Design document for the unified scheme data

How are things connected?

The unified scheme takes all its data from the `build.info` files seen throughout the source tree. These files hold the minimum information needed to build end product files from diverse sources. See the section on `build.info` files below.

From the information in `build.info` files, `Configure` builds up an information database as a hash table called `%unified_info`, which is stored in configdata.pm, found at the top of the build tree (which may or may not be the same as the source tree).

`Configurations/common.tmpl` uses the data from `%unified_info` to generate the rules for building end product files as well as intermediary files with the help of a few functions found in the build-file templates. See the section on build-file templates further down for more information.

build.info files

As mentioned earlier, `build.info` files are meant to hold the minimum information needed to build output files, and therefore only (with a few possible exceptions [1]) have information about end products (such as scripts, library files and programs) and source files (such as C files, C header files, assembler files, etc). Intermediate files such as object files are rarely directly referred to in `build.info` files (and when they are, it's always with the file name extension `.o`), they are inferred by `Configure`. By the same rule of minimalism, end product file name extensions (such as `.so`, `.a`, `.exe`, etc) are never mentioned in `build.info`. Their file name extensions will be inferred by the build-file templates, adapted for the platform they are meant for (see sections on `%unified_info` and build-file templates further down).

The variables `PROGRAMS`, `LIBS`, `MODULES` and `SCRIPTS` are used to declare end products. There are variants for them with `_NO_INST` as suffix (`PROGRAM_NO_INST` etc) to specify end products that shouldn't get installed.

The variables `SOURCE`, `DEPEND`, `INCLUDE` and `DEFINE` are indexed by a produced file, and their values are the source used to produce that particular produced file, extra dependencies, include directories needed, or C macros to be defined.

All their values in all the `build.info` throughout the source tree are collected together and form a set of programs, libraries, modules and scripts to be produced, source files, dependencies, etc etc etc.

Let's have a pretend example, a very limited contraption of OpenSSL, composed of the program `apps/openssl`, the libraries `libssl` and `libcrypto`, an module `engines/ossltest` and their sources and dependencies.

```
# build.info
LIBS=libcrypto libssl
```

```
INCLUDE[libcrypto]=include
INCLUDE[libssl]=include
DEPEND[libssl]=libcrypto
```

This is the top directory `build.info` file, and it tells us that two libraries are to be built, the include directory `include/` shall be used throughout when building anything that will end up in each library, and that the library `libssl` depend on the library `libcrypto` to function properly.

```
# apps/build.info
PROGRAMS=openssl
SOURCE[openssl]=openssl.c
INCLUDE[openssl]=.. ../include
DEPEND[openssl]=../libssl
```

This is the `build.info` file in `apps/`, one may notice that all file paths mentioned are relative to the directory the `build.info` file is located in. This one tells us that there's a program to be built called `apps/openss` (the file name extension will depend on the platform and is therefore not mentioned in the `build.info` file). It's built from one source file, `apps/openssl.c`, and building it requires the use of `.` and `include/` include directories (both are declared from the point of view of the `apps/` directory), and that the program depends on the library `libssl` to function properly.

```
# crypto/build.info
LIBS=../libcrypto
SOURCE[../libcrypto]=aes.c evp.c cversion.c
DEPEND[cversion.o]=buildinf.h

GENERATE[buildinf.h]=../util/mkbuildinf.pl "$(CC) $(CFLAGS)" "$(PLATFORM)"
DEPEND[buildinf.h]=../Makefile
DEPEND[../util/mkbuildinf.pl]=../util/Foo.pm
```

This is the `build.info` file in `crypto/`, and it tells us a little more about what's needed to produce `libcrypto`. LIBS is used again to declare that `libcrypto` is to be produced. This declaration is really unnecessary as it's already mentioned in the top `build.info` file, but can make the info file easier to understand. This is to show that duplicate information isn't an issue.

This `build.info` file informs us that `libcrypto` is built from a few source files, `crypto/aes.c`, `crypto/evp.c` and `crypto/cversion.c`. It also shows us that building the object file inferred from `crypto/cversion.c` depends on `crypto/buildinf.h`. Finally, it also shows the possibility to declare how some files are generated using some script, in this case a perl script, and how such scripts can be declared to depend on other files, in this case a perl module.

