Configuration Builder Documentation

Who Needs to Know about PVCS Configuration Builder?

The folks who need to have a working knowledge of Configuration Builder (CB) are Developers and build engineers at New York Life who work on software applications. These developers are responsible for writing the code that powers the applications that keep the company in business. If your responsibilities include building the latest and greatest versions of these applications, then you need to know about Configuration Builder for PVCS. PVCS Version Manager archives all your files along with their position in your structure. You can build your components and systems directly from the PVCS archives by using PVCS Configuration Builder.

What is Configuration Builder?

Configuration Builder (CB) is a utility that builds a program, or group of programs. CB does a top-down build. PVCS Configuration Builder is one component of the suite of components that make up the PVCS tool suite, which includes PVCS Version Manger. Version management is the process of managing change to the components of a software system. Version Management is a sort of library for computer code. Each individual file is like a book, with a number assigned to each file, and new, logical numbers assigned when a file is changed or updated. It's like the Dewey Decimal system for software. The number assigned to the file is called a version number.

What does Configuration Builder do?

Configuration Builder ensures that the correct version of each appropriate file is used to "build" a specific application. You need to link and compile files to build your executable program. CB is a utility that "makes" or builds files and applications. (Building an application entails pulling together all the different files of code from various archive locations that are needed to run the application, and the subsequent creating of executables via compiling and linking).

With many developers working on numerous files and several applications simultaneously, it is important that the correct version of a file be included when an application is built. It may be the latest release, or a past release for maintenance. Previous versions of an application can be recreated by pulling together a different set of files.

How does it work?

While Version Manger creates archives of your program source code and other application files and keeps track of which files go with which application, Configuration Builder lets you decide which files have to be pulled together simultaneously to make your application executable. CB is a useful tool because it automates the build process. This frees the builder from having to constantly monitor the build, and can eliminate the need to constantly check "OK" at various stages of the build process. You write a small text file that contains the names of the files your application will need. This is called a "build script." It's like a recipe, where you list the ingredients you'll need to make your favorite dish. But it can change over time, so you must update your recipe. As your software applications grow, more files or components are added. The build scripts have to reflect the new tasty recipe items.

Then, you simply read your recipe when you want to cook your favorite dish, or in this case, compile and link your executable program. The recipe, or build script, will remember those ingredients and take them

off the library shelves the next time you need to "cook' your application. And it will only take the freshest ingredients, (most updated code). Or it can take older code if you want to create an earlier version of your application. (Stale ingredients work better for software than cakes). You may want to revert back to an earlier release that was bug free if the latest release isn't working. You may want to maintain parallel versions of software as well. For instance, your application may be targeted to both the Windows and Unix platforms. Common archives could be used with the build scripts. One script for Windows, and another script for Unix.

Why is Configuration Builder necessary?

With software applications being so complex and having so many revisions and versions, it's necessary to have a utility that is a sort of traffic cop for code. A build script is a centralized record of which files are needed where, and how one file depends on another in the grand scheme of things. This is known as **dependency**. The script pulls together all the required files, and can be called upon to automate what would be a slow manual process. Just as you need a shopping list to be efficient at the supermarket, you need a build script to be efficient at the code library. You also have the flexibility of reusing your build scripts depending on your needs, such as recreating an earlier version of an application to zero-in on where bugs occurred. If a new version of an application is not working properly, you can revert back to a previous version that *was* working properly. You just "plug-in" the version you want built, and the build script does the rest. This section of your request would look something like this at the command line;

release= "x".

In this case, x represents the release you want built. For example, you might request release #1, or # 2 or 3.01 or TEST or 8.00A, depending on

how you name or number your releases. You also need to sue the right build script to match the release you want to build. Different releases may contain more or fewer files as dependencies, or earlier or later versions of those files. Version Manager also archives the individual build scripts that build the various versions of your application. When you choose to build a version of your application, you use the build script that has the same label as all the files and modules that will be used in that particular build. That script will "pull" the files that it needs, and it's not necessarily the latest version, but the version that you specify.

What is the Softcraft Process?

The Softcraft Laboratory, with world headquarters based in Southington, CT, has adapted Configuration Builder to better meet the specific needs of developers at New York Life. Through a series of templates and commonly shared files, referred to as "**Rosetta Stones**," the build process has been made more streamlined and automated than it would be out of the box. We will discuss these Rosetta Stone files in detail later, but the four are :

Directory. inc
Tools.inc
Tools32.inc
Translate.inc

As you can see, each is an "include" file. (inc.)

We created build script templates for the three major types of builds: Top Level, Middle Level and Bottom Level

Top Level Build (i.e. NYLapps3)

.

Middle Level Build (i.e. Apps)

Bottom Level Build (i.e. askio.dll)

A top level build script can be used to build an entire application. This can be a very large program which consists of many components. This build script would construct the various directories within the hierarchical structure of the large application. The top level build script calls the lower level make files. But how do you build the bottom level make files? That's done with the bottom level build scripts. These are used to build the various software components, such as .dll's, .ocx's and .exe's.

Additionally, there's the "Middle Level" build scripts. These can be used when an application is very large. Middle level scripts allow subsections of the giant application to be built. It is a subset of the overall system. There may be times when building an entire application is too time consuming or not necessary, and a middle level build would be more appropriate.

Softcraft's build scripts help support the individual system architecture of a particular application. There is the flexibility to support subsystems. The templates supplied by Softcraft support changes made to top, middle or bottom level scripts. You are free to build all or part of a system at any time. The choice is yours. For example, you can choose to build a single component like a .dll, .ocx .cls etc. or a complete application or suite of applications. One advantage of CB is the build time is reduced. Since the build is based on time- stamps, only elements that have been changed since the last build need to be rebuilt. This eliminates the need to rebuild every file or every module or every subdirectory. This saves

valuable time each time you choose to make your application.

To sum up, The Softcraft Laboratory has provided the templates and four common files, (Rosetta Stone Files) to build your application automatically by building the directories and populating them with your preferred set of files. Configuration Builder automatically checks out the designated versions from the PVCS *archives.* The archives are the location where your source files are kept along with the additional data such as time stamps. Archives contain the current copy of the file as well as information leading back to the original copy of the archived file. Archives contain the changes made to a file, descriptions of the changes, information about who made the changes, and the dates and times when the changes were made. Every time you modify a file and check it into the archive, it becomes a new revision. When you check out a revision from an archive, it becomes a workfile.

Using Configuration Builder with Version Manager enables you to extract files stored in archives for use in the build process. You can also use a build script to check workfiles into archives when you are done working with them.

How do I buld my application? (For DOS/Windows)

There are a few steps involved

1. Decide where on your system the application will be built.
2. Check out the necessary files from PVCS into that location.
3. Access the DOS prompt.
4. Have the DOS prompt point to your build script.
5. Type the build command.
6. Hit **enter** on your keyboard.

Your application should start building. You will see information on your DOS screen indicating what stage your build is in, or if there is an error building your application.

Top-level build files:

A top-level build file creates the build directory hierarchy under the directory from which it is executed. For instance, let's assume your project is named Nylapps and it has the following structure:

nylapps
nylapps.mak
    apps
    apps.mak
        annuity
            annuity.exe
        fpra
            fpra.exe
        nylaz
        nylaz.exe
        lifapps
            Appsup.ocx
            Clientprofile.ocx
            Crules.ocx
        sehaz.exe
        sesimp.exe
    dlls
    dlls.mak
        askio.dll
        askreq.dll
        common.dll
    include
        common.h
        mfedit.h
        d4all.h
    ref
        5ytq2.ocx

allocation.ocx
amendctl.ocx

The top-level build script will create the directories highlighted in yellow. It will also populate those folders with their associated **make** files (.mak highlighted in green). The make files are then executed, which results in a lower level build. (It's like begettin', or creating the next generation. The parent breeds the child which breeds the grandchild in this hierarchy. The top level begets the middle level which begets the bottom level. The begettin' continues until the lowest level build scripts are reached.)

Some directories contain items that are not "built" per se.. The **include** directory for example simply holds header files that are checked out of PVCS. The four **Rosetta Stone** files are include files. Other build scripts will use these common files in their builds. Another example above is the **ref** directory. It contains binary reference files that were checked into the archives. In this case, visual basic needs these reference files to maintain binary compatibility. (Binary compatibility implies that an earlier or later version of a program can still run successfully even though some aspects of the application platform may have been modified i.e. adding a new user interface.)

