Intro

This directory contains a few sets of files that are used for configuration in diverse ways:

```
*.conf      Target platform configurations, please read
            'Configurations of OpenSSL target platforms' for more
            information.
*.tmpl      Build file templates, please read 'Build-file
            programming with the "unified" build system' as well
            as 'Build info files' for more information.
*.pm        Helper scripts / modules for the main `Configure`
            script.  See 'Configure helper scripts for more
            information.
```

Configurations of OpenSSL target platforms

Configuration targets are a collection of facts that we know about different platforms and their capabilities. We organise them in a hash table, where each entry represent a specific target.

Note that configuration target names must be unique across all config files. The Configure script does check that a config file doesn't have config targets that shadow config targets from other files.

In each table entry, the following keys are significant:

```
    inherit_from    => Other targets to inherit values from.
                       Explained further below. [1]
    template        => Set to 1 if this isn't really a platform
                       target.  Instead, this target is a template
                       upon which other targets can be built.
                       Explained further below.  [1]

    sys_id          => System identity for systems where that
                       is difficult to determine automatically.

    enable          => Enable specific configuration features.
                       This MUST be an array of words.
    disable         => Disable specific configuration features.
                       This MUST be an array of words.
                       Note: if the same feature is both enabled
                       and disabled, disable wins.

    as              => The assembler command.  This is not always
                       used (for example on Unix, where the C
                       compiler is used instead).
```

```
asflags         => Default assembler command flags [4].
cpp             => The C preprocessor command, normally not
                   given, as the build file defaults are
                   usually good enough.
cppflags        => Default C preprocessor flags [4].
defines         => As an alternative, macro definitions may be
                   given here instead of in 'cppflags' [4].
                   If given here, they MUST be as an array of
                   the string such as "MACRO=value", or just
                   "MACRO" for definitions without value.
includes        => As an alternative, inclusion directories
                   may be given here instead of in 'cppflags'
                   [4].  If given here, the MUST be an array
                   of strings, one directory specification
                   each.
cc              => The C compiler command, usually one of "cc",
                   "gcc" or "clang".  This command is normally
                   also used to link object files and
                   libraries into the final program.
cxx             => The C++ compiler command, usually one of
                   "c++", "g++" or "clang++".  This command is
                   also used when linking a program where at
                   least one of the object file is made from
                   C++ source.
cflags          => Defaults C compiler flags [4].
cxxflags        => Default  C++ compiler flags [4].  If unset,
                   it gets the same value as cflags.

(linking is a complex thing, see [3] below)
ld              => Linker command, usually not defined
                   (meaning the compiler command is used
                   instead).
                   (NOTE: this is here for future use, it's
                   not implemented yet)
lflags          => Default flags used when linking apps,
                   shared libraries or DSOs [4].
ex_libs         => Extra libraries that are needed when
                   linking shared libraries, DSOs or programs.
                   The value is also assigned to Libs.private
                   in $(libdir)/pkgconfig/libcrypto.pc.

shared_cppflags => Extra C preprocessor flags used when
                   processing C files for shared libraries.
shared_cflag    => Extra C compiler flags used when compiling
                   for shared libraries, typically something
                   like "-fPIC".
```

```
shared_ldflag   => Extra linking flags used when linking
                   shared libraries.
module_cppflags
module_cflags
module_ldflags  => Has the same function as the corresponding
                   'shared_' attributes, but for building DSOs.
                   When unset, they get the same values as the
                   corresponding 'shared_' attributes.

ar              => The library archive command, the default is
                   "ar".
                   (NOTE: this is here for future use, it's
                   not implemented yet)
arflags         => Flags to be used with the library archive
                   command.  On Unix, this includes the
                   command letter, 'r' by default.

ranlib          => The library archive indexing command, the
                   default is 'ranlib' it it exists.

unistd          => An alternative header to the typical
                   '<unistd.h>'.  This is very rarely needed.

shared_extension => File name extension used for shared
                    libraries.
obj_extension   => File name extension used for object files.
                   On unix, this defaults to ".o" (NOTE: this
                   is here for future use, it's not
                   implemented yet)
exe_extension   => File name extension used for executable
                   files.  On unix, this defaults to "" (NOTE:
                   this is here for future use, it's not
                   implemented yet)
shlib_variant   => A "variant" identifier inserted between the base
                   shared library name and the extension.  On "unixy"
                   platforms (BSD, Linux, Solaris, MacOS/X, ...) this
                   supports installation of custom OpenSSL libraries
                   that don't conflict with other builds of OpenSSL
                   installed on the system.  The variant identifier
                   becomes part of the SONAME of the library and also
