Abstract

A drop-in MAKE structure for many similar and not so similar projects is described in this document. Based on the excellent treatises on Paul Smith's website, I created a platform-independent, multiple-environment MAKE structure based on GNU MAKE, automating most elements of Makefile maintenance. This document not only explains how it works, but how it has grown to be what it is today, so you may follow its path and insert deviations wherever necessary.
I use C, C++ and assembler to illustrate the structures, since I know these languages well. If you have another language that follows the same ways (compile multiple source files into as many objects, linking the objects into one executable, possibly using libraries), you'll find it easy to accommodate your additional requirements. Basic provisions for code-generators (`lex`, `yacc`, `IDL`) have `been` made; other needs may prove more difficult to incorporate.

Introduction

In modern development environments provisions have been integrated to build the whole project or parts of it. If and when you're developing for only one such environment, building your project is not a big issue. However, as soon as cross-platform development is a requirement, problems arise:

- There is not one development environment that covers all platforms;
- Various development environments are incompatible;
- In many cases you have to accommodate multiple compilers and linkers, all with their own set of options.

And even when only one platform and one compiler/linker combination needs to be supported, in many cases the configuration of one user cannot easily be transferred to other users, if only because of the different installation paths or root directories for the sources.
In many such cases the use of a portable MAKE-utility, especially GNU MAKE, is suggested to overcome these issues.
However, another issue arises especially with such a solution: Most developers would like to have the ease of use of an integrated environment like Microsoft's Visual Studio, and have no inclination to learn a whole new trade "setting up and maintaining the build environment". Admittedly, using GNU MAKE isn't always trivial.
From personal experience I can fill volumes about unsuccessful deployment of some sort of build system, often not really exceeding the level of a batch-file-based build system:

- The list of source files is only adapted whenever the linker complains about unresolved symbols, resulting in a completely missing module when a developer had forgotten to check in his locally developed sources for several weeks;
- The list of include directories contained so many unneeded and double entries that the cleaning of that list resulted in a 20% decrease in build-time;
- The dependencies between sources and include files was so unreliable, that a whole development team of more than 10 developers had to rebuild from scratch every time they touched an include file.

[Remark: These three specific illustrations occurred in different teams, although combinations have occurred in some other cases.]
For most of those who have some interest in build environments this will all be old news. So what do I intend to do here?
My intention is to develop a complete build environment for C/C++ projects, illustrating every step of the way using GNU MAKE, automating as much as possible, to obtain an environment where one can concentrate on the projects at hand and minimizing the maintenance of the environment itself. I will provide the reasoning for every step, implement the step, and provide test suites. If you will, use the Makefiles as-is, or follow the single steps for your own environment leaving out everything you don't need.

Recursive MAKE considered harmful?

Please note the question mark!

My initial impetus to undertake this development effort was the paper by Peter Miller, "Recursive MAKE considered harmful"[1], as available on the Internet at http://aegis.sourceforge.net/auug97.pdf. I can see his points, but I need not share his opinion as to the solutions. I've learned to look at recursion as a way to simplify solutions if applied to suitable problems. And traversing a directory tree, doing the same job at every level, certainly makes me look at a recursive approach.
The problems Mr. Miller lists in section 2 of his paper are not disputed. All these problems and more do exist and they are real. Only his opinion, that the recursive use of MAKE is to blame, is not mine.
This development effort goes to show that recursive use of (GNU) MAKE is feasible, and need not suffer from the real drawbacks illustrated by Mr. Miller.
In the final chapter, after I've presented my system, I will address all listed problems, and discuss these in detail, including the solution for each I will present in these pages.

One Acknowledgement

This will not be a long section, but one acknowledgement is unavoidable. I learned much of what I know about MAKE's possibilities from the articles Paul D. Smith, the current maintainer of the GNU MAKE package, listed on his website, http://make.paulandlesley.org/. And several if not most of his hints and tips have been incorporated in the structure presented here. Thanks, Paul!

Prerequisites

Some basics have to be present before we start. So I'll list them here.
Since cross-platform development is to be achieved, the environment has to run on a variety of host systems, most notably Windows, Linux and UNIX.
For that we'll need GNU MAKE. I will use version 3.81, the most recent version as of this writing, but 3.79.1 should be sufficient for most needs. I will explicitly mention used features exceeding 3.79.1, giving alternatives if possible.
In order to avoid the difficulties of differing command shells I have used `bash` throughout. The version shouldn't be an issue, but for the record I have used 3.1. For Windows I used Cygwin (http://www.cygwin.com). If you want to follow the examples step by step, be sure to install the GNU C compiler `gcc`.
However, many embedded development efforts are hosted on Windows. So, seeing the need, I have added support for Windows using `CMD.EXE`, using a version of GNU MAKE 3.81 targeted at Windows without the Cygwin-DLL, using Visual Studio Express 2008 as an example compiler.
I'll develop both versions in parallel.
For some specific features some utilities will be needed. These will be presented as needed, and then listed here.
Let's start this project in the 'correct' way, let's create a list of requirements first.

Requirements

No development project should be undertaken without a decent set of requirements. So what do we intend to achieve in the end? Don't worry, we'll tackle these requirements one-by-one, but here's the complete list:

- Store the objects in a different place from the sources. Better yet, we want the system to perform properly when presented with a read-only source tree (CD-ROM?).
- Store the objects in a structure reflecting the manner of building: For what processor are we compiling? Which compiler/linker combination are we using? Etc.
- Accommodate an arbitrarily deep source directory structure automatically, no (or hardly any) maintenance for the build system necessary when a directory or a directory level is added.
- Find the source files to be processed automatically, unless a developer explicitly wants otherwise.
- Find the necessary `include` directories automatically.

[1] Miller, P.A. (1998), *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25.

- Provide separate configuration files for all supported compiler/linker/toolset combinations, reusable between projects.
- Centralize all non-trivial Makefiles. This means that all Makefiles in various project-specific directories have to be trivial.
- Accommodate 3rd-party sources with their own Makefiles as sub trees within a project.
- Provide for tools that generate source code, like `lex`, `yacc` or `IDL`.

Detailed Requirements

Store objects in a different place

For small projects, having the objects in the same directory as the sources is no real problem. Maybe you can even live with 400 source files and their objects in one directory. But have you ever used such a configuration over a slow network? Or, better, in a ClearCase dynamic view? Things get really slow really fast. I have encountered a 400 file project where separating-out the object files (and creating them on a local disk instead of a ClearCase dynamic View) reduced the build time from 4 hours to 30 minutes maximum. So we will need a way to tell the system to create intermediate files somewhere else.