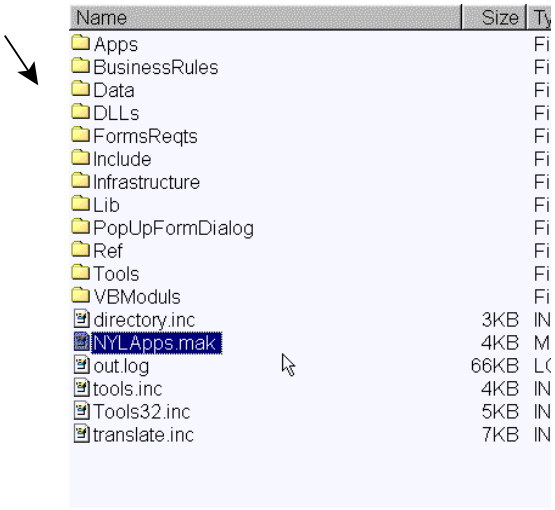
Building an application's hierarchy has the domino effect. You build one component, which then allows another component to be built, and so on, and so on and so on. You work from the top down. From a top level build script to a bottom level build script. The code trickles down the mountain of files.

For example. the top level build script, which is contained in a file called Nylapps.mak, builds some applications and some .dll's, In turn, those

applications have make files, and those .dll's have make files. And they in turn produce applications that contain .mak files. Apps.mak leads to annuity and life apps. Dlls.mak leads to askio which will breed askio.mak and common . Common then leads to common.mak and common.dll. It's a cascading effect. Of course, if something is wrong somewhere down the line, the trickle will stop, (unless you force it to continue with a -k option, *see below*) and you'll get an error message. At the bottom, your build script will be calling for individual bottom level files like .c files or .h files or a call to the compiler.

Here are the set of directories a top level build script might actually build as viewed from windows explorer;

Build C:/Temp/nylapps.mak



Top level build

Steps for building your application:

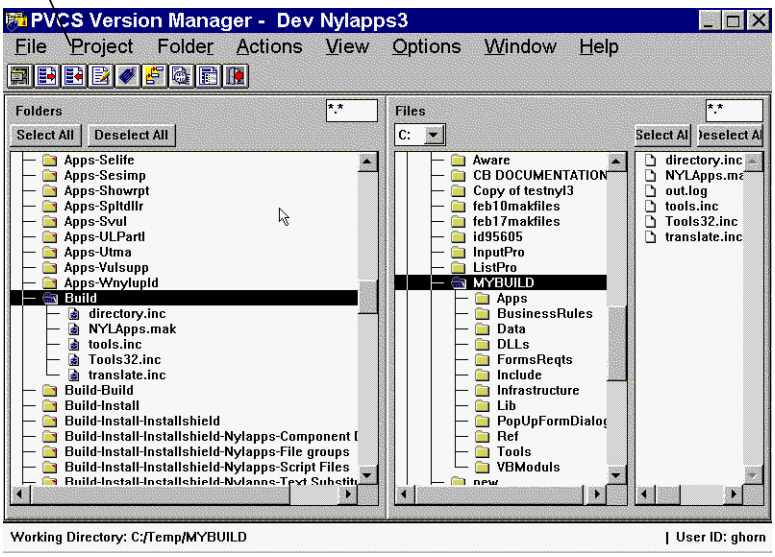
1. Create a folder in the location you would like your application to be built, such as C:\Build or C:\BuildV10. In our example, the folder is in our Temp directory and is a file called "Mybuild."
2. Open the PVCS Version Manager GUI.

3. From the PVCS Project Menu, select "Open project"
4. Choose the PVCS project that contains the source code for the application or module you want to build i.e. DevNylapps3.
5. Within PVCS, on the right hand side of the screen, select the drive in which you created the new folder in Step 1.
6. Open that folder i.e. "Mybuild".
7. "Check- out" the top level make file and the four "Rosetta Stone Files." In our example, the top level make file is called "Nylapps.mak and the "Rosetta Stone Files are:

Directory. inc
Tools.inc
Tools32.inc
Translate.inc

You can check out each file individually by clicking on it and choosing "check out" or hold the control key on your keyboard and highlight all five files and then check them out simultaneously into the folder you created on your hard drive (in our example its called "Temp\Mybuild." You can check-out the files from

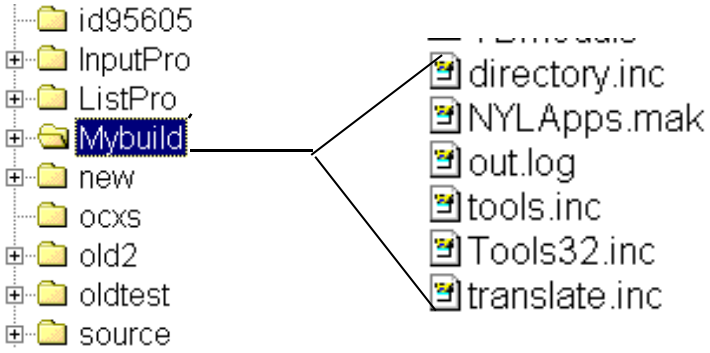
PVCS by using the pull-down menu under “Actions” or clicking on the red arrow that points to the right. This is the “check-out files” arrow.



Check-out” arrow

Select “read only” when checking out these files. Click “OK.”
The five files you checked-out should now be visible on your hard drive in the folder you created in Step 1 i.e. Mybuild.

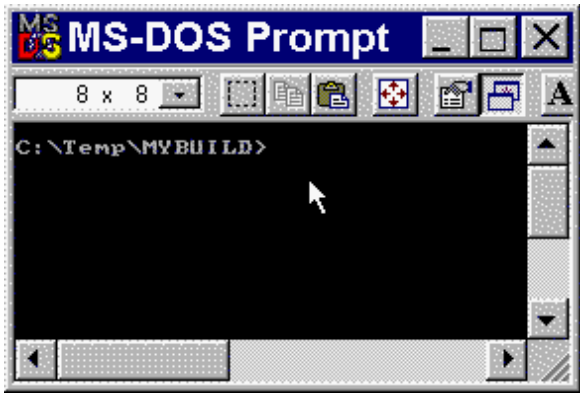
C:\Temp|Mybuild(Windows Explorer view)



“Rosetta Stones”

Now that you have checked out the files that you need to build your application, the next step is to go to the DOS prompt on your PC and type the build command, which is described below.

To run an existing build script in order to build your application, go to the DOS prompt. You must then change to the directory that contains your top level build script. The DOS prompt might look like this:
C:\TEMP\MYBUILD

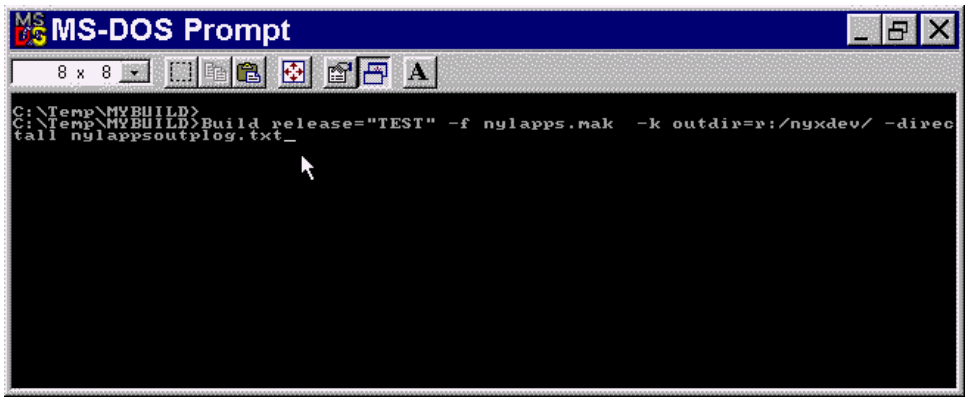


Next, type the **build** comma

nd.