                   any symbol versions (symbol versions are not used or
                   needed with MacOS/X).  For example, on a system
                   where a default build would normally create the SSL
                   shared library as 'libssl.so -> libssl.so.1.1' with
                   the value of the symlink as the SONAME, a target
                   definition that sets 'shlib_variant => "-abc"' will
                   create 'libssl.so -> libssl-abc.so.1.1', again with
                   an SONAME equal to the value of the symlink.  The
                   symbol versions associated with the variant library
                   would then be 'OPENSSL_ABC_<version>' rather than
                   the default 'OPENSSL_<version>'. The string inserted
                   into symbol versions is obtained by mapping all
                   letters in the "variant" identifier to upper case
                   and all non-alphanumeric characters to '_'.

thread_scheme   => The type of threads is used on the
                   configured platform.  Currently known
                   values are "(unknown)", "pthreads",
                   "uithreads" (a.k.a solaris threads) and
                   "winthreads".  Except for "(unknown)", the
                   actual value is currently ignored but may
                   be used in the future.  See further notes
                   below [2].
dso_scheme      => The type of dynamic shared objects to build
                   for.  This mostly comes into play with
                   modules, but can be used for other purposes
                   as well.  Valid values are "DLFCN"
                   (dlopen() et al), "DLFCN_NO_H" (for systems
                   that use dlopen() et al but do not have
                   fcntl.h), "DL" (shl_load() et al), "WIN32"
                   and "VMS".
asm_arch        => The architecture to be used for compiling assembly
                   source.  This acts as a selector in build.info files.
uplink_arch     => The architecture to be used for compiling uplink
                   source.  This acts as a selector in build.info files.
                   This is separate from asm_arch because it's compiled
                   even when 'no-asm' is given, even though it contains
                   assembler source.
perlasm_scheme  => The perlasm method used to create the
                   assembler files used when compiling with
                   assembler implementations.
shared_target   => The shared library building method used.
                   This serves multiple purposes:
                   - as index for targets found in shared_info.pl.
                   - as linker script generation selector.
                   To serve both purposes, the index for shared_info.pl
                   should end with '-shared', and this suffix will be
                   removed for use as a linker script generation
                   selector.  Note that the latter is only used if
                   'shared_defflag' is defined.
build_scheme    => The scheme used to build up a Makefile.
                   In its simplest form, the value is a string
                   with the name of the build scheme.
                   The value may also take the form of a list
                   of strings, if the build_scheme is to have
                   some options.  In this case, the first
                   string in the list is the name of the build
                   scheme.
                   Currently recognised build scheme is "unified".
                   For the "unified" build scheme, this item
                   *must* be an array with the first being the
                   word "unified" and the second being a word
                   to identify the platform family.

multilib        => On systems that support having multiple
                   implementations of a library (typically a
                   32-bit and a 64-bit variant), this is used
                   to have the different variants in different
                   directories.

bn_ops          => Building options (was just bignum options in
                   the earlier history of this option, hence the
                   name). This is a string of words that describe
                   algorithms' implementation parameters that
                   are optimal for the designated target platform,
                   such as the type of integers used to build up
                   the bignum, different ways to implement certain
                   ciphers and so on. To fully comprehend the
                   meaning, the best is to read the affected
                   source.
                   The valid words are:

                   THIRTY_TWO_BIT       bignum limbs are 32 bits,
                                        this is default if no
                                        option is specified, it
                                        works on any supported
                                        system [unless "wider"
                                        limb size is implied in
                                        assembly code];
                   BN_LLONG             bignum limbs are 32 bits,
                                        but 64-bit 'unsigned long
                                        long' is used internally
                                        in calculations;
                   SIXTY_FOUR_BIT_LONG  bignum limbs are 64 bits
                                        and sizeof(long) is 8;
                   SIXTY_FOUR_BIT       bignums limbs are 64 bits,
                                        but execution environment
                                        is ILP32;
```

```
                       RC4_CHAR             RC4 key schedule is made
                                            up of 'unsigned char's;
                       RC4_INT              RC4 key schedule is made
                                            up of 'unsigned int's;
```

[1] as part of the target configuration, one can have a key called `inherit_from` that indicates what other configurations to inherit data from. These are resolved recursively.

Inheritance works as a set of default values that can be overridden by corresponding key values in the inheriting configuration.

Note 1: any configuration table can be used as a template. Note 2: pure templates have the attribute `template => 1` and cannot be used as build targets.