Store the objects in a structure reflecting the manner of building

The simplest case: You have a debug version and a release version. If you change versions from debug to release or vice versa, do you need to rebuild everything? I've seen many environments where this was absolutely necessary. Now add to that: A choice between simulation target and real-world target, a choice between multiple deployment platforms (Windows, AIX, Linux Debian/Redhat/SuSe, Solaris...), different compilers (native, cross-compiler, GCC-variant), and soon you have more versions than you'd have planned for. Doing a cleanup before switching platforms/versions may be doable when compilation times don't exceed a few minutes, but even half an hour is too much for such a concept, especially when you switch over often.
So, keeping all temporary files, generated files, binaries, etc. in separate directories differentiated by version would seem a valid requirement. If you touch one include file, all versions are updated minimally: Just what is really necessary, nothing more.

Accommodate an arbitrarily deep source directory structure automatically

Many, if not most build structures have been designed for a certain level of modularity: One level for the target, one for the module, one for the sub-module, and finally the files. If you would need an extra level (for instance, if you need a module at the sub-module level), many build structures cannot accommodate the extra level, at least not without much work.
I would like an arbitrarily deep directory structure accommodated and minimal work to the build structure whenever a directory is added.

Find the source files to be processed automatically

This may be a controversial topic, but I don't like changing Makefiles whenever I add a source file or an include file. Every manual action may introduce errors that cause a delay at the least, even if they are found immediately.
Some developers tend to keep temporary source files in their directories. Consider for yourself if you want or need to allow that. I don't.
Of course a method shall be provided to explicitly exclude one or more source files, or even to specify all source files 'by hand'.

Find the necessary include directories automatically

This is even more important than the previous requirement, especially when multiple files with identical names might exist. Maintaining a list with include directories can be a real pain. So as a rule I would like `MAKE` to find most include directories automatically. Of course some compiler include paths or library include paths need to be specified, but not for every include file belonging to the project.

Provide separate configuration files

A build structure shall be universal – in an ideal world. Modularity shall help achieve at least part of such universality. So we shall need configuration files for compilers, for linkers (possibly combined with compilers), for target platforms, for source platforms, for third-party libraries, etc. All these shall be reusable for new projects using the same resource.

Centralize all non-trivial Makefiles

The idea is to have trivial Makefiles in all/most of the directories relevant to the project, so that it's easy for developers to create (or copy) a valid Makefile for new directories they create. The complexity shall be encapsulated in centrally held MAKE-include files.
This is a non-omnipresent feature of GNU MAKE. If your MAKE cannot do that, you can get GNU MAKE for your platform(s), or stop reading here (well, maybe not quite that drastic...).

Accommodate 3rd-party sources with their own Makefiles

One cannot accommodate every single type of 3rd party software, without even knowing what they'd need. But if a subdirectory structure contains a Makefile with all necessary configurations, it shall be possible to accommodate that at least. And since most 3rd party sources can be set up using a Makefile, such a provision would cover most necessary source code distributions. Be aware that binary deliveries are another very common type, providing just headers for interfacing and libraries in binary form. They need a passive form of interface into the build structure only.

Provide for tools that generate source code

A tool that generates source code from some sort of configuration file is a challenge for most building systems. The problem is that the files produced are source files, but with the status of intermediate (object) files (they can be different for different platforms; they may need to be rebuilt if the source changes...). But since `lex`, `yacc`, and `IDL` are a fact of development life, we'd better cater for them.

Physical Software Architecture

Often ignored, the physical architecture of a project is what MAKE gets as input. MAKE doesn't care about interfaces, UML or singletons, it gets files and directories.
In order for MAKE to be able to automatically determine directory structure, files to be compiled, and `include` files to be found, I will pose some requirements for the physical architecture.
Please note: Most of these requirements (or restrictions, if you will) are not elementary to using the MAKE structure, only for the ease of use and the level of automation I myself require. Feel free to try variations, or respect the limitations of your specific environment.

About modules and sub-modules

One of the problems Peter Miller presented was to get the order of recursion into subdirectories correct. I will describe a way to specify dependencies between subdirectories, albeit on one directory level only. This is not only – in my eyes – an acceptable restriction, but also sound architecture: If you need the directory `/some/generator/directory` to be built before the directory `/some/other/folder`, the dependency is between `other` and `generator` within the `/source` directory. And circular dependencies are to be avoided altogether, on that level! If the directory `/some/generator/user` is again dependent on the directory `/some/other/folder`, you will have to rethink your directory structure, and I would advise you to review your physical software architecture.

Placement of include files

This issue seems to be an area of dispute, at least in my personal experience. I will describe my reasoning for the path taken. If you have other ideas, feel free to provide variations.
I consider a module to possibly consist of multiple source files with their corresponding header files, all in the module's directory (one directory per module), and at least one interface header. These interface headers are to be considered somewhat separately from their originating module, since they offer the module's interface to the outside world. Therefore, in my projects I create a directory in parallel to such a module's directory, called `include`, containing the interface headers of all the sibling module directories. Following the rules of encapsulation, the interface header of a module can be used by its siblings, but not the module-internal header files (module-internal interfaces).
Such a structure enables an automatic harvesting of include paths: When traversing the directory tree, on every level, if present, a directory called `include` is added to the include path used for the directory itself and its children. In this way we have a maintenance-free structure for providing include paths.
If a module's interface is needed somewhere else in the source tree, the interface header (not necessarily the module itself!) may be moved one or more steps (`include` directories) towards the file system root, thus getting a larger scope.
You may easily rename the directory for collecting interface headers, and if you do without automation, you may of course maintain an include path by hand using any other or no convention at all.
External libraries are different, and need a one-time setup only.

Test setup

For creating and testing the Makefile structure under development, I use the directory structure as depicted in Figure 1: A directory called `make` will contain all non-trivial Makefiles (`*.mk`, to be included), the `source` directory as well as all (sub-) module directories will contain a file called `Makefile` (with purely trivial content), and the various `include` directories will contain all interface headers.

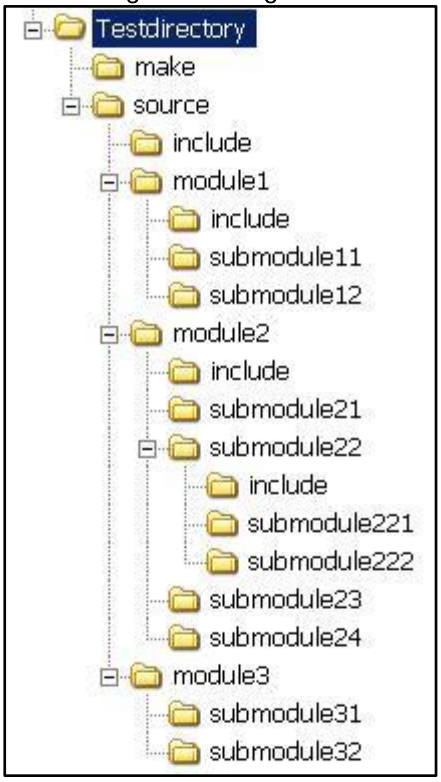

Figure 1 Directory Structure

The base directory (here `Testdirectory`) can be anywhere in your directory tree. For now it has to reside on a normal (writeable) hard disk, since we will start working in this directory and we need write permissions. The pathname up to `Testdirectory` shall not contain any spaces (since spaces are delimiters to GNU MAKE), other than that, no special restrictions apply.
In order to create Makefiles which are independent from their relative positions in the source tree, we need to define an environment variable pointing to the directory `Testdirectory`:

```
Set
YT_PBASE=/cygdrive/c/CSrc/makedev/Testdi
rectory
```

The construct using `cygdrive` is a Cygwin-specific way to indicate DOS-paths using a Unix-style pathname, indicating a DOS-pathname `C:\CSrc\makedev\Testdirectory`.