Here is an example of how the build command might look at the DOS promt:

Build release=”TEST” –f nylapps.mak –k outdir=r:/nyxdev/ -directall nylappsoutplog.txt



The build command will happen at the DOS prompt as long as it is pointed at the correct directory. Let’s look at each part of the build command. The first item to type is:

build

This command tells the system that you are invoking configuration builder and are going to use a build script to build an application or component. The next item to type at the command line is:

Release=

This command dictates what will be built…what level of build (Top, Middle, Bottom) or which particular release of an application.

The next item to type will specify the filename i.e.

”TEST”

This specifies which particular version of a script to build. In this case, it is a script named “TEST.” This may be a particular version of an application. The release name is case sensitive.

The next item to type is the name of the file that you wish to build:

Nylapps.mak

In addition, flags or switches are added to the command line.
The entire line at the command prompt might look like this:

Build release=”TEST” –f nylapps.mak –k outdir=r:/nyxdev/ -directall nylappsoutplog.txt

Lets translate the switches in the command line;

“**-f**” specifies the filename of the build script you wish to build.
“**-k**” means keep on going….keep building the script despite errors.
The "-k" switch tells Configuration Builder to keep going even if an error is encountered during a build. Why would we want to use this switch? For example, if a build is being run overnight, we can build as many components as possible without having to restart the build. Someone can return in the morning and read the build log to see what was built and what was not built. If you don’t use a -k switch and a problem is encountered, the build process is terminated at the point of error. This switch truly automates the build so it does not need constant monitoring.

"**outdir**" is the location where the output of the script (your application) gets written (optional). It's a copy of your application placed in a shared file where others can see it.
"**-directall**" is a Configuration Builder command or option(or switch) that delivers the output to the screen and saves it to a file. This file is created automatically. It can be used as a record of the build procedure and is helpful in seeing if any components were not built correctly.
So, in our above example, we are telling Configuration Builder to build the TEST version of nylapps.mak and to keep on building despite possible errors and to place the output in a directory on the r drive called nyxdev.

Type the follwoing:

CD C:\Temp\Mybuild

Hit ENTER.

Your command line should now look like this:
C:\Temp|Mybuild

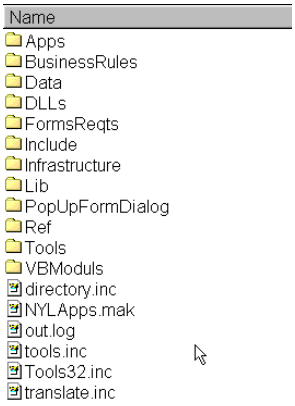
Type the following at the "C" prompt:

Build release="TEST" -f nylapps.mak -k -directall out.log

Hit **Enter.**

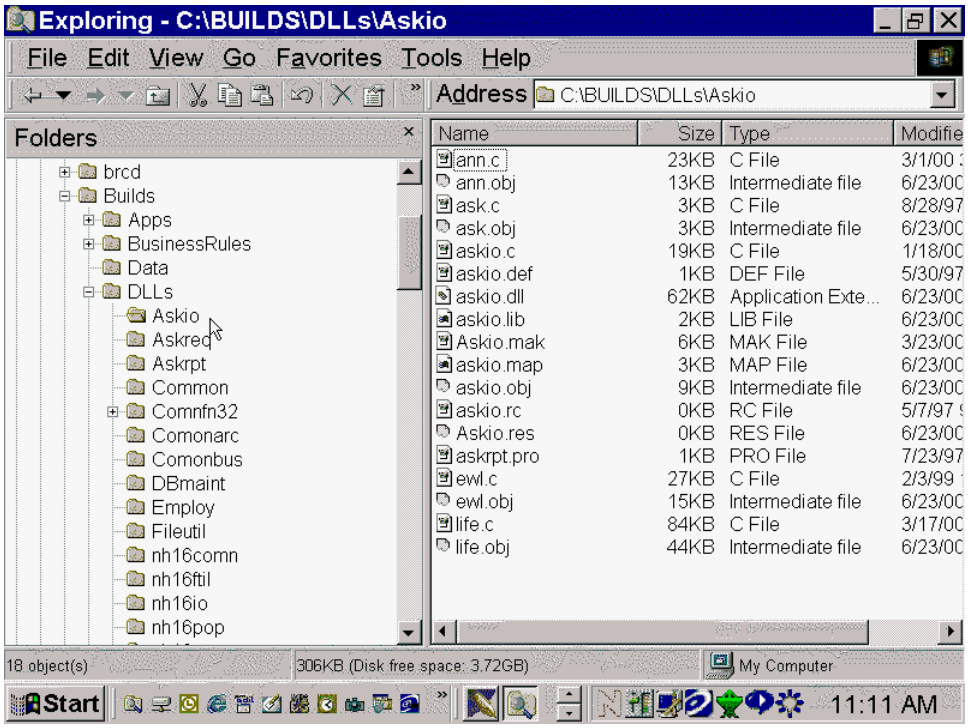
If you typed the parameters correctly, the application is now being built on your system. You have chosen a version of your nylapps application called "TEST." All the files that have you "striped" with the version label "TEST" will be pulled from the PVCS archives. By the way, this parameter is case sensitive, so if the version is called TEST, you cannot

enter it as “test” in lower case letters. In addition, quotes must be used around the name of the release i.e. release=”TEST” The build should result in the following directories and files
being created in your Temp/Mybuild folder.



Procedure for running a low-level configuaration builder script:

The overall build directory structure must be established before you can run a low level build script. The way to accomplish this is to run a top-level build script first. Running the top-level build script will build all of the necessary subdirectories. For example, let’s say you wanted to build a .dll called askio.dll.
The .dll appears in the build hierarchy as follows:


Exploring - C:\BUILDS\DLLs\Askio
File Edit View Go Favorites Tools Help
Address C:\BUILDS\DLLs\Askio
Folders
brcd
Builds
Apps
BusinessRules
Data
DLLs
Askio
Askred
Askrpt
Common
Comnfn32
Comonarc
Comonbus
DBmaint
Employ
Fileutil
nh16comn
nh16ftil
nh16io
nh16pop
Name Size Type Modifie
ann.c 23KB C File 3/1/00
ann.obj 13KB Intermediate file 6/23/00
ask.c 3KB C File 8/28/97
ask.obj 3KB Intermediate file 6/23/00
askio.c 19KB C File 1/18/00
askio.def 1KB DEF File 5/30/97
askio.dll 62KB Application Exte... 6/23/00
askio.lib 2KB LIB File 6/23/00
Askio.mak 6KB MAK File 3/23/00
askio.map 3KB MAP File 6/23/00
askio.obj 9KB Intermediate file 6/23/00
askio.rc 0KB RC File 5/7/97
Askio.res 0KB RES File 6/23/00
askrpt.pro 1KB PRO File 7/23/97
ewl.c 27KB C File 2/3/99
ewl.obj 15KB Intermediate file 6/23/00
life.c 84KB C File 3/17/00
life.obj 44KB Intermediate file 6/23/00
18 object(s)
306KB (Disk free space: 3.72GB)
My Computer
Start
11:11 AM

The path is:
C:\Mybuild\Dlls\Askio

directory.inc

"ROSETTA STONE FILES"

tools.inc
tools32.inc
nylapps.mak
translate.inc
apps
dlls
askio
askio.mak
askio.c
askio.dll.....
include
hd4data.h
d4declar.h
f4flag.h
e4expr.h ..

The point of the above diagram is to show that the askio folder and its files are in a hierarchy. The build engineer sets up this hierarchy. Askio files are located in a subdirectory called askio. This subdirectory is in a directory called Dlls. The Dlls directory is in the *mybuild* directory which the user manually created. As described in the top level build procedure, the build engineer creates a build directory (in DOS or Windows Explorer) and checks out the "Rosetta Stone" files along with the top-level build script. The top level build script in this example is nylapps.mak. Calling the nylapps.mak file will create all of the necessary subdirectories, including the askio subdirectory.

The important thing is that the subdirectories needed by the low-level build script must be created. Generally, shared subdirectories like "include" are created by the top-level build script. If you run the top level build script once, all of the needed subdirectories are built. In order to build a "bottom-level" component, you do not have to run a top level build every time. Once the empty directory structure has been created the first time, you can use low and middle level build scripts successive times.