If several configurations are given in the `inherit_from` array, the values of same attribute are concatenated with space separation. With this, it's possible to have several smaller templates for different configuration aspects that can be combined into a complete configuration.

Instead of a scalar value or an array, a value can be a code block of the form `sub { /* your code here */ }`. This code block will be called with the list of inherited values for that key as arguments. In fact, the concatenation of strings is really done by using `sub { join(" ",@_) }` on the list of inherited values.

An example:

```
    "foo" => {
            template => 1,
            haha => "ha ha",
            hoho => "ho",
            ignored => "This should not appear in the end result",
    },
    "bar" => {
            template => 1,
            haha => "ah",
            hoho => "haho",
            hehe => "hehe"
    },
    "laughter" => {
            inherit_from => [ "foo", "bar" ],
            hehe => sub { join(" ",(@_,"!!!")) },
            ignored => "",
    }

    The entry for "laughter" will become as follows after processing:

    "laughter" => {
            haha => "ha ha ah",
            hoho => "ho haho",
            hehe => "hehe !!!",
            ignored => ""
    }
```

[2] OpenSSL is built with threading capabilities unless the user specifies `no-threads`. The value of the key `thread_scheme` may be `(unknown)`, in which case the user MUST give some compilation flags to `Configure`.

[3] OpenSSL has three types of things to link from object files or static libraries:

- shared libraries; that would be libcrypto and libssl.
- shared objects (sometimes called dynamic libraries); that would be the modules.
- applications; those are apps/openssl and all the test apps.

Very roughly speaking, linking is done like this (words in braces represent the configuration settings documented at the beginning of this file):

```
shared libraries:
    {ld} $(CFLAGS) {lflags} {shared_ldflag} -o libfoo.so \
        foo/something.o foo/somethingelse.o {ex_libs}

shared objects:
    {ld} $(CFLAGS) {lflags} {module_ldflags} -o libeng.so \
        blah1.o blah2.o -lcrypto {ex_libs}

applications:
    {ld} $(CFLAGS) {lflags} -o app \
        app1.o utils.o -lssl -lcrypto {ex_libs}
```

[4] There are variants of these attribute, prefixed with `lib_`, `dso_` or `bin_`. Those variants replace the unprefixed attribute when building library, DSO or program modules specifically.

Historically, the target configurations came in form of a string with values separated by colons. This use is deprecated. The string form looked like this:

```
"target" => "{cc}:{cflags}:{unistd}:{thread_cflag}:{sys_id}:{lflags}:
             {bn_ops}:{cpuid_obj}:{bn_obj}:{ec_obj}:{des_obj}:{aes_obj}:
             {bf_obj}:{md5_obj}:{sha1_obj}:{cast_obj}:{rc4_obj}:
             {rmd160_obj}:{rc5_obj}:{wp_obj}:{cmll_obj}:{modes_obj}:
             {padlock_obj}:{perlasm_scheme}:{dso_scheme}:{shared_target}:
             {shared_cflag}:{shared_ldflag}:{shared_extension}:{ranlib}:
             {arflags}:{multilib}"
```

Build info files

The `build.info` files that are spread over the source tree contain the minimum information needed to build and distribute OpenSSL. It uses a simple and yet fairly powerful language to determine what needs to be built, from what sources, and other relationships between files.

For every `build.info` file, all file references are relative to the directory of the `build.info` file for source files, and the corresponding build directory for built files if the build tree differs from the source tree.

When processed, every line is processed with the perl module Text::Template, using the delimiters `{-` and `-}`. The hashes `%config` and `%target` are passed to the perl fragments, along with $sourcedir and $builddir, which are the locations of the source directory for the current `build.info` file and the corresponding build directory, all relative to the top of the build tree.

`Configure` only knows inherently about the top `build.info` file. For any other directory that has one, further directories to look into must be indicated like this:

```
SUBDIRS=something someelse
```

On to things to be built; they are declared by setting specific variables:

```
PROGRAMS=foo bar
LIBS=libsomething
MODULES=libeng
SCRIPTS=myhack
```

Note that the files mentioned for PROGRAMS, LIBS and MODULES *must* be without extensions. The build file templates will figure them out.