Two things are worth an extra note:

`DEPEND[cversion.o]` mentions an object file. DEPEND indexes is the only location where it's valid to mention them

```
# ssl/build.info
LIBS=../libssl
SOURCE[../libssl]=tls.c
```

This is the build.info file in `ssl/`, and it tells us that the library `libssl` is built from the source file `ssl/tls.c`.

```
# engines/build.info
MODULES=dasync
SOURCE[dasync]=e_dasync.c
DEPEND[dasync]=../libcrypto
INCLUDE[dasync]=../include

MODULES_NO_INST=ossltest
SOURCE[ossltest]=e_ossltest.c
DEPEND[ossltest]=../libcrypto.a
INCLUDE[ossltest]=../include
```

This is the `build.info` file in `engines/`, telling us that two modules called `engines/dasync` and `engines/ossltest` shall be built, that `dasync`'s source is `engines/e_dasync.c` and `ossltest`'s source is `engines/e_ossltest.c` and that the include directory `include/` may be used when building anything that will be part of these modules. Also, both modules depend on the library `libcrypto` to function properly. `ossltest` is explicitly linked with the static variant of the library `libcrypto`. Finally, only `dasync` is being installed, as `ossltest` is only for internal testing.

When `Configure` digests these `build.info` files, the accumulated information comes down to this:

```
LIBS=libcrypto libssl
SOURCE[libcrypto]=crypto/aes.c crypto/evp.c crypto/cversion.c
DEPEND[crypto/cversion.o]=crypto/buildinf.h
INCLUDE[libcrypto]=include
SOURCE[libssl]=ssl/tls.c
INCLUDE[libssl]=include
DEPEND[libssl]=libcrypto

PROGRAMS=apps/openssl
SOURCE[apps/openssl]=apps/openssl.c
INCLUDE[apps/openssl]=. include
DEPEND[apps/openssl]=libssl

MODULES=engines/dasync
SOURCE[engines/dasync]=engines/e_dasync.c
DEPEND[engines/dasync]=libcrypto
INCLUDE[engines/dasync]=include
```

```
MODULES_NO_INST=engines/ossltest
SOURCE[engines/ossltest]=engines/e_ossltest.c
DEPEND[engines/ossltest]=libcrypto.a
INCLUDE[engines/ossltest]=include

GENERATE[crypto/buildinf.h]=util/mkbuildinf.pl "$(CC) $(CFLAGS)" "$(PLATFORM)"
DEPEND[crypto/buildinf.h]=Makefile
DEPEND[util/mkbuildinf.pl]=util/Foo.pm
```

A few notes worth mentioning:

LIBS may be used to declare routine libraries only.

PROGRAMS may be used to declare programs only.

MODULES may be used to declare modules only.

The indexes for SOURCE must only be end product files, such as libraries, programs or modules. The values of SOURCE variables must only be source files (possibly generated).

INCLUDE and DEPEND shows a relationship between different files (usually produced files) or between files and directories, such as a program depending on a library, or between an object file and some extra source file.

When Configure processes the build.info files, it will take it as truth without question, and will therefore perform very few checks. If the build tree is separate from the source tree, it will assume that all built files and up in the build directory and that all source files are to be found in the source tree, if they can be found there. Configure will assume that source files that can't be found in the source tree (such as crypto/bildinf.h in the example above) are generated and will be found in the build tree.

The %unified_info database

The information in all the build.info get digested by Configure and collected into the %unified_info database, divided into the following indexes:

```
depends   => a hash table containing 'file' => [ 'dependency' ... ]
             pairs.  These are directly inferred from the DEPEND
             variables in build.info files.

modules   => a list of modules.  These are directly inferred from
             the MODULES variable in build.info files.

generate  => a hash table containing 'file' => [ 'generator' ... ]
             pairs.  These are directly inferred from the GENERATE
             variables in build.info files.

includes  => a hash table containing 'file' => [ 'include' ... ]
```

```
             pairs.  These are directly inferred from the INCLUDE
             variables in build.info files.

install   => a hash table containing 'type' => [ 'file' ... ] pairs.
             The types are 'programs', 'libraries', 'modules' and
             'scripts', and the array of files list the files of
             that type that should be installed.

libraries => a list of libraries.  These are directly inferred from
             the LIBS variable in build.info files.

programs  => a list of programs.  These are directly inferred from
             the PROGRAMS variable in build.info files.

scripts   => a list of scripts.  There are directly inferred from
             the SCRIPTS variable in build.info files.

sources   => a hash table containing 'file' => [ 'sourcefile' ... ]
             pairs.  These are indirectly inferred from the SOURCE
             variables in build.info files.  Object files are
             mentioned in this hash table, with source files from
             SOURCE variables, and AS source files for programs and
             libraries.

shared_sources =>
             a hash table just like 'sources', but only as source
             files (object files) for building shared libraries.
```