To build a low-level component:

1. Create needed subdirectories (as described above).
2. At the DOS prompt, change the directory to point to the low-level build component folder that you want to build. In this example, type the following and hit enter:

c:\> cd c:\mybuild\dlls\askio

This will change your directory to:
c:\mybuild\dlls\askio>

At the DOS prompt, execute the build script:
c:\mybuild\dlls\askio>build release="v 8.00A1" -f askio.mak
This will result in building the release labeled "v 8.00A1" of the askio.dll

---

The "mid-level build " builds a subset of the overall system. A mid -level build script in our example is dlls.mak. When the dlls.mak script is executed, the following occurs:

1. The subdirectories for each .dll component are created if they don't already exist.

2. The build script for each component is checked out of PVCS into the appropriate subdirectory or folder. If you look at the contents of dlls.mak you can see that the list of directories and make files are defined in the makeFileList *macro* (see appendix) as shown below:

```
makeFileList = Askio\Askio.mak \
               Askreq\Askreq.mak \
               Askrpt\Askrpt.mak \
               Common\Common.mak \
               Comnfn32\Comnfn32.bld \
               Comonarc\Comonarc.mak \
               Comonbus\Comonbus.mak \
               DBmaint\DBmaint.mak \
               Employ\Employ.mak \
               Fileutil\Fileutil.mak \
               nh16comn\nh16comn.bld \
               nh16io\nh16io.bld \
               nh16ftil\nh16ftil.bld \
               nh16pop\scl_nh16pop.mak \
               nh16prt\nh16prt.bld \
               nh16wfut\nh16wfut.bld \
               nh16wrfu\nh16wrfu.bld \
               Nyllib\Nyllib.mak \
               Nylnetio\Nylnetio.mak \
               Postproc\scl_Postproc.mak \
               Sehtwt\Sehtwt.mak \
               Sestatbl\Sestatbl.mak \
               Upload\Upload.mak \
               Uw\Uw.mak \
               Wfutils\Wfutils.mak \
               Wrapfunc\Wrapfunc.mak \
               Wep\Wep.mak \
```

Nylio\Nylio.mak

3. As described in the low-level build description, you cannot simply run this mid-level script without having run, at some earlier point, the top-level script in the current build directory. Directories above the mid-level script must be created. The "include" directory is one example. The reason why the mid-level script has the same problem as the low-level script is because the mid-level script ends up actually calling the low level scripts specified by the makeFileList macro. In our example, the Dlls build script will call the following low-level makefiles:

Askio.mak Askreq.mak Askrpt.mak Common.mak Comnfn32.bld Comonarc.mak Comonbus.mak DBmaint.mak Employ.mak Fileutil.mak nh16comn.bld nh16io.bld nh16ftil.bld scl_nh16pop.mak nh16prt.bld nh16wfut.bld nh16wrfu.bld Nyllib.mak Nylnetio.mak scl_Postproc.mak Sehtwt.makSestatbl.makUpload.mak Uw.mak Wfutils.mak Wrapfunc.mak Wep.mak Nylio.mak

Note that a higher level make file passes its macro definitions down to the lower levels. When you execute the dlls.mak file with a command line like...

C:\mybuild\dlls\>build release="v 9.00A3" -f dlls.mak -k

Configuration Builder will build all of the "v 9.00A3" components that are listed in Dlls.mak. The *release* macro was defined once and is then passed to each of the lower level make files.
The -f switch (flag, see appendix)) tells Configuration Builder that the next string is the filename of the build script that is to be executed.

The "-k" switch tells Configuration Builder to keep going even if an error is encountered during a build. Why would we want to use this switch?

For example, if a build is being run overnight, we can build as many components as possible without having to restart the build. Someone can return in the morning and read the build log to see what was built and what was not built. If you don't use a -k switch and a problem is encountered, the build process is terminated at the point of error. This switch truly automates the build so it does not need constant monitoring.

One problem that might prevent a build from completing even with the-k switch set, is a Visual Basic .VBP file (version 4.0 or earlier) that opens the visual basic development application. An error may display a dialogue box that requires an active user to click it before the build can continue.

Following is an example of an actual build script. It has been created by the Softcraft Lab and can be called by you at the DOS prompt to run and build an application. It can only be called by you if you have all the necessary archives, build scripts and files.
If you were to open the file called "NYLApps.mak" in a text editor, this is what you would see. The words that are highlighted indicate places where the person building the application can substitute their particular files but still use the same script skeleton. You can also see the comments that were added by the build engineer each time the script was checked out of PVCS. You could cut and paste this script into your own .mak file, changing whatever highlighted filenames or paths are particular to your application.

```
#----------------------------------------------------------------------
# $Workfile:   NYLApps.mak  $
#
```

```
# $Logfile:  R:/nyxdev/PVCS/pvcsarch/nylapp3/Build/NYLApps.mak_v
$
#
#   Description:       This PVCS Configuration Builder creates, if needed,
                                the top level NYLAPPS directory shell,
                                populates it with the  make files specified,
                                and executes those make files  to build the
                                NYLAPPs system.
#
#
#   Note(s) :       [1] To build production code:
#                   build production="" release="v ?" -f Nylapps.mak
#
#                [2] To build debug code:
#                   build  release="v ?" -f Nylapps.mak
#
#                [3] Any flags or macros that you specify on the command
#                   line will be passed down to the lower-level make
#                   files.
#
#
#   Reference(s):
#
#  $Revision:   1.8  $     $Date:    Jan 18 2000 10:50:00  $     $Author:
t15bh13  $
#-------------------------------------------------------------------------
#
# $Log:   R:/nyxdev/PVCS/pvcsarch/nylapp3/Build/NYLApps.mak_v  $
#
#     Rev 1.8   Jan 18 2000 10:50:00   t15bh13
#  Added Ref as required subdir, changed
#  VBModules to VBModuls
```

```
#
#
#    Rev 1.7   Dec 07 1999 09:29:28   t15bdiq
#  the repository directory was changed to match
#  the NYL location
#
#    Rev 1.6   Oct 11 1999 14:55:12   mspaner
#  changed repos. dir to r:\..\nylapp3
#
#
#    Rev 1.5   Sep 28 1999 17:10:40   kroussea
#  Removed reference to data.mak
#
#    Rev 1.4   Sep 28 1999 15:53:52   kroussea
#  added data, Include, Lib, VBModules to misc dir list
#
#    Rev 1.3   Jun 23 1999 15:34:58   mspaner
#  working file.
#
#    Rev 1.2   Jun 23 1999 08:36:48   mspaner
#  changed .mav to .mak_v
#
#    Rev 1.1   Jun 17 1999 14:26:38   mspaner
#  changed .path from .mak_v to .mav for testing
#
#    Rev 1.0   Jun 17 1999 13:55:04   mspaner
#  top level config. builder file
#
#

#
# The following  macro definitions control both the creation of the
```

```
#directory shell and the execution  of the make files. Use
#miscDirectoryList to create miscellaneous  build directories at this level.
#Use subDirectoryList to specify any common subdirectory structure
#found below each directory listed in makeFileList.
#
# If you have to make files in lower directories, repeat this design at the
# next lower level so that the next level make file creates the directory
shell
# and executes the make files below it.

# The next line of the build script shows where the repository that holds
the # files is located.  The repository is the top level directory of the PVCS
# archives  for the project.  You can change this address to point to your
# project's  particular repository.

repository = r:\nyxdev\pvcs\pvcsarch\nylapp3

mainDir = $(repository)

#  Next up is the the miscDirectoryList macro.  This macro defines the
#  miscellaneous directories that should be created as part of the build.
 # They do not contain lower level build scripts.  You may substitute your
# own list.

miscDirectoryList = Data Include Lib VBModuls Ref

# Next is the subdirectory list.
# Use this list  to specify any common subdirectory structure found below
# each directory listed in makeFileList.

subDirectoryList =bin \
                                                text\
```

```
                                                    src\


# The makefilelist macro contains the next level of directories and build
# scripts that must be executed. Lower level make files may also contain
#  makefilelist which would create an even lower level structure....

makeFileList =                  Apps\Apps.mak \
                                                    BusinessRules\BusinessRules.mak
\
                                                    DLLs\DLLs.mak \
                                                    FormsReqts\FormsReqts.mak \
                                                    Infrastructure\Infrastructure.mak
\

PopUpFormDialog\Popupformdialog.mak \
                                                    Tools\Tools.mak

# Next, include a Rosetta stone file:
.INCLUDE .\translate.inc


# Include another Rosetta stone file
.INCLUDE .\directory.inc

# Include another Rosetta stone file
.INCLUDE .\tools.inc
```

```
# Next, we need the PATH directive.**(see definition below)

.PATH.mak_v=$(mainDir)\*

# We build the target called  'all' by going to each  directory and building
# all of the scripts in the makefilelist macro.

all: $(makeFileList)
  %ForEach file in $(makeFileList)
    (silent) %Cd $[Directory,$(file)]
    build $(_FlagsMacros) -f $[Base, $(file)].mak
    (silent) %Cd ..
  %EndFor


# We'll use the .PROLOG reserved target to start the error log (if any) and
# to modify the environment.

# First we make all of the directories listed in makefilelist, subdirectorylist
# and miscdirectorylist.

.PROLOG:
  %ForEach file in $(makeFileList)
    (Silent)%Set directory = $[Directory, $(file)]
    %If %Dir( $(directory) )
    %Else
                                (Silent)mkdir $(directory)
    %EndIf
    %ForEach subdirectory in $(subDirectoryList)
      %If %Dir( $(directory)\$(subdirectory) )
                                %Else
        (Silent)mkdir $(directory)\$(subdirectory)
```

```
                                %EndIf
    %EndFor
  %EndFor
  %ForEach directory in $(miscDirectoryList)
    %If %Dir( $(directory) )
    %Else
                                (Silent)mkdir $(directory)
    %EndIf
  %EndFor
```