For each thing to be built, it is then possible to say what sources they are built from:

```
PROGRAMS=foo bar
SOURCE[foo]=foo.c common.c
SOURCE[bar]=bar.c extra.c common.c
```

It's also possible to tell some other dependencies:

```
DEPEND[foo]=libsomething
DEPEND[libbar]=libsomethingelse
```

(it could be argued that 'libsomething' and 'libsomethingelse' are source as well. However, the files given through SOURCE are expected to be located in the source tree while files given through DEPEND are expected to be located in the build tree)

It's also possible to depend on static libraries explicitly:

```
DEPEND[foo]=libsomething.a
DEPEND[libbar]=libsomethingelse.a
```

This should be rarely used, and care should be taken to make sure it's only used when supported. For example, native Windows build doesn't support building static libraries and DLLs at the same time, so using static libraries on Windows can only be done when configured `no-shared`.

In some cases, it's desirable to include some source files in the shared form of a library only:

```
SHARED_SOURCE[libfoo]=dllmain.c
```

For any file to be built, it's also possible to tell what extra include paths the build of their source files should use:

```
INCLUDE[foo]=include
```

It's also possible to specify C macros that should be defined:

```
DEFINE[foo]=FOO BAR=1
```

In some cases, one might want to generate some source files from others, that's done as follows:

```
GENERATE[foo.s]=asm/something.pl $(CFLAGS)
GENERATE[bar.s]=asm/bar.S
```

The value of each GENERATE line is a command line or part of it. Configure places no rules on the command line, except that the first item must be the generator file. It is, however, entirely up to the build file template to define exactly how those command lines should be handled, how the output is captured and so on.

Sometimes, the generator file itself depends on other files, for example if it is a perl script that depends on other perl modules. This can be expressed using DEPEND like this:

```
DEPEND[asm/something.pl]=../perlasm/Foo.pm
```

There may also be cases where the exact file isn't easily specified, but an inclusion directory still needs to be specified. INCLUDE can be used in that case:

```
INCLUDE[asm/something.pl]=../perlasm
```

NOTE: GENERATE lines are limited to one command only per GENERATE.

Finally, you can have some simple conditional use of the `build.info` information, looking like this:

```
IF[1]
 something
ELSIF[2]
 something other
```

```
ELSE
 something else
ENDIF
```

The expression in square brackets is interpreted as a string in perl, and will be seen as true if perl thinks it is, otherwise false. For example, the above would have “something” used, since 1 is true.

Together with the use of Text::Template, this can be used as conditions based on something in the passed variables, for example:

```
IF[{- $disabled{shared} -}]
  LIBS=libcrypto
  SOURCE[libcrypto]=...
ELSE
  LIBS=libfoo
  SOURCE[libfoo]=...
ENDIF
```

Build-file programming with the “unified” build system

“Build files” are called `Makefile` on Unix-like operating systems, `descrip.mms` for MMS on VMS, `makefile` for `nmake` on Windows, etc.

To use the “unified” build system, the target configuration needs to set the three items `build_scheme`, `build_file` and `build_command`. In the rest of this section, we will assume that `build_scheme` is set to “unified” (see the configurations documentation above for the details).

For any name given by `build_file`, the “unified” system expects a template file in `Configurations/` named like the build file, with `.tmpl` appended, or in case of possible ambiguity, a combination of the second `build_scheme` list item and the `build_file` name. For example, if `build_file` is set to `Makefile`, the template could be `Configurations/Makefile.tmpl` or `Configurations/unix-Makefile.tmpl`. In case both `Configurations/unix-Makefile.tmpl` and `Configurations/Makefile.tmpl` are present, the former takes precedence.

The build-file template is processed with the perl module Text::Template, using `{-` and `-}` as delimiters that enclose the perl code fragments that generate configuration-dependent content. Those perl fragments have access to all the hash variables from configdata.pem.

The build-file template is expected to define at least the following perl functions in a perl code fragment enclosed with `{-` and `-}`. They are all expected to return a string with the lines they produce.

```
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", ... ]
                                generator => [ "generatingfile", ... ]
                                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 ffile 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.
```

In all cases, file file paths are relative to the build tree top, and the build file actions run with the build tree top as current working directory.

Make sure to end the section with these functions with a string that you thing is appropriate for the resulting build file. If nothing else, end it like this:

```
  "";       # Make sure no lingering values end up in the Makefile
-}
```

Configure helper scripts

Configure uses helper scripts in this directory:

Checker scripts

These scripts are per platform family, to check the integrity of the tools used for configuration and building. The checker script used is either `{build_platform}-{build_file}-checker.pm` or `{build_platform}-checker.pm`, where `{build_platform}` is the second `build_scheme` list element from the configuration target data, and `{build_file}` is `build_file` from the same target data.

If the check succeeds, the script is expected to end with a non-zero expression. If the check fails, the script can end with a zero, or with a `die`.