As an example, here is how the `build.info` files example from the section above would be digested into a `%unified_info` table:

```
our %unified_info = (
    "depends" =>
        {
            "apps/openssl" =>
                [
                    "libssl",
                ],
            "crypto/buildinf.h" =>
                [
                    "Makefile",
                ],
            "crypto/cversion.o" =>
                [
                    "crypto/buildinf.h",
                ],
            "engines/dasync" =>
```

```
                [
                    "libcrypto",
                ],
            "engines/ossltest" =>
                [
                    "libcrypto.a",
                ],
            "libssl" =>
                [
                    "libcrypto",
                ],
            "util/mkbuildinf.pl" =>
                [
                    "util/Foo.pm",
                ],
        },
    "modules" =>
        [
            "engines/dasync",
            "engines/ossltest",
        ],
    "generate" =>
        {
            "crypto/buildinf.h" =>
                [
                    "util/mkbuildinf.pl",
                    "\"\$(CC)",
                    "\$(CFLAGS)\"",
                    "\"$(PLATFORM)\"",
                ],
        },
    "includes" =>
        {
            "apps/openssl" =>
                [
                    ".",
                    "include",
                ],
            "engines/ossltest" =>
                [
                    "include"
                ],
            "libcrypto" =>
                [
                    "include",
                ],
            "libssl" =>
                [
                    "include",
                ],
            "util/mkbuildinf.pl" =>
                [
                    "util",
                ],
        }
    "install" =>
        {
            "modules" =>
                [
                    "engines/dasync",
                ],
            "libraries" =>
                [
                    "libcrypto",
                    "libssl",
                ],
            "programs" =>
                [
                    "apps/openssl",
                ],
       },
    "libraries" =>
        [
            "libcrypto",
            "libssl",
        ],
    "programs" =>
        [
            "apps/openssl",
        ],
    "sources" =>
        {
            "apps/openssl" =>
                [
                    "apps/openssl.o",
                ],
            "apps/openssl.o" =>
                [
                    "apps/openssl.c",
                ],
            "crypto/aes.o" =>
                [
                    "crypto/aes.c",
                ],
            "crypto/cversion.o" =>
                [
                    "crypto/cversion.c",
                ],
            "crypto/evp.o" =>
                [
                    "crypto/evp.c",
                ],
            "engines/e_dasync.o" =>
                [
                    "engines/e_dasync.c",
                ],
            "engines/dasync" =>
                [
                    "engines/e_dasync.o",
                ],
            "engines/e_ossltest.o" =>
                [
                    "engines/e_ossltest.c",
                ],
            "engines/ossltest" =>
                [
                    "engines/e_ossltest.o",
                ],
            "libcrypto" =>
                [
                    "crypto/aes.c",
                    "crypto/cversion.c",
                    "crypto/evp.c",
                ],
            "libssl" =>
                [
                    "ssl/tls.c",
                ],
            "ssl/tls.o" =>
                [
                    "ssl/tls.c",
                ],
        },
);
```

As can be seen, everything in `%unified_info` is fairly simple suggest of information. Still, it tells us that to build all programs, we must build `apps/openssl`, and to build the latter, we will need to build all its sources (`apps/openssl.o` in this case) and all the other things it depends on (such as `libssl`). All those dependencies need to be built as well, using the same logic, so to build `libssl`, we need to build `ssl/tls.o` as well as `libcrypto`, and to build the latter…