If you're working under a Unix/Linux variant, use the normal pathname.
When that is set, we create the trivial Makefile using just one line:

```
include $(YT_PBASE)/make/main.mk
```

and copy this Makefile into every (sub-) module's directory. That way, we create 14 files called `Makefile`, all containing just the single line:

```
source/Makefile
source/module1/Makefile
source/module1/submodule11/Makefile
```

```
        source/module1/submodule12/Makefile
        source/module2/Makefile
        source/module2/submodule21/Makefile
        source/module2/submodule22/Makefile
        source/module2/submodule22/submodule221/Makefile
        source/module2/submodule22/submodule222/Makefile
        source/module2/submodule23/Makefile
        source/module2/submodule24/Makefile
        source/module3/Makefile
        source/module3/submodule31/Makefile
        source/module3/submodule32/Makefile
```

If we now start GNU MAKE from the `source` directory, MAKE will report an error, since it cannot find `main.mk`.
We'll start developing that file now.

Traversing the tree

For now we will leave the directories empty, and implement the recursive descent in `main.mk`.
The following Makefile will be sufficient at first:

```
  all: RECURSE

  list_dirs := $(dir $(wildcard ./*/Makefile))
  list_dirs := $(patsubst ./%,%,$(list_dirs))

  .PHONY: RECURSE $(list_dirs)

  ifneq ($(strip $(list_dirs)),)

    RECURSE: $(list_dirs)

    $(list_dirs):
<T>  +@echo "Make[$(MAKELEVEL)]:$@"
<T>  +@$(MAKE) --directory=$@ --no-print-directory
$(MAKECMDGOALS)

  endif
```

You can find a commented version of this `main.mk` in the file `001Traverse.zip`. I'll dissect the Makefile here, too.
We define a default target `all`, dependent on the phony target `RECURSE`. If such a phony target is defined in the Makefile, the associated commands are unconditionally executed, but MAKE will not complain if the target is not present.
Then, using the MAKE function `wildcard`, all subdirectories containing a file called `Makefile` are concatenated into the variable `list_dirs`, retaining only the path component (with terminating slash `/`). The pattern substitution `patsubst` removes the preceding pathname components, keeping only the subdirectory's name with the slash appended.
In the leaf-directories the variable `list_dirs` will now be empty, so the rest of the Makefile will remain empty, and MAKE will not need to do anything. In other cases, we have still found subdirectories to traverse into, so we declare `RECURSE` to be dependent on the phony target(s) in `list_dirs`, and all such targets are to have commands performing the actual recursive call for MAKE, calling MAKE using the respective subdirectory as working directory (so the contained `Makefile` will be used automatically). `$(MAKECMDGOALS)` will contain any goals we indicated on the initial command line, like `all`, or (not supported yet) `clean`.
Two notes: The `<T>` indicates the use of a real TAB-character, necessary for MAKE to differentiate between MAKE syntax and commands to be passed on to a shell.
The `echo` command is just to have MAKE bring at least some output as to where it is currently executing, in a way more in line with my preferences. Leaving out the MAKE option `-no-print-directory` would also produce this output on entering and leaving a directory, but the different message layouts clutter the output window too much (in my opinion).
The `+` modifier will execute these commands even if MAKE is told not to execute any commands, just to list them (option `-n`), and `@` tells MAKE not to echo any command lines.
Experiment with this setup until you get a feel for what it does.
This small example will already recurse through any directory tree, of course without doing anything inside these directories, but finding all relevant siblings and respawning itself there.

Directory Dependencies

What even this simple structure already offers is the possibility to indicate dependencies between modules or sub-modules, on any level. Refer to the file source/module2/Makefile in the file 001Traverse.zip.
Since every subdirectory containing a Makefile is made into a phony target for MAKE, indicating dependencies between directories is trivial:

```
submodule21/ : submodule23/
```

This indicates that the sub-tree starting at source/module2/submodule23 must be traversed before starting in the sub-tree starting at source/module2/submodule21. If you now start MAKE, you will see that submodule23 is processed before submodule21. That's all there is to it. Please observe the terminating slashes; they are necessary, since the phony targets MAKE creates in list_dirs also contain these terminating slashes.
Please make sure that such module dependencies are listed after the include directive, in order not to influence the default target all. The default target is the first target encountered in the Makefile, so listing the dependencies before the include directive influences the first target MAKE 'sees' in the respective Makefiles, which is not what we want.
Of course such dependencies cannot be automatically detected by MAKE, so you will have to supply them yourself whenever necessary.

Using CMD.EXE

If you have a MAKE available for use in the command shell windows offers (CMD.EXE for Windows XP), the structure we have defined can also be used for that command shell. We only have one prerequisite incompatible with CMD.EXE, the definition of the environment variable YT_PBASE for the root of the project.
If you set YT_PBASE to

```
Set YT_PBASE=c:/CSrc/makedev/Testdirectory
```

(Note the use of forward slashes, instead of backslashes), all works just like when using bash from the Cygwin distribution. As we proceed through the development, this will not remain the only issue. But I'll point out the issues as we encounter them, and offer solutions whenever possible.
One restriction applies, as far as I can see: The CMD-version of GNU MAKE seems to be unable to use multiple threads of execution, the -j option will be ignored. From what I read, this may have been remedied in version 3.81 or one of the subsequent patches, but I'll have to verify that.

Creating a mirror tree for intermediate files

Until now, the source tree is mostly empty, and we've only traversed the source tree doing nothing. What we really want is to compile some real source files.
Before we introduce some source files, however, we first need to think about where to store the intermediate files (e.g. object files, but not just those).
According to an essay by Paul Smith, it is easier for MAKE to create 'things' in the current directory while searching for the sources than the other way around. Features like VPATH and the extension-based variant vpath are especially intended for such use.
So the next step would be to create (if not already existing) such a directory, switch to it, and run MAKE from there. But the recursion should still be driven by the source tree!
You may want to refer to the excellent essay by Paul D. Smith, since the techniques used here will basically mimic his recommendations.

