Adding a dependency

The quickest and most robust way to solve a problem is often to incorporate an existing solution. In project terms - why implement a new feature if we can find an implementation and just integrate it into ours? The answer, usually, is that there's no reason - integrating externally-available functionality is at the heart of much of what we've done in sipXecs, but there are some rules and some processes to be followed to keep it from being disruptive to others and/or harmful to the project.

The following outlines the steps you should take when incorporating an external dependency to ensure that you don't disrupt the work of other developers or the construction of release builds and ISO images.

Distribution Platform Availability

Make sure that the required version of the dependency is available for all of our current distribution platforms. If it is not, then we will have to build and distribute it, which significantly adds to the work that **you** and others will need to do.

Component Organization

The top level directory when you check out sipXecs has directories for *components*, along with a few files that control how those components relate to each other. Mostly, when adding a dependency, you're going to be either adding something to be used by some component or adding an entire standalone component.

Autoconf test for dependency

The traditional instructions for how to build and open source project from source are to

1. download the source tarball
2. unpack it
3. cd into the directory it creates
4. run `configure`
5. run `make all install`

We'd like those instructions to work for sipXecs (a goal we have not really achieved, since to begin with we don't create a single source tarball for everything). We would also would like it to be true that it's easy to get started one step earlier by checking out from the source control system. In order to achieve this, we need to automate as much as possible of the checking for dependencies. We use the GNU build system (see box at right) as our top level build framework, so if you add a dependency, you *must* add checks to the the files that control the build.

This wiki page is not going to attempt to do a full tutorial on autoconf or the other tools we use - for more complete documentation, see the GNU Build System.

`configure.ac` and the `config` directory

The `configure.ac` file at the top of each component, and the `config` directory for the project (which is soft-linked into each component) are the inputs that control how the source is prepared for building on any given platform. If the component has an external dependency, the generated `configure` script should test for whether or not that dependency is available.

Some packages define autoconf test macros and install them in `/usr/share/aclocal`; check first there - autoconf will find those macro definitions automatically, so you can just reference them directly (the fact that a macro definition is missing will cause the `autoconf` invocation to fail, which will indicate that the package is not installed, or if you want to be friendy you can copy that file and check it into the `config` directory to ensure that it is there).

If there is no pre-packaged test for your dependency, you need to write one. You should create it in a file `config/`*`component`*`.m4` then include that file in your configure.ac.

Here is a simple test:

```
# ================ NET SNMP ================
AC_DEFUN([CHECK_FOO],
[
    AC_MSG_CHECKING([for foo includes ])
    include_path="$includedir $prefix/include /usr/include /usr/local/include"
    include_check="foo/foo.h"

    foundpath=""
    for dir in $include_path ; do
        if test -f "$dir/$include_check";
        then
            foundpath=$dir;
            break;
        fi;
    done
    if test x_$foundpath = x_; then
       AC_MSG_ERROR('$include_check' not found)
    else
       AC_MSG_RESULT($foundpath/$include_check)
    fi
])
```

This code defines the `CHECK_FOO` test macro; it is written in the **m4** macro language, which when processed generates a shell script code fragment that is incorporated into the `configure` script.

The `AC_MSG_CHECKING` macro causes `configure` to print a progress message about what it is looking for - this message does not get a newline at the end.

The `for` loop checks in a series of directories for the `foo/foo.h` include file. The search list should be chosen to look in all the places the target file might be on different platforms, including those installed from rpm (or equivalent) packages or from source. In this case, all the choices were ones that will be in the default search path for include files, so no manipulation of the path is needed. There are a number of examples in `config/general.m4` that do more elaborate searches and set variables used by other parts of the build.

The `AC_MSG_RESULT` macro causes `configure` to print a success message about where it found the file.

The `AC_MSG_ERROR` macro causes `configure` to print a failure message, and aborts `configure`.

Once you have the test macro defined, you just need to add an invocation of it to the `configure.ac` file at the top of any component that requires it:

```
CHECK_FOO
```

RPM Spec file check for dependency

In the top level directory for each sipXecs component, there is a

```
_component_.spec.in
```

file; this file is processed as part of building the source RPM (essentially a tar file that contains the spec file and some metadata); it is the control file for the `rpmbuild` command that constructs the RPM(s) for the component.

In the spec file, you must add a line for any external package that the component requires at build time, and one for any component that it needs at run time.

To declare a build time dependency, add:

```
BuildRequires: foo
```

To declare an install time dependency, add:

```
Requires: foo
```

In either case, the name is the name of the RPM package that provides whatever it is the component depends on. If a particular version is needed, the spec file can declare that as well (usually the >= relation is the one you want):

```
BuildRequires: foo >= 2.0
Requires: foo >= 2.0
```

There are exceptions - for example, if all that's needed is a library that is linked (either statically or dynamically) into an executable installed by the component RPM, then you don't need to declare that - the build system detects and includes those dependencies automatically. However, it never hurts to add them manually, especially if a particular version is needed.