Build-file templates

Build-file templates are essentially build-files (such as `Makefile` on Unix) with perl code fragments mixed in. Those perl code fragment will generate all the configuration dependent data, including all the rules needed to build end product files and intermediary files alike. At a minimum, there must be a perl code fragment that defines a set of functions that are used to generates specific build-file rules, to build static libraries from object files, to build shared libraries from static libraries, to programs from object files and libraries, etc.

```
generatesrc - function that produces build file lines to generate
              a source file from some input.

              It's called like this:

                    generatesrc(src => "PATH/TO/tobegenerated",
                                generator => [ "generatingfile", ... ]
                                generator_incs => [ "INCL/PATH", ... ]
                                generator_deps => [ "dep1", ... ]
                                incs => [ "INCL/PATH", ... ],
                                deps => [ "dep1", ... ],
                                intent => one of "libs", "dso", "bin" );

              'src' has the name of the file to be generated.
              'generator' is the command or part of command to
              generate the file, of which the first item is
              expected to be the file to generate from.
              generatesrc() is expected to analyse and figure out
              exactly how to apply that file and how to capture
              the result.  'generator_incs' and 'generator_deps'
              are include directories and files that the generator
              file itself depends on.  'incs' and 'deps' are
              include directories and files that are used if $(CC)
              is used as an intermediary step when generating the
              end product (the file indicated by 'src').  'intent'
              indicates what the generated file is going to be
              used for.

src2obj     - function that produces build file lines to build an
              object file from source files and associated data.

              It's called like this:
                    src2obj(obj => "PATH/TO/objectfile",
                            srcs => [ "PATH/TO/sourcefile", ... ],
                            deps => [ "dep1", ... ],
                            incs => [ "INCL/PATH", ... ]
                            intent => one of "lib", "dso", "bin" );

              'obj' has the intended object file with `.o`
              extension, src2obj() is expected to change it to
              something more suitable for the platform.
              'srcs' has the list of source files to build the
              object file, with the first item being the source
              file that directly corresponds to the object file.
              'deps' is a list of explicit dependencies.  'incs'
              is a list of include file directories.  Finally,
              'intent' indicates what this object file is going
              to be used for.

obj2lib     - function that produces build file lines to build a
              static library file ("libfoo.a" in Unix terms) from
              object files.

              called like this:

                    obj2lib(lib => "PATH/TO/libfile",
                            objs => [ "PATH/TO/objectfile", ... ]);

              'lib' has the intended library file name *without*
              extension, obj2lib is expected to add that.  'objs'
              has the list of object files to build this library.

libobj2shlib - backward compatibility function that's used the
              same way as obj2shlib (described next), and was
              expected to build the shared library from the
              corresponding static library when that was suitable.
              NOTE: building a shared library from a static
              library is now DEPRECATED, as they no longer share
              object files.  Attempting to do this will fail.

obj2shlib   - function that produces build file lines to build a
              shareable object library file ("libfoo.so" in Unix
              terms) from the corresponding object files.

              called like this:

                    obj2shlib(shlib => "PATH/TO/shlibfile",
                              lib => "PATH/TO/libfile",
                              objs => [ "PATH/TO/objectfile", ... ],
                              deps => [ "PATH/TO/otherlibfile", ... ]);

              'lib' has the base (static) library file name
              *without* extension.  This is useful in case
              supporting files are needed (such as import
              libraries on Windows).
              'shlib' has the corresponding shared library name
              *without* extension.  'deps' has the list of other
              libraries (also *without* extension) this library
              needs to be linked with.  'objs' has the list of
              object files to build this library.

obj2dso     - function that produces build file lines to build a
              dynamic shared object file from object files.

              called like this:

                    obj2dso(lib => "PATH/TO/libfile",
                            objs => [ "PATH/TO/objectfile", ... ],
                            deps => [ "PATH/TO/otherlibfile",
                            ... ]);

              This is almost the same as obj2shlib, but the
              intent is to build a shareable library that can be
              loaded in runtime (a "plugin"...).

obj2bin     - function that produces build file lines to build an
              executable file from object files.

              called like this:

                    obj2bin(bin => "PATH/TO/binfile",
                            objs => [ "PATH/TO/objectfile", ... ],
                            deps => [ "PATH/TO/libfile", ... ]);

              'bin' has the intended executable file name
              *without* extension, obj2bin is expected to add
              that.  'objs' has the list of object files to build
              this library.  'deps' has the list of library files
              (also *without* extension) that the programs needs
              to be linked with.

in2script   - function that produces build file lines to build a
              script file from some input.
              called like this:

                    in2script(script => "PATH/TO/scriptfile",
                              sources => [ "PATH/TO/infile", ... ]);

              'script' has the intended script file name.
              'sources' has the list of source files to build the
              resulting script from.
```

Along with the build-file templates is the driving template `Configurations/common.tmpl`, which looks through all the information in `%unified_info` and generates all the rulesets to build libraries, programs and all intermediate files, using the rule generating functions defined in the build-file template.

As an example with the smaller `build.info` set we've seen as an example, producing the rules to build `libcrypto` would result in the following calls:

```
# Note: obj2shlib will only be called if shared libraries are
# to be produced.
# Note 2: obj2shlib must convert the '.o' extension to whatever
# is suitable on the local platform.
obj2shlib(shlib => "libcrypto",
          objs => [ "crypto/aes.o", "crypto/evp.o", "crypto/cversion.o" ],
          deps => [  ]);

obj2lib(lib => "libcrypto"
        objs => [ "crypto/aes.o", "crypto/evp.o", "crypto/cversion.o" ]);

src2obj(obj => "crypto/aes.o"
        srcs => [ "crypto/aes.c" ],
        deps => [ ],
        incs => [ "include" ],
        intent => "lib");

src2obj(obj => "crypto/evp.o"
        srcs => [ "crypto/evp.c" ],
        deps => [ ],
        incs => [ "include" ],
        intent => "lib");

src2obj(obj => "crypto/cversion.o"
        srcs => [ "crypto/cversion.c" ],
        deps => [ "crypto/buildinf.h" ],
        incs => [ "include" ],
        intent => "lib");
```

```
generatesrc(src => "crypto/buildinf.h",
            generator => [ "util/mkbuildinf.pl", "\"$(CC)",
                           "$(CFLAGS)\"", "\"$(PLATFORM)\"" ],
            generator_incs => [ "util" ],
            generator_deps => [ "util/Foo.pm" ],
            incs => [ ],
            deps => [ ],
            intent => "lib");
```

The returned strings from all those calls are then concatenated together and written to the resulting build-file.