Choosing the name

We need to specify how the base of the intermediate's directory shall be called. This choice is not arbitrary because of the way I've chosen to differentiate the intermediate's directory from the source directory. When MAKE is started in some directory of the source tree, I scan for the name of the intermediate's directory in the current pathname. If it is not found, I assume to be somewhere in the source tree, and change over into the objects' directory.
I chose the name objects.
Be sure to specify a meaningful name. Say you prefer obj as the name. If now you have a module called globjob, you're potentially in trouble because of the embedded obj.

You may, of course, choose a completely different strategy, for instance comparing with the full path `$(YT_PBASE)/objects`, or testing for the presence of a special file or directory, that's up to you.

Restarting MAKE in the intermediate's directory

Using this name we shall create a subdirectory parallel to `make` and `source` (i.e. `$(YT_PBASE)/objects`). For briefness and for now I will keep everything in one Makefile, `main.mk`, in contrast to Paul Smith's solution with included Makefiles. We'll split this up later.
Since creating a new directory is done using different commands depending on the shell, we start with defining a variable `makedirectory` containing the command to test for the existence of a directory, and if it doesn't exist, to create it:

```
ifeq (1,$(YT_YN_USE_CMDEXE))
  makedirectory=if not exist $(1) md $(subst /,\\,$(1))
else
  makedirectory=if [ ! -d $(1) ]; then mkdir $(1); fi
endif
```

Here we use an environment variable to indicate whether we are using the Windows' `CMD.EXE` or not. For now, this differentiates between Cygwin `bash` and `CMD.EXE`; we'll expand on this later. For this to work, you'll need to set the environment variable `YT_YN_USE_CMDEXE` to `1` in `CMD.EXE`, and remove or clear the value for `bash`. It might be an idea to set this variable to `1` in the control panel, and then use your `.bash_profile` to set the value to `0`, otherwise `bash` will inherit the value set for `CMD.EXE`.
As you will recognize, `makedirectory` contains commands for the respective shell, using a parameter indicated as `$(1)` for the directory to be created. This will be necessary to create the intermediates' directory, as well as any subdirectories we'll want to create later.
Next we define a variable `YT_OBJDIRNAME` to contain the chosen name for the intermediates' directory:

```
export YT_OBJDIRNAME:=objects
```

We can check for the presence of this name in the full path of the current directory. If we don't find it, we're in the source directory somewhere, and we need to change over to the intermediates' directory, creating it in the process, if necessary. For that I more-or-less used the file `switch.mk` from Paul Smith on his website:

```
ifeq (,$(findstring $(YT_OBJDIRNAME),$(CURDIR)))
  YT_OBJDIR := $(YT_PBASE)/$(YT_OBJDIRNAME)

  .SUFFIXES:

  .PHONY: $(YT_OBJDIR)
  $(YT_OBJDIR):
<T>  +@$(call makedirectory,$@)
<T>  +@echo "Make : $@"
<T>  +@$(MAKE) -C $@ -f $(CURDIR)/Makefile \
YT_SRCVPATH=$(CURDIR) YT_OBJBASE=$(YT_OBJDIR) \
--no-print-directory $(MAKECMDGOALS)

  Makefile : ; @:
  %.mk :: ;

  % :: $(YT_OBJDIR) ; @:

else
```

Refer to Paul's website for a detailed description, but here's the gist of it:

- We disable most of the built-in rules using `.SUFFIXES`.
- We declare the intermediates' directory a phony target, and provide instructions to create the directory if necessary, and then restart MAKE 1.) in that directory using `-C`, 2.) with the current Makefile using `-f`, 3.) with a reference to the relevant source directory in `YT_SRCVPATH`, 4.) and a reference to the relevant intermediates' directory using `YT_OBJBASE`.
- We provide empty rules for the Makefiles because of the fit-all rule below, to prevent endless recursion.

- And finally we provide a rule for whatever target is valid, dependent on the intermediates' directory target, effectively restarting MAKE in that directory.
- The `else` listed is to encapsulate the rest of the Makefile, a corresponding `endif` can be found as the final statement of the Makefile.

For layout reasons, the long line containing `$(MAKE)` is split into multiple lines, with backslashes escaping the newline. Even if it might work, I do not recommend such a notation in Makefiles, and urge you to keep it all in one line, even if this document has to accept the layout restrictions. The samples provided will always keep everything necessary in one single line.

Recursion through the source directory

In the remaining part of the Makefile you can see the contents from the previous instalment's Makefile. However, a few changes are necessary, since we now want to execute MAKE from the intermediates' directory, but the Makefiles still reside in the source tree, and we want to traverse that tree recursively.
The first change is the way to populate the `list_dirs` variable: Since the Makefiles reside in the source tree, we need to replace the '`./`' (indicating the current directory) by the source tree reference `YT_SRCVPATH`, both when scanning for Makefiles and when replacing the path component to extract only the subdirectory's name.

```
list_dirs := $(dir $(wildcard $(YT_SRCVPATH)/*/Makefile))
list_dirs := $(patsubst $(YT_SRCVPATH)/%,%,$(list_dirs))
```

Finally, the only changes left are in the commands used to traverse the source tree.
I personally like it when my intermediate files are stored in a directory tree mimicking the source tree, i.e. all directories in the source tree have an identically named representation in the object tree. If you would like to have it all in one directory, you'll have to adapt.
My structure possibly requires the creation of a (sub-) directory before starting the next recursion of MAKE. For that I reuse the `makedirectory` variable we already discussed, of course with a different pathname, composed of the directory we are currently in (`YT_OBJBASE`) and the next directory to be traversed, indicated by `$@`:

```
<T>  +@$(call makedirectory,$(YT_OBJBASE)/$@)
```

The next statement is just information as to where we intend to go (I'll get back to the `patsubst` in a second), and the final change is the recursive call to MAKE:

```
<T>  +@$(MAKE) -C $(YT_OBJBASE)/$(patsubst %/,%,$@) \
YT_SRCVPATH=$(YT_SRCVPATH)/$(patsubst %/,%,$@) \
YT_OBJBASE=$(YT_OBJBASE)/$(patsubst %/,%,$@) -f \
$(YT_SRCVPATH)/$(patsubst %/,%,$@)/Makefile -no-print-directory \
$(MAKECMDGOALS)
```

Again be aware that all is on one line!

- We start the new MAKE in the subdirectory we just created (in the intermediates' tree)
- using the source tree reference augmented with the new directory component in `YT_SRCVPATH`
- using the intermediates' tree reference augmented with the new directory component in `YT_OBJBASE`
- using the Makefile in the source directory

A final remark on the use of `patsubst` here: There seems to be a difference in the representation of `$@` depending on whether the directory already exists or not. If the directory does not exist, `$@` expands to the directory's name without a trailing slash, like `module12`, if that subdirectory (in the intermediates' tree) already exists, MAKE appends a slash like in `module12/`. In most cases, this is no problem, however, when composing directories using `$@` this may result in double slashes being used. To avoid all such problems, the trailing slash is first removed, and appended only where necessary.
It's time for the second instalment, the files can be found in `002Objects.zip`.

