for doing so).

Questions:

- Should we still support "traditional" cross-compilation using the toolchain directory to hold target libraries/header files?

This is probably no longer really useful. On the other hand, it probably doesn't really hurt either ...

- What about header files in a multiarch system?

The current multiarch spec does not provide for multiple locations for header files. This works only if headers are identical across all targets. This is usually true, and where it isn't, can be enforced by use of conditional compilation. In fact, the latter is how things currently work on traditional bi-arch distributions. In the alternative, the toolchain could also provide for multiarch'ed header directories along the lines of

```
/usr/include/local/${target-multiarch}
/usr/include/${target-multiarch}
```

which are included in addition to (and before) the usual directories. It will most likely not be necessary to do this for the GCC and toolchain directory include paths, as they are already target-specific.

## Multiarch and multilib

The multilib mechanism provides a way to support multiple incompatible ABI versions on the same ISA. In a multiarch world, this is supposed to be handled by different multiarch prefixes, to enable use of the package management system to handle libraries for all those variants. How can we reconcile the two systems?

It would appear that the way forward is based on the existing "OS multilib suffix" mechanism. GCC already expects to need to handle naming conventions provided by the OS for where incompatible versions are to found. In a multiarch system, the straightforward solution would appear to be to use the multiarch names as-is as OS multilib suffixes.

In fact, this could even handle the *\*default\** multiarch name without requiring any further changes to GCC. For example, in a bi-arch amd64 setup, the GCC back-end might register "x86\_64-linux" as default OS multilib suffix, and "i386-linux" as OS multilib suffix triggered by the (ABI changing) -m32 option. Without any further change, GCC would now search

```
/usr/lib/x86_64-linux
```

or

```
/usr/lib/i386-linux
```

as appropriate, depending on the command line options used. (This assumes that the main libdir is /usr/lib, i.e. no bi-arch configure options were used.)

Note that GCC would still use its own internal multilib suffixes for its private libraries, but that seems to be OK.

Caveat: This would imply those multilib names, and their association with compiler options, becomes hard-coded in the GCC back-end. However, this seems to have already been necessary (see above).

## Summary

From the preceding discussion, it would appear a multiarch setup allowing parallel installation of run-time libraries and development packages, and thus providing support for running binaries of different native ISAs and ABI variants as well as cross-compilation, might be feasible by implementing the following set of changes. Note that parallel installation of main *\*executable\** files by multiarch packages is not (yet) supported.

- Define well-known multiarch suffix names for each supported ISA/ABI combination. Well-known in particular means it is allowed for them to be



hard-coded in toolchain source files, as well as in executable files and libraries built by such a toolchain.

- Change GCC back-ends to use the well-known multiarch suffix as OS multilib suffix, depending on target and ABI-changing options. Also include multiarch suffix in ELF interpreter name passed to ld. (Probably need to change ld default directory search paths as well.)
- Change the dynamic loader ld.so to optionally append the multiarch suffix (as a constant string pre-determined at ld.so build time) after each directory search path, before further appending capability suffixes. (See above as to open questions about -rpath and LD\_LIBRARY\_PATH.)
- Change package build/install rules to install libraries and ld.so into multiarch library directories (not covered in this document). Change system installation to provide for backward-compatibility fallbacks (e.g. symbolic links to the ELF interpreter).
- If capability-optimized ISA/ABI-compatible library variants are desired, they can be built just as today, only under the (same) multiarch suffix. They could be packaged either within a single package, or else using multiple packages (of the same multiarch type).
- Enforce platform-independent header files across all packages. (In the alternative, provide for multiarch include paths in GCC.)
- Build cross-compiler packages with --with-sysroot=

This document is primarily the work of Ulrich Wiegand, and came out of a discussion at the Linaro sprint in Prague, July 2010.

I'd appreciate any feedback or comments! Have I missed something?