Here is a sample of a build script that shows the paths to find the necessary files. The locations highlighted in orange are the areas the individual build engineers or developers can modify to locate specific files for the project they want to build.

```
#-------------------------------------------------------------------------
#  $Workfile $
#
#  $Logfile$
#
                                # Description:
                                    This PVCS Configuration Builder
                                    fragment defines the
#                   common search paths for each file type used in the
#                   system.
#
#   Note(s) :       You should add locally used search paths to the local
#                   make files to reduce overall search time!
#
```

```
#   Reference(s):
#
#  $Revision$     $Date$     $Author$
#-------------------------------------------------------------------------
#
# $Log$
#
#
#
# The following are the directory declarations. Notice that the first
# directories searched are all subsumed under the current subdirectory.
# After that, the search paths explored are all in the repository.

mainSrc = $(mainDir)
mainInc = $(repository)/Include
mainLib = $(repository)/Lib
vbModls = $(repository)/vbModuls
.PathSearch Strict
#.PATH.bld_v=$(mainSrc)
.PATH.bld=.
.PATH.c_v=$(mainSrc)
.PATH.c=.
.PATH.cpp_v=$(mainSrc)
.PATH.cpp=.
.PATH.clw_v=$(mainSrc)
.PATH.clw=.
.PATH.odl_v=$(mainSrc)
.PATH.odl=.
.PATH.mak_v=$(mainSrc);$(mainSrc)\*
.PATH.mak=.
.PATH.bmp_v=$(mainSrc)
.PATH.bmp=.
```

```
.PATH.txt_v=$(mainSrc)
.PATH.txt=.
.PATH.ico_v=$(mainSrc)
.PATH.ico=.
.PATH.def_v=$(mainSrc)
.PATH.def=.
.PATH.dlg_v=$(mainSrc)
.PATH.dlg=.
.PATH.exp_v= $(mainSrc);$(mainInc)
.PATH.exp=.;..\include;..\..\include;$(include)
.PATH.h_v= $(mainSrc);$(mainInc)
.PATH.h=.;..\include;..\..\include;$(include)
.PATH.pro_v=$(mainSrc);$(mainInc)
.PATH.pro=.
.PATH.rc_v=$(mainSrc)
.PATH.rc=.
.PATH.rc2_v=$(mainSrc)
.PATH.rc2=.
.PATH.res_v=$(mainSrc)
.PATH.res=.
.PATH.tlk_v=$(mainSrc)
.PATH.tlk=.
.PATH.obj_v=$(mainSrc);$(mainLib)
.PATH.obj=.;..\Lib;..\..\Lib
.PATH.lib_v=$(mainSrc);$(mainLib)
.PATH.lib=.;..\Lib;..\..\Lib
#VB file types
.PATH.bas_v=$(mainSrc)
.PATH.bas=.
.PATH.cls_v=$(mainSrc);$(vbModls)
.PATH.cls=.;..\vbModuls;..\..\vbModuls
.PATH.ctl_v=$(mainSrc)
```

```
.PATH.ctl=.
.PATH.ctx_v=$(mainSrc)
.PATH.ctx=.
.PATH.frm_v=$(mainSrc)
.PATH.frm=.
.PATH.frx_v=$(mainSrc)
.PATH.frx=.
.PATH.pag_v=$(mainSrc)
.PATH.pag=.
.PATH.pgx_v=$(mainSrc)
.PATH.pgx=.
.PATH.vbp_v=$(mainSrc)
.PATH.vbp=.
#-------------------------------------------------------------------------------------------------------------
```