Providing sources

The next step is a very small one: I've created sources in the directory structure. If you look at those, you can see some of the conventions I tend to use.

1. Use include guards with a consistent naming convention, enabling identification of include guards using wildcards (`__INCLUDED_*__`).

2. Use external include guards in C/C++ sources from the second include directive onwards (the first wouldn't make sense).
3. Interface headers for a module belong in the include directory in parallel to the module itself, or more towards the root (see the sub-modules for module 3).

A first try at compilation of sources

Now that we have populated the source tree with sources, we start thinking about compiling source files. Since real compilers demand too attention to details, for now we pretend to have a compiler imitated by the `touch` command.

Adding a pattern rule

We add the following lines towards the end of `main.mk`, but of course before the final `endif`.

```
.SUFFIXES:
.SUFFIXES: .c

%.obj: %.c
<T>  @echo .c   to $(YT_OBJEXT): $(notdir $<)
<T>  touch $@
```

The first `.SUFFIXES:` rule deletes all 'known suffixes' and their associated 'suffix rules'. This isn't strictly necessary, since we will be defining all needed rules as pattern rules, overriding the ‚suffix rules', but it is good documentation anyway.
If you use an extra script for calling `MAKE`, consider using the option `-r` as a standard option in your script, as it disables all built-in rules from the start.
What this pattern rule is missing is the location of the files. For the object files this is no problem, since they are intended to be created in the intermediates' directory, which happens to be the current directory. But then, how is `MAKE` to find the corresponding source file(s)? Here's where `VPATH`, or better, the more flexible `vpath` comes in. It lets you find your source files, without indicating them in the rules.
I tend to keep the `vpath` instructions in the vicinity of the `.SUFFIXES:` rule, to easily make sure I've covered all extensions, but because of the two-pass structuring of `MAKE`, there is no real need:

```
vpath %.c $(YT_SRCVPATH)
```

Listing the objects

The Makefile now knows how to build „objects", but it doesn't yet know which objects to build. For that, we'll first determine which sources are available:

```
YT_SOURCES:=$(notdir $(wildcard $(YT_SRCVPATH)/*.c))
```

We list all C files in the source directory, strip the pathname components and put the names into the variable `YT_SOURCES`. And from that variable we create the list of objects to create:

```
YT_OBJECTS := $(patsubst %.c,%.obj,$(YT_SOURCES))
```

And finally, we have to indicate, when we call `MAKE`, that we want these objects to be created. To do that, we indicate the objects as prerequisites of `all`:

```
all: RECURSE $(YT_OBJECTS)
```

Under `bash`, this will probably run immediately. Under `CMD.EXE`, a tool called `touch.exe` will be missing. You may go a get a free version of this tool at http://www.codeproject.com/KB/applications/touch_win.aspx (registration required), also linked from my website. If you then store `touch.exe` where you stored your `MAKE` utility, you're set.
And since they're both called 'touch', we need not differentiate in the Makefile.
It's time for the third instalment, the files can be found in `003Sources.zip`.

Looking for includes

Our 'touch'-compiler is quite happy, but a real compiler will need something more; it will need to find all the include files. We'll look for compiler- and library includes later. For now we just want to find the project's own include files.
The starting point is easy: The directory where the primary source file (the one we are trying to compile) resides is also the first candidate when we're looking for includes:

$(YT_SRCVPATH). But then, the compiler will know that without help (unless you're using options to make the compiler act differently), so we do not want the current directory specified in the compiler's command line. We need to tell MAKE, however. So we start with a new vpath directive:

```
vpath %.h $(YT_SRCVPATH)
```

This time for include files only.

Harvesting include directories

Of course, since we'll also need interfacing include files, not all include files will reside in the C-file's directory. Refer to the section "Placement of include files" (page 5) for the rationale and the structure.

```
export YT_INCLUDEPATH := $(strip $(wildcard
$(YT_SRCVPATH)/include) $(YT_INCLUDEPATH))
```

What this does is the following:

- At each level of the Makefile recursion (except the first one, when we switch from the source tree to the object tree), MAKE looks for the existence of a directory called include being a valid subdirectory of the current source directory.
- If yes, it is added to a string of such valid directories, the nearest one being the first.
- If not, it is forgotten.
- The string is exported for use in further recursive calls to MAKE.

Be aware that this string can vary over the various levels of MAKE recursion, but if your partitioning is correct, the compiler will always find the necessary include files.
However, for more than just this reason, it is not recommended to have multiple files with identical names in one source tree. If the wrong file is found by the compiler, this can be serious and potentially hard to find. But also, dependency checking – introduced later – is dependent on all files having different names.
One thing remains: We now have a variable indicating the directories where to look for include files, but we still must inform MAKE by expanding our vpath directive:

```
vpath %.h $(YT_SRCVPATH) $(YT_INCLUDEPATH)
```

The vpath directive for include files is not co-located with the vpath directive for source files, since those files are not directly relevant to the pattern rules, but directly relevant whenever the way of finding include files is modified. They are not needed for any compiler calls, since we can use the variable itself (in modified form) for that, but later on, the dependency checking needs to find the source files, including the include files. We could have left it out here, and then revisited the subject when dependency checking is discussed, but I decided to put it in here. The rationale is more easily conveyed in this context.
If you want to confirm that this is working (before we start deploying a real compiler!), you can add another echo statement to the pattern rule performing the touch command:

```
<T>  echo Include path: $(YT_SRCVPATH) $(YT_INCLUDEPATH)
```

Everything will work for both bash and CMD.EXE, even though both will use different pathnames. Cygwin will offer Unix-like pathnames augmented with /cygdrive/…, whereas the Windows shell will offer drive letters and the like, using forward slashes as separators. We will (need to) return to these differences several times.

Engaging a real compiler

Basically, we have all the information that a compiler needs, so let's start using a real compiler. For simplicity reasons, I'll start with Cygwin and the GNU C compiler gcc. Don't worry, we'll be using the Microsoft Visual Studio Express 2008 compiler a few sections later, both with Cygwin bash and with the CMD.EXE Windows shell, but at the moment I do not want to clutter the issue with the relatively complex pathname conversions necessary. And we'll need to do some partitioning before VS 2008 can be easily accommodated.

Compiler command line

Using Cygwin, most standard utilities reside in the default path, so chances are good that gcc is installed in /usr/bin and directly callable from the bash-prompt. If not, you'll have to provide a full pathname to the executable. Other than that, the command line for invoking the compiler is pretty straightforward:

```
<T>  gcc -c $< -o $@
```

- `-c` indicates that we do not want to link any objects (yet)
- `$<` represents the source file to be compiled
- `-o` introduces the output filename, represented by
- `$@`

Let's run the Makefile and see what happens. OK, MAKE will fail, since the compiler cannot know anything yet about include files, so the compile will fail. But just for the fun of it, let's have a look at the `gcc` command line as MAKE will produce it:

```
gcc -c
/cygdrive/c/CSrc/makedev/Testdirectory/source/module1/submodule11
/sm11.c -o sm11.obj
```

[All is output on one line]
Here you see the effect of the `vpath` directive for finding source files: The file name we're looking for is `sm11.c`, but since we're executing from somewhere along the intermediate's path, MAKE will need the `vpath` pointer to find the source file; and when it does, it will use the full pathname for any such reference.

Include options

To let the compiler know where to look for include files, we'll have to provide the variable `YT_INCLUDEPATH` on the command line (for now; we'll enhance that later). But the compiler will want the option `-I` prepended to each single path, so we have to do some string handling:

```
<T>  gcc -c $< $(patsubst %,-I%,$(YT_INCLUDEPATH)) -o $@
```

The `patsubst` function takes every space-delimited string, and puts a '`-I`' in front of it. Simple, but that's it. Now it works, creating a real object file, instead of 'touching' a wannabe.

Some small enhancements

Selective silencing

It's time for cleaning up a bit, and I'll start with a little trick I've found useful for almost 10 years: An environment variable, I'll use `YT_S` for that, set to equal `@`.
Remember that GNU MAKE echoes all commands it executes before execution, unless you issue some option to silence MAKE, or use the `@`-sign in front of the command? Well, this wasn't quite the granularity I normally need.
Especially when dealing with complex and long compiler/linker/locater command lines, whenever they work, you don't want to see them. But as soon as they don't work, you'll need them in all detail, on any workstation showing the problem, and if necessary also remote or on the phone. And if you've tried instructing an experienced developer to change a Makefile, under guidance, only to find his editor settings clobbering the TABs…
So my Makefiles contain `@`-signs for most of the uninteresting commands like `echo` or `mkdir`, and interesting and/or complex commands are preceded with `$(YT_S)` instead:

```
<T>  $(YT_S)gcc -c $< $(patsubst %,-I%,$(YT_INCLUDEPATH)) -o $@
```

In my standard environment, `YT_S` is always set to `@`; but if necessary, I can clear the variable, run MAKE, and analyze the output, if necessary also when remotely supporting a developer.

Recognizing CFLAGS

Another issue has to do with a common MAKE variable `CFLAGS`. This variable is customarily used to communicate specific options to the C compiler. A user specifies it on MAKE's command line to add to or override the defaults MAKE has built-in.
OK, I will not use MAKE's built-in rules, but I can still be a decent citizen in the developer world and add `CFLAGS` to the options presented to the compiler.
But cleaning up, I'll lay the foundation for further enhancements later on, and define my own compiler flags.

```
YT_CPPFLAGS := $(patsubst %,-I%,$(YT_INCLUDEPATH))
YT_CFLAGS   := $(CFLAGS)
```

Other flags, for assembler, C++ and many more, can and will be defined later.
And now these options are used as parameters to the `gcc` command:

```
<T>  $(YT_S)gcc -c $(YT_CPPFLAGS) $(YT_CFLAGS) $< -o $@
```

This will make later enhancements localizable and clearer, without creating overlong command lines (at least in source form; expanded in a real project is a different issue, but for that we have YT_S!).

A 'clean' target

Well, this is a controversial topic. Considering the many opinions on this, I will have to address it, though. For now, I offer the simplest solution for the Makefile, a recursive clean:

```
.PHONY: clean
clean: RECURSE
<T>  @rm -f *.obj
```

It is certainly not the most efficient, for now it only works on Cygwin or Linux/Unix, it potentially deletes more than MAKE can create, is maintenance intensive, well, you name it. But it works!

Cleaning up

Before we implement the same functionality for Windows' CMD.EXE, it's time to clean up somewhat. In the end, we will differentiate the operating system, the shell and the tool-chain, all posing different requirements, so we'd better define a consistent structure from the beginning.
The first part of the Makefile to deserve its own file is the part to switch over into the intermediates' directory. Just like Paul D Smith, from whom the details come anyway, we'll call it switch.mk and put an include directive into main.mk:

```
ifeq (,$(findstring $(YT_OBJDIRNAME),$(CURDIR)))
  include $(YT_PBASE)/make/switch.mk
else
...
```

A second part, very small until now, also get's an own file: Everything that can be done once, and exported for use in recursive calls to MAKE, bounded by the ifeq checking for the variable MAKELEVEL being zero. Currently, we have only one assignment there, but we'll expand on that shortly:

```
ifeq ($(MAKELEVEL),0)
  include $(YT_PBASE)/make/level0.mk
endif
```

One thing to do in this level0.mk is to determine where we are at the start: Operating system, shell, etc. These things may require some elaborate methods, especially when, later, versions may matter, so we'd better do them only once. Be aware that this file is only read once, even before we restart in the intermediates' directory!
And when we know that, we can easily determine ourselves, which version of some 'subroutines' will be required. Therefore, for now the final 'cleaning' action is to move the makedirectory definition to after level0.mk.

The difference between...

... operating systems and shells

We need to differentiate between the way CMD.EXE is working, and the way bash is working. We're not using too many features of either, but we'll have to circumnavigate the discrepancies. So we'll start with initialization of the MAKE variables we want to use:

```
YT_USE_WINDOWS:=0
YT_USE_LINUX   :=0
YT_USE_CMDEXE :=0
YT_USE_BASH    :=0
```

The first difference is easy, at least for now: If we're on Windows (from NT 4.00 onwards, for all versions known to me, including Vista and 2008 Server), the environment variable OS is set to Windows_NT, mind the underscore. And if not, we're on Linux using bash:

```
ifneq (Windows_NT,$(OS))
  YT_USE_LINUX:=1
  YT_USE_BASH:=1
```

If it is, we're on Windows, but we don't yet know the shell we're using.
I chose to use the `PATH` environment variable: `CMD.EXE` uses semicolons as path separators, and as far as I know, semicolons are not allowed as legitimate characters in pathnames, neither on Unix/Linux, nor on Windows. So if we find at least one semicolon in the `PATH` environment variable, we're using `CMD.EXE`. Otherwise we're using `bash` (Cygwin assumed):

```
else
  YT_USE_WINDOWS:=1
  ifeq (;,$(findstring ;,$(PATH)))
    YT_USE_CMDEXE:=1
  else
    YT_USE_BASH:=1
  endif
endif
```

If you're using different environments (a stand-alone `bash`-implementation, a completely different environment, `COMMAND.COM` (?!), ...) you may very well need to differentiate-out some more details, or even expand on the variations offered here.
And finally, we're in `level0.mk`, where nothing makes much sense unless exported:

```
export YT_USE_WINDOWS YT_USE_LINUX YT_USE_CMDEXE YT_USE_BASH
```

Note how we used a different value to indicate the use of `CMD.EXE` than before. The `YT_YN_USE_CMDEXE` contained the `YN` (for 'yes/no'), indicating a user's choice. This choice has never been a real choice, but now we can delete the variable from our start-up scripts; the environment used is now determined automatically.