Here is an example of a mid level build script. It makes all the Dlls.

```
#-----------------------------------------------------------------------
#  $Workfile:   DLLs.mak  $
#
#  $Logfile:   R:/nyxdev/PVCS/pvcsarch/nylapp3/DLLs/DLLs.mak_v  $
#
#   Description:  This PVCS Configuration Builder creates, if needed,
#                      the DLLs directory shell, populates it with the
#                      make files specified, and executes those make files to
#                      build the BusinessRules components of the NYLAPPS
#                                                  system.
#
#
#   Note(s) :   [1] To build production code:
#                     build production="" release="v ?" -f DLLs.mak
```

```
#
#               [2] To build debug code:
#                   build  release="v ?" -f DLLs.mak
#
#               [3] Any flags or macros that you specify on the command
#                   line will be passed down to the lower-level make
#                   files.
#
#
#   Reference(s):
#
#  $Revision:   1.19  $     $Date:   Mar 23 2000 10:12:34  $     $Author:
# t15bdiq  $
#-------------------------------------------------------------------------
#
# $Log:   R:/nyxdev/PVCS/pvcsarch/nylapp3/DLLs/DLLs.mak_v  $
#
#    Rev 1.19   Mar 23 2000 10:12:34   t15bdiq
#  commented out nh32comn temporarily
#
#    Rev 1.18   Mar 20 2000 09:22:58   t15bdiq
#
#
#    Rev 1.17   Mar 02 2000 09:45:46   t15bdiq
#  uncommented nylio and nh32comn.
#  deleted nylsecur since it is not in r:\nyxdev\bin\se
#
#    Rev 1.16   Dec 16 1999 14:56:22   t15bdiq
#  SCL mods for 12-16-1999
#
#    Rev 1.27   Dec 15 1999 10:47:12   mspaner
#  added nh16wfut.bld to the make file list
```

```
#
#    Rev 1.26   Dec 15 1999 10:21:12   mspaner
#  added nh16prt.bld to the makefile list
#
#    Rev 1.25   Dec 14 1999 16:30:42   mspaner
#  added nh16wrfu to the makefile list
#
#    Rev 1.24   Dec 14 1999 16:03:24   mspaner
#  added nh16io,bld to makefile list. changed order
#  of makefiles (just to alphabetize)
#
#    Rev 1.23   Dec 14 1999 15:20:44   mspaner
#  added nh16comn to the makefile list
#
#    Rev 1.22   Dec 14 1999 14:34:28   mspaner
#  added nh16ftil to makefile list. took off nh32comn from ]
#  the list since it is not working right.
#
#    Rev 1.21   Dec 14 1999 14:26:42   mspaner
#  added nh16ftil to the makefile list
#
#    Rev 1.15   Dec 03 1999 14:30:34   t56az0n
#  SCL 12-3-1999
#
#    Rev 1.20   Nov 30 1999 16:11:46   mspaner
#  added wep.mak to makefile list
#
#    Rev 1.19   Nov 30 1999 14:53:56   mspaner
#  added comnfn32.bld to makefile list.
#  added ability to handle .bld extensions for
#  config. builder make files.
#
```

```
#    Rev 1.18   Nov 29 1999 13:47:08   mspaner
#  added common.mak to the makefile list
#
#    Rev 1.17   Nov 24 1999 10:04:52   mspaner
#  fixed nh16pop line, deleted askio duplicate line
#
#    Rev 1.16   Nov 24 1999 10:01:12   mspaner
#  added nh16pop to the makefile list
#
#    Rev 1.15   Nov 11 1999 15:52:52   mspaner
#  consolidated scl and nyl DLLs.mak files
#
#    Rev 1.14   Nov 05 1999 10:33:24   t15bdiq
#  removed Memgl.mak from list. memgl is an exe not dll.
# It should be built under tools
#
#    Rev 1.13   Oct 12 1999 13:30:52   t15bdiq
#  put nylsecur on to-do list.
#
#    Rev 1.12   Oct 12 1999 13:24:50   t15bdiq
#  fixed repos. to r:\...\nylapp3\
#
#    Rev 1.11   Oct 12 1999 12:30:12   t15bdiq
#  comonfn32.dll is cpp. comment it out.
#
#    Rev 1.8   Oct 11 1999 16:17:06   mspaner
#  moved commented out wep to the bottom of list.
#
#    Rev 1.7   Oct 11 1999 15:47:04   mspaner
#  commented out un-tested or incomplete make files.
#  did this to clarify testing status.
#
```

```
#     Rev 1.6   Oct 11 1999 11:45:14   mspaner
#  fixed postproc.mak to scl_postproc.mak
#
#     Rev 1.5   Sep 28 1999 15:50:06   kroussea
#   Fixed repository and maindir
#
#     Rev 1.4   Aug 11 1999 08:18:58   mspaner
#  added Memgl to makefilelist
#
#     Rev 1.3   Jul 22 1999 11:40:48   mspaner
#  scl- fixed repository path
#
#     Rev 1.2   Jul 06 1999 15:48:14   kroussea
#  Changed relative path for .inc files to look up a level in the dir tree
#
#
#     Rev 1.1   Jun 24 1999 09:24:18   t15bhpf
#  config builder script. tested at softcraft.
#
#     Rev 1.0   Jun 17 1999 13:46:18   mspaner
#  Config Builder .mak file
#
#


#
# The following macro definitions control both the creation of the
directory
# shell and the execution of the make files. Use miscDirectoryList to
#  create miscellaneous build directories at this level. Use
subDirectoryList
# to specify any common subdirectory structure found below each
# directory listed in makeFileList.
```

```
#
# If you have to make files in lower directories, repeat this design at the
# next lower level so that the next level make file creates the directory
shell
# and executes the make files below it.
#
#
repository = r:\nyxdev\pvcs\pvcsarch\nylapp3
mainDir = $(repository)\Dlls
miscDirectoryList =
subDirectoryList =
makeFileList =          Askio\Askio.mak \
                        Askreq\Askreq.mak \
                        Askrpt\Askrpt.mak \
                        Common\Common.mak \
                        Comnfn32\Comnfn32.bld \
                        Comonarc\Comonarc.mak \
                        Comonbus\Comonbus.mak \
                        DBmaint\DBmaint.mak \
                        Employ\Employ.mak \
                        Fileutil\Fileutil.mak \
                        nh16comn\nh16comn.bld \
                        nh16io\nh16io.bld \
                        nh16ftil\nh16ftil.bld \
                        nh16pop\scl_nh16pop.mak \
                        nh16prt\nh16prt.bld \
                        nh16wfut\nh16wfut.bld \
                        nh16wrfu\nh16wrfu.bld \
                        Nyllib\Nyllib.mak \
                        Nylnetio\Nylnetio.mak \
                        Postproc\scl_Postproc.mak \
                        Sehtwt\Sehtwt.mak \
```

```
                Sestatbl\Sestatbl.mak \
                Upload\Upload.mak \
                Uw\Uw.mak \
                Wfutils\Wfutils.mak \
                Wrapfunc\Wrapfunc.mak \
                Wep\Wep.mak \
                Nylio\Nylio.mak
                #nh32comn\nh32comn.bld

.INCLUDE ..\translate.inc

.PATH.bld_v=$(mainDir)\*
.PATH.mak_V=$(mainDir)\*

all: $(makeFileList)
  %ForEach file in $(makeFileList)
    (silent) %Cd $[Directory,$(file)]
    build $(_FlagsMacros) -f $[Base, $(file)].$[Extension, $(file)]
    (silent) %Cd ..
  %EndFor

#
# We'll use the .PROLOG reserved target to start the error log (if any) and
# to modify the environment.
#
.PROLOG:
  %ForEach file in $(makeFileList)
    (Silent)%Set directory = $[Directory, $(file)]
    %If %Dir( $(directory) )
    %Else
      (Silent)mkdir $(directory)
    %EndIf
```

```
    %ForEach subdirectory in $(subDirectoryList)
      %If %Dir( $(directory)\$(subdirectory) )
      %Else
        (Silent)mkdir $(directory)\$(subdirectory)
      %EndIf
    %EndFor
  %EndFor
  %ForEach directory in $(miscDirectoryList)
    %If %Dir( $(directory) )
    %Else
      (Silent)mkdir $(directory)
    %EndIf
  %EndFor

.INCLUDE ..\directory.inc

.INCLUDE ..\tools.inc

#-------------------------------------------------------------------------------
```

Here is an example of a Visual Basic build script:
It's called Investment.mak

```
#-------------------------------------------------------------------------------
#
#     $Workfile:   Investment.mak  $
#
#     $Logfile:
```

```
#
 # R:/nyxdev/PVCS/pvcsarch/nylapp3/Apps/LifeApp/investment/Investment.mak_v  $
 #
 #      Description:    This PVCS Configuration Builder file populates the Infrastructure-Interfaces
 #                      directory and builds all buildable components in that directory
 #
 #      Note(s):
 #
 #      Reference(s):
 #
 #      $Revision:   1.3  $     $Date:   24 Jan 2000 11:17:40  $        $Author:   t15bdiq  $
 #-------------------------------------------------------------------------------
 #
 #      $Log:   R:/nyxdev/PVCS/pvcsarch/nylapp3/Apps/LifeApp/investment/Investment.mak_v  $
 #
 #    Rev 1.3   24 Jan 2000 11:17:40   t15bdiq
 #  removed %exit and added regsrv32 switches
 #
 #    Rev 1.2   Jan 19 2000 14:29:42   t15bh13
 #  Changed vbmodules to vbmoduls
 #
 #    Rev 1.1   Jan 12 2000 10:25:06   t15bh13
 #  Added NoShell operation line modifier,
 #  removed quotes from vb5exe, removed
 #  recursive call to copyfiles
 #
 #    Rev 1.0   22 Dec 1999 14:30:00   T15BH13
 #
 #
 #    Rev 1.2   Nov 11 1999 15:32:48   kroussea
 #  Template A
 #
 #    Rev 1.1   Oct 21 1999 10:27:34   kroussea
 #  hard coded out dir - will need to change
 #
 #
 #    Rev 1.0   Oct 15 1999 14:29:30   kroussea
 #  Initial revision
 #
 #
 #
 repository = r:\nyxdev\pvcs\pvcsarch\Nylapp3
 mainDir = $(repository)\Apps\LifeApp\investment
```

```
#vb5Exe = c:\progra~1\devstu~1\vb\vb5.exe

.INCLUDE ..\..\..\translate.inc

Target =Investment.ocx

vbProj =Investment.vbp
vbForms=
vbCtrls=Investment.ctl
vbPPags=
vbModls=..\..\..\vbmoduls\EventID.bas ..\..\..\vbmoduls\Mdeclare.bas
vbClass=CinvestPresentation.cls CInvestValidation.cls
vbCmpat=..\..\..\Ref\$(Target)
vbObjs =
vbRefs =
vbVbxs =
vbDecls=
vbBins =Investment.ctx

Targ: $(Target)
res=

$(Target): $(vbProj) $(vbForms) $(vbCtrls) $(vbPPags) $(vbModls) $(vbClass) $(vbObjs)
$(vbRefs) $(vbVbxs) $(vbDecls) $(vbBins) $(res)
	%If ("$(vbCmpat)" != "")
	   %If (%Exists($(vbCmpat)))
	      (noshell)(silent) attrib -r $(vbCmpat)
	      del $(vbCmpat)
	   %EndIf
	   (noshell)(silent) get -q $(getFlags) $(repository)\ref\$(Target)_v($(vbCmpat))
	%EndIf
	%If ("$(vbCmpat)" == "") || (%Exists($(vbCmpat)))
	   (noshell) $(vb5Exe) /make "$(vbProj)" /out "$(_TargRoot)Build.log" /outdir "$(_Cwd)"
	   %Do CopyFiles
	%Else
	   #May want more than just an error message.  perhaps a log or a retcode?
	   @Echo Error, could not find compatibility file $(vbCmpat).
	%EndIf

..\..\..\vbmoduls\EventID.bas:
    (noshell)(silent) get -q $(getFlags) $(repository)\vbmoduls\EventID.bas_v($(_Target))

..\..\..\vbmoduls\Mdeclare.bas:
```

```
    (noshell)(silent) get -q $(getFlags) $(repository)\vbmoduls\MDeclare.bas_v($(_Target))


CopyFiles:
      %If %exists("$(Target)")
         xcopy32 "$(Target)" c:\se
         regsvr32 /s /c "c:\se\$(Target)"
      %Endif

.PROLOG:

.INCLUDE ..\..\..\tools.inc
.INCLUDE ..\..\..\directory.inc

#-------------------------------------------------------------------------------------------------------------------
```