... compilers

One thing a Makefile cannot automatically deduce is the compiler to use. Many compilers can be installed on one single development system, possibly even in different versions. I will show here how you can select a compiler - even a default compiler if that suits your environment - and the variables to be defined by such a choice of compilers.
For one, using Cygwin we have directly used the GNU C compiler `gcc`. This is not a good practice, so we have to define variables for using the compiler.
The standard way for GNU `MAKE` is to use the `MAKE` variable `CC` for that. So, when `CC` is set (as an environment variable), we'd use that definition for the compiler, no questions asked. If it is not set, however, we need another definition.
I always define a so-called 'toolchain'. A compiler usually is not the only tool to be used; we know pre-processors, assemblers, linkers, loaders, all belonging to only one tool-chain.
For the moment, I'll use defaults for the tool-chain:

- If `CMD.EXE` is to be the shell, I'll default to the Microsoft compiler `CL.EXE`, I've taken Visual C++ 9.0 Express Edition;
- If `bash` is the shell of choice, the GNU compiler tool-chain shall be the default, my current Cygwin `gcc` compiler is version 3.4.4.

But of course, such defaults shall be overridable. If the developer sets the environment or `MAKE` variable `YT_TC_SELECT`, the selected tool-chain shall be taken. Allowed/recognized values shall be `GCC`, `GCC344`, `VC`, and `VC9`, for now. The strings without version make sure that the newest available version shall be used.
And now for the implementation of all this...

Implementation

First we start with creating two additional Makefiles, `tc_GCC344.mk` and `tc_VS9.mk`; `tc` stands for 'toolchain'. Since at the moment we only need the C compiler proper, both files check for the presence of the CC variable, using that if found defined, and using `gcc` resp. `cl.exe` if undefined. Be aware that `CC` is a `MAKE` variable by default, so we need to check for `undefined` or `default`, since unless `MAKE` is called with the `-r` option, `CC` is never undefined.
As an example, here's `tc_VS9.mk`:

```
ifeq (,$(strip $(filter-out undefined default,$(origin CC))))
  export YT_CC := CL.EXE
else
  export YT_CC := $(CC)
endif
```

Now, in `level0.mk`, we check for a tool-chain selection by the user: If `YT_TC_SELECT` is defined (see above for allowed values), that value is used. Otherwise, `GCC` is used for all `bash`-shells, `VS` is used for `CMD.EXE`:

```
ifeq ($(origin YT_TC_SELECT),undefined)
  ifeq (1,$(YT_USE_CMDEXE))
    YT_TC_SELECT := VS
  endif
  ifeq (1,$(YT_USE_BASH))
    YT_TC_SELECT := GCC
  endif
  ifeq ($(origin YT_TC_SELECT),undefined)
    $(error No supported shell for default toolchain detection)
  endif
endif
```

Next, we define a variable `YT_TC_MAKE_INCLUDE` to contain the intended `MAKE` include, using `$(wildcard` to select all relevant files (like `tc_VS7.mk`, `tc_VS8.mk` and `tc_VS9.mk`), `$(sort` the list and use `$(lastword`[2] to get the highest version number. Be aware, when defining new support files, that the sorting is done alphabetically; `MAKE` doesn't know about (version) numbers, so `VS9` is 'newer' than `VS12`! Use `VS09` or similar, if necessary.
Here's the code:

```
YT_TC_MAKE_INCLUDE := $(lastword $(sort $(wildcard
$(YT_PBASE)/make/tc_$(YT_TC_SELECT)*.mk)))
ifeq (,$(strip $(YT_TC_MAKE_INCLUDE)))
  $(error Toolchain make include not found.)
endif

include $(YT_TC_MAKE_INCLUDE)
```

(The first line is split across two lines for layout reasons only.)
If you define an environment variable `YT_TC_SELECT` equal to `GCC344`, `MAKE` might take a `GCC3447`, but not a `GCC295`, and it will always select `GCC`, even if using `CMD.EXE`. Of course, the selected compiler has to be available on your system, accessible through your path, but that's your responsibility. If necessary, look for `vcvars32.bat` in your Microsoft Visual Studio directory; `gcc` is usually installed in your path, if at all.
And if all is performed as described, we can now compile the whole tree also with the Visual Studio compiler.

... platform-dependent tools

When you are testing this setup, you'll find one thing broken, at least using `CMD.EXE`: `MAKE clean` reports errors. The reason is quite simple: `CMD.EXE` doesn't know the command `rm`, until now, hard-coded in the Makefile.
To complete the differentiation for now, we create two kinds of `MAKE` includes, one for the host platform we're working with (named `WINDOWS` and `LINUX`), and one for each of the different shells we may be using (named `CMDEXE` and `BASH`). In `level0.mk` we define (and export) two further variables, `YT_NAME_PLATFORM` and `YT_NAME_SHELL`, initialized appropriately.
From those definitions we can include two `MAKE` includes, one for the platform:

```
include $(YT_PBASE)/make/platform_$(YT_NAME_PLATFORM).mk
```

and one for the shell:

```
include $(YT_PBASE)/make/shell_$(YT_NAME_SHELL).mk
```

All four possible files are initially created empty.
Now back to the original problem, a command for deleting files. Since GNU `MAKE` defines the `MAKE` variable `RM` by default, we again use our variant, but without the possibility for an override (that seems overkill in this case). In `shell_CMDEXE.mk` we define:

[2] `$(lastword` is a new feature of GNU `MAKE` 3.81; in order to support also versions earlier than that, the replacement `$(word $(words text), text)` is used, slower, but more compatible.

```
export YT_RM := del /q 2>NUL
```

The 2>NUL makes the command remain silent even if no files are found to be deleted, /q skips asking for confirmation. And in the bash variant shell_BASH.mk we keep the command as in the past:

```
export YT_RM := rm -f
```

So now we again have the same functionality as in the previous instalment, now for both Cygwin/Linux bash using gcc, and CMD.EXE using CL.EXE. Quite an effort for such a small improvement, but it'll get even better.

Mixing and mingling

Say, we like Cygwin as a shell, but are required to use VS9 for a tool-chain. Problems galore:

- Cygwin paths are unlike Windows paths
- Tools from the tool-chain will require drive letters when absolute pathnames are to be used
- Certain tools will require backslashes for path separators, even when relative paths are used
- All changes need to be "backward transparent": If we change something specific for Cygwin using VS9, other combinations still need to work properly.

Let's go one at a time.
First we set YT_TC_SELECT to VS in the Cygwin environment (use .bash_profile if necessary). From this moment on, using Cygwin will produce errors from the compiler. But as long as MAKE can find CL.EXE, all is well in respect to your setup so far. If not, please find out how to set the appropriate environment variables for Visual Studio, including the PATH.

Converting Cygwin pathnames

If you now start MAKE in a Cygwin bash, CL.EXE is found, but the command line looks curious for a CMD-based tool:

```
CL.EXE -c
-I/cygdrive/c/CSrc/makedev/Testdirectory/source/module1/include
-I/cygdrive/c/CSrc/makedev/Testdirectory/source/include
/cygdrive/c/CSrc/makedev/Testdirectory/source/module1/submodule11
/sm11.c -o sm11.obj
```

As can be expected, the cygdrive-containing pathnames are not understandable by the compiler; the compiler produces an error and exits.
The problem lies mainly in the directory paths MAKE will create for us traversing the source tree or using vpath, and used in the pattern rules. But all of these paths derive and expand from YT_PBASE, so if we replace all occurrences of $(YT_PBASE) with the corresponding drive-letter-path combination, the command will work.
When would we need this? Well, when using a Windows/Cygwin combination using non-Cygwin tools: Platform Windows, shell Bash. To make such conversion quick, we'll do the real conversion only once, for YT_PBASE, and create an exported variable YT_PBASE_WDL (meaning: with drive letter), and do a substitution wherever a tool needs a non-Cygwin pathname. And to avoid problems when using other combinations of shell and platform, we'll define the YT_PBASE_WDL variable also in those cases, but identical to YT_PBASE.
So in those cases, effectively no change is done.
Let's see how it works. First we define a helpful variable in level0.mk, representing a space:

```
null:=
space:=$(strip $(null)) $(strip $(null))
```

Then we go to shell_BASH.mk:

```
ifeq (1,$(YT_USE_WINDOWS))
  export YT_PBASE_WDL := $(word 2,$(subst
/,$(space),$(YT_PBASE))):/$(subst $(space),/,$(wordlist
3,99,$(subst /,$(space),$(YT_PBASE))))
else
  # If using bash on Linux, we have no problem.
  export YT_PBASE_WDL := $(YT_PBASE)
endif
```

Here's where the work is done. Be aware that there's only one line between the `ifeq` and the `else`, starting with `export`, and ending with a quadruple closing parenthesis. This is how it works:

- We take `$(YT_PBASE)` and replace all slashes with spaces, creating a space-delimited list of directory components, preceded with `cygdrive` and `<driveletter>`
- Of that we take the second word (the drive letter)
- We place a colon and a forward-slash
- And append the rest of the list starting at the third word, while replacing every space with a forward-slash again

So now `YT_PBASE_WDL` contains a path with a drive letter. The `else` clause is for all people without such problems, and it is repeated in `shell_CMDEXE.mk`.
What we need to replace are the pathnames for the source files and include directories. So the command line becomes:

```
<T>  $(YT_S)$(YT_CC) -c $(subst
$(YT_PBASE),$(YT_PBASE_WDL),$(YT_CPPFLAGS)) $(YT_CFLAGS) $(subst
$(YT_PBASE),$(YT_PBASE_WDL),$<) -o $@
```

All in one line, of course. And like this, it works[3].
Just for good measure, take a look at Cygwin using `GCC`. This combination now also starts using paths with drive letters, since we do not differentiate yet between native Cygwin tool-chains, and external (DOS-) tool-chains. But Cygwin `gcc` will handle such pathnames very gracefully, so need not take any action. If you want, you may override the definition of `YT_PBASE_WDL` in `tc_GCC344.mk` and make it equal to `YT_BASE` again, independent from the platform or shell.

Different intermediates

If you've been following this discourse, you may have noticed one problem: If you do a build with `CMD.EXE`, and immediately afterwards with `bash`, `MAKE` will report everything being up-to-date, even though the object files were produced by a completely different compiler. Before actually trying to link a binary we have to solve this issue.
The idea is to have separate data areas for all intended configurations, using separate directories for each one, for instance, a debug and a release version, different target platforms like Windows, Linux or embedded hardware configurations, or versions using different tool-chains.
In order to do that, we augment the intermediates' directory name with a separate subdirectory, more or less named after the configuration we are currently using.
We introduce a `MAKE` variable `YT_DIFFDIR` initialized to `Im`, adding subsequent components starting with a dash for each relevant configuration 'decision' made:

```
YT_DIFFDIR := Im
```

We already made two decisions: The platform (Windows or Linux), and the tool-chain (GCC or VS). So in the corresponding `MAKE` includes, we augment the definition of `YT_DIFFDIR`. As an example, look at `tc_VS9.mk`:

```
YT_DIFFDIR := $(YT_DIFFDIR)-tcVS
```

You may choose to add the version number, increasing the length of the directory name.
And at the end of `level0.mk`, we just need to make sure the correct value is exported.
To use this extra directory level, we need to augment `YT_OBJDIR` in `switch.mk`:

```
YT_OBJDIR := $(YT_PBASE)/$(YT_OBJDIRNAME)/$(YT_DIFFDIR)
```

Since all further subdirectories are derived from `YT_OBJDIR`, this is all. If you now build the tree with `CMD.EXE` (using VS by default), you'll have a different intermediates' directory from

[3] It still produces a warning about the use of the '-o' option being deprecated. We'll handle that later.

building with Cygwin `bash` (using GCC by default): `Im-pfWIN-tcVS` vs. `Im-pfWIN-tcGCC`, so both builds will keep out of each other's way. If you're using Linux to follow this discourse, your path will be augmented with `Im-pfLINUX-tcGCC`.
This has been quite some addition, so it's time for the next instalment, `005Differentiate.zip`.

Summary

What do I need to do to make this setup work as described?

1. Unpack one of the provided archive files to a directory **without spaces** in the pathname. The one with the highest number will have the most functionality; refer to the text for details.
2. Define an environment variable `YT_PBASE` to point to the absolute path where you copied the archive file (using forward slashes also when using `CMD.EXE` on Windows!); in this directory will be a subdirectory called `make`, containing all centralized `*.mk` Makefiles.
3. The environment variable `YT_YN_USE_CMDEXE` is no longer needed or relevant.

This will be enough for the fifth instalment. As before, I will expand on this documentation, extend this document, offer zipped snapshots of the `Testdirectory` project at every step, and announce each instalment through my blog.
Happy making!

Johan Bezem
http://www.bezem.de/
http://blog.bezem.de/