Here is a build script for MS Visual Basic 4 files.
It's called NYLAZ.mak

```
#-------------------------------------------------------------------------------
#
#      $Workfile:   nylaz.mak  $
#
#      $Logfile:   R:/nyxdev/PVCS/pvcsarch/nylapp3/Apps/Nylaz/Nylaz.mak  $
#
#      Description:    This PVCS Configuration Builder file populates the Infrastructure-Interfaces
#                      directory and builds all buildable components in that directory
#
#      Note(s):
#
#      Reference(s):
#
#      $Revision:   1.4  $     $Date:   Jan 28 2000 16:15:28  $        $Author:   t15bh13  $
#-------------------------------------------------------------------------------
#
#      $Log:   R:/nyxdev/PVCS/pvcsarch/nylapp3/Apps/Nylaz/Nylaz.mak  $
#
#    Rev 1.4   Jan 28 2000 16:15:28   t15bh13
#  Fixed Copyfiles , vb build line does not
#  accept path to build to.
#
#    Rev 1.3   Jan 28 2000 10:56:36   t15bh13
```

```
#  fixed regserver switch
#
#    Rev 1.2   Jan 28 2000 10:52:54   t15bh13
#  fixed targed added copyfiles
#
#    Rev 1.1   Jan 12 2000 10:29:04   t15bh13
#  Added NoShell operation line modifier,
#  removed quotes from vb5exe, removed
#  recursive call to copyfiles
#
#    Rev 1.0   22 Dec 1999 14:34:58   T15BH13
#
#
#    Rev 1.2   Nov 11 1999 15:32:48   kroussea
#  Template A
#
#    Rev 1.1   Oct 21 1999 10:27:34   kroussea
#  hard coded out dir - will need to change
#
#
#    Rev 1.0   Oct 15 1999 14:29:30   kroussea
#  Initial revision
#
#
# #
# #


#
#
#
repository = r:\nyxdev\pvcs\pvcsarch\Nylapp3
mainDir = $(repository)\Apps\Nylaz
#vb5Exe = c:\progra~1\devstu~1\vb\vb5.exe

.INCLUDE ..\..\translate.inc

Target =Nylaz.exe

vbProj =Nylaz.vbp
vbForms=Q1.FRM  Q16_19.FRM  Q1OCC.FRM  Q2.FRM  Q3_6.FRM  Q7.FRM  Q11.FRM
Q8_10.FRM REPLACE.FRM
vbCtrls=
```

```
vbPPags=
vbModls=COMMON.BAS CHKRULES.BAS POPUPS.BAS REPLACE.BAS SSOCX.BAS HELPER.BAS
vbClass=
vbCmpat=..\..\Ref\$(Target)
vbObjs =
vbRefs =
vbVbxs =
vbDecls=
vbBins =Q1.frx Q16_19.frx Q1OCC.frx Q2.frx Q3_6.frx Q7.frx Q11.frx Q8_10.frx REPLACE.frx

Targ: $(Target)
res=

$(Target): $(vbProj) $(vbForms) $(vbCtrls) $(vbPPags) $(vbModls) $(vbClass) $(vbObjs) $(vbRefs) $(vbVbxs) $(vbDecls) $(vbBins) $(res)
	%If ("$(vbCmpat)" != "")
	   %If (%Exists($(vbCmpat)))
	      (noshell)(silent) attrib -r $(vbCmpat)
	      del $(vbCmpat)
	   %EndIf
	   (noshell)(silent) get -q $(getFlags) $(repository)\ref\$(Target)_v($(vbCmpat))
	%EndIf
	%If ("$(vbCmpat)" == "") || (%Exists($(vbCmpat)))
	   (noshell) $(vb4Exe) /make $(vbProj)
	   %Do CopyFiles
	%Else
	   #May want more than just an error message.  perhaps a log or a retcode?
	   @Echo Error, could not find compatibility file $(vbCmpat).
	%EndIf

CopyFiles:
	%If %exists(c:\se\$(Target))
	   xcopy32 c:\se\$(Target)
	%Endif


.PROLOG:

.INCLUDE ..\..\tools.inc
.INCLUDE ..\..\directory.inc
```

Here is a build script for MS Visual Basic 5 files.
It's called q1m.mak

```
#-------------------------------------------------------------------------------
#
#      $Workfile:   q1m.mak  $
#
#      $Logfile:   R:/nyxdev/PVCS/pvcsarch/nylapp3/Apps/LifeApp/Q1/q1m.mak_v  $
#
#      Description:    This PVCS Configuration Builder file populates the Infrastructure-Interfaces
#                      directory and builds all buildable components in that directory
#
#      Note(s):
#
#      Reference(s):
#
#      $Revision:   1.3  $     $Date:   24 Jan 2000 11:35:08  $        $Author:   t15bdiq  $
#-------------------------------------------------------------------------------
#
#      $Log:   R:/nyxdev/PVCS/pvcsarch/nylapp3/Apps/LifeApp/Q1/q1m.mak_v  $
#
#    Rev 1.3   24 Jan 2000 11:35:08   t15bdiq
#  removed %exit and added regsrv32 switches
#
#    Rev 1.2   Jan 19 2000 14:28:32   t15bh13
#  Changed vbmodules to vbmoduls
#
#    Rev 1.1   Jan 12 2000 10:42:54   t15bh13
#  Added NoShell operation line modifier,
#  removed quotes from vb5exe, removed
#  recursive call to copyfiles
#
#    Rev 1.0   22 Dec 1999 14:57:16   T15BH13
#
#
#    Rev 1.2   Nov 11 1999 15:32:48   kroussea
#  Template A
#
#    Rev 1.1   Oct 21 1999 10:27:34   kroussea
#  hard coded out dir - will need to change
```

```
#
#
#    Rev 1.0   Oct 15 1999 14:29:30   kroussea
#  Initial revision
#
#
# #
# #


#
#
#
repository = r:\nyxdev\pvcs\pvcsarch\Nylapp3
mainDir = $(repository)\Apps\LifeApp\Q1
#vb5Exe = c:\progra~1\devstu~1\vb\vb5.exe

.INCLUDE ..\..\..\translate.inc

Target =Q1M.ocx

vbProj =q1m.vbp
vbForms=
vbCtrls=Q1m.ctl
vbPPags=
vbModls=..\..\..\vbmoduls\MDeclare.bas ..\..\..\vbmoduls\EventID.bas
vbClass=q1validation.cls q1presentation.cls
vbCmpat=..\..\..\Ref\$(Target)
vbObjs =
vbRefs =
vbVbxs =
vbDecls=
vbBins =Q1m.ctx

Targ: $(Target)
res=

$(Target): $(vbProj) $(vbForms) $(vbCtrls) $(vbPPags) $(vbModls) $(vbClass) $(vbObjs)
$(vbRefs) $(vbVbxs) $(vbDecls) $(vbBins) $(res)
        %If ("$(vbCmpat)" != "")
           %If (%Exists($(vbCmpat)))
              (noshell)(silent) attrib -r $(vbCmpat)
              del $(vbCmpat)
```

```
        %EndIf
        (noshell)(silent) get -q $(getFlags) $(repository)\ref\$(Target)_v($(vbCmpat))
    %EndIf
    %If ("$(vbCmpat)" == "") || (%Exists($(vbCmpat)))
        (noshell) $(vb5Exe) /make "$(vbProj)" /out "$(_TargRoot)Build.log" /outdir "$(_Cwd)"
        %Do CopyFiles
    %Else
        #May want more than just an error message.  perhaps a log or a retcode?
        @Echo Error, could not find compatibility file $(vbCmpat).
    %EndIf

..\..\..\vbmoduls\EventID.bas:
    (noshell)(silent) get -q $(getFlags) $(repository)\vbmoduls\EventID.bas_v($(_Target))

..\..\..\vbmoduls\Mdeclare.bas:
    (noshell)(silent) get -q $(getFlags) $(repository)\vbmoduls\MDeclare.bas_v($(_Target))


CopyFiles:
    %If %exists("$(Target)")
        xcopy32 "$(Target)" c:\se
        regsvr32 /s /c "c:\se\$(Target)"
    %Endif

.PROLOG:

.INCLUDE ..\..\..\tools.inc
.INCLUDE ..\..\..\directory.inc
```

Some additional notes about Visual Basic builds.

The build script for putting together Visual Basic modules is different from the scrips for C modules. There are also differences within the VB build scripts, depending on whether a module being built is from Visual Basic version 4.0 or 5.0. Here is a list of some of the macros you'll find in the visual basic build scripts, and their associated definitions.

vbProj - This macro contains the filename of the .vbp (VB project) file. The project file can be thought of as Visual Basic's internal makefile. It contains a list of the next level of source dependencies

vbForms - This macro contains a space delimited list of the .frm (VB Form) file(s) referred to in the project file in a 'Form=' line. This is the bread and butter of a dialog-based application.

vbCtrls - This macro lists the control (.ctl) file(s) that are required for this project. There will be a corresponding line in the project file UserControl= for each .ctl file. Control files are principly found inprojects that build an ocx (ActiveX Control). (VB5 and later only)

vbPPags - This macro can be used to list .pag (Property Page) files. They might be used to provide an interface to the settings of a control or object at design time.

vbBins - Each of the previous three types of source files are text files. Any binary data needed to create the user interface (icons, bitmaps, etc.) are stored in binary companion files with the extensions .frx, .ctx and .pgx respectively. It is important to populate this macro appropriately. Unlike the text files there is no direct reference to these files in the project files. They will be referenced only in the .frm, .ctl or .pag that uses them. If vb is 'unable to load myform.frm' for example, suspect that you may have missed an .frx file that VB needs to properly display the form.

No set up of a macro for some of the newer source file types for VB 5 and 6 such as ActiveX document (.dob) files or Active Designer (.dsr) files, but such could be done easily.

vbModls - Here we find a list of the VB standard module (.bas) files. Note that here we may start to see relative path filenames to a common directory (*repository*\vbmoduls) to find these files.

vbClass - Similar to standard modules, class modules (.cls files) contain code for exposed code

objects. This is where COM objects are likely to be found. Again we will see many .cls files that are common to multiple projects and so will be found in the vbmoduls archive folder.

vbObjs, vbRefs, vbVbxs - These are currently unused, but would be used to specify ocxs, vbxs and ActiveX dlls that a project is dependent on. The original thought behind the template was to be as complete as possible in explicitly outlining the binary dependencies of each component. However time constraints forced a decision to forgo for the time being whatever benefit these explicit requirements might give. They would be the key to forcing recompilation of components based on newer lower level components. Instead a recommendation was made to enforce binary compatibility in all components whenever possible. All of the 32bit components we worked with were COM components and forcing binary compatibility takes advantage of an unchanging interface to allow ocxs or dlls dependent on a low level component to continue to function without recompilation.

vbDecls – As with the OLE/ActiveX/COM binary dependencies above, this is currently unused. It was meant to be used to establish dependencies created by Declare statements in the VB code files. The Declare statement allows a VB component to use a function exported from a standard dll (or exe) such as might be created using C or C++. This is one way for VB programers to gain access directly to the Windows or Win32 APIs for example.

vbCmpat - In order to build a version compatible COM component, one must have a(n earlier) reference copy of the component to compare the resultant interface to. VB uses the same GUIDs if the interface remains compatible. The first step in building the target is to extract a copy of the appropriate binary file from the *repository*\ref archive folder.

These Macros are defined in tools.inc

vb4Exe - This macro supplies the fully qualified path to the VB 4 vb.exe file

vb5Exe - Similarly, this macro supplies the fully qualified path to the VB 5 vb5.exe file

vb5rcExe - Fully qualified path to the 32 bit resource compiler rc.exe for VB 5.

#This is the target line that defines the dependency of Target on each of the types of files described above

```
$(Target): $(vbProj) $(vbForms) $(vbCtrls) $(vbPPags) $(vbModls) $(vbClass) $(vbObjs) $(vbRefs)
\  $(vbVbxs) $(vbDecls) $(vbBins) $(res)

#If we are going to build, first make sure the vbCmpat file is up to date.
 %If ("$(vbCmpat)" != "")
        %If (%Exists($(vbCmpat)))
#Make it writeable anf delete it first
   (noshell)(silent) attrib -r $(vbCmpat)
   del $(vbCmpat)
        %EndIf
#Then check it out of the archive
```

```
 (noshell)(silent) get -q $(getFlags) $(repository)\ref\$(Target)_v($(vbCmpat))
  %EndIf
#So long as we were successful getting the compatibility file,
   %If ("$(vbCmpat)" == "") || (%Exists($(vbCmpat)))
#compile the component.  The noshell modifier prevents VB from returning until all spawned processes are
#complete.  Note that VB 4 16bit does not deal well with long file names.
         (noshell) $(vb4Exe) /make $(vbProj)
#or for VB 5
    (noshell) $(vb5Exe) /make "$(vbProj)" /out "$(_TargRoot)Build.log" /outdir "$(_Cwd)"
#Note that VB 5 allows us to both control where the target will be built and allows us to specify a build log.
#CopyFiles is implemented as a dummy target.
    %Do CopyFiles
   %Else
#The vbCmpat is not an empty string but the file is not there.
   #May want more than just an error message.  perhaps a log or a retcode?
   @Echo Error, could not find compatibility file $(vbCmpat).
 %EndIf
#For VB 4 we can not control where the component is built we must copy it from the \se directory to the
#local directory.
CopyFiles:
 %If %exists(c:\se\$(Target))
   xcopy32 c:\se\$(Target)
    %Endif
#For VB 5, we'll build here in the local directory then copy and register the component into the \se
# directory  and can copy it to other directories (e.g. R:\Nyxdev\Bin\Se) as well
CopyFiles:
  %If %exists("$(Target)")
         xcopy32 "$(Target)" c:\se
  regsvr32 /s /c "c:\se\$(Target)"
```

%Endif

The only other oddity comes if one needed to specify that a .cls be pulled from the component's folder instead of using the common vbmoduls directory. This can be corrected with a .PATH.cls directive overriding the default behavior in the .PROLOG section after the .INCLUDE directory.inc instruction

---

Appendix

Various flags that can be used at the command line:

A flag is a single- or multi-character command entered from the ConfigurationBuilder command line that controls the build or dependency generation process. Flags provide a temporary and flexible method of controlling the program's actions. Flags are also referred to as switches or options.

Specifying Flags

Syntax:

To specify a flag, precede it with a single hyphen (-) or forward slash (/). You
can use upper-case or lower-case letters. Separate each flag from the next with a space, as follows:
build -noexecute -debug -summary
build /noexecute /debug /summary

Flag arguments:

Several of the Configuration Builder and ScanDeps flags accept an argument. Specify each flag argument after the flag that requires it. Configuration Builder requires a space between the flag and its argument. The following examples show Configuration Builder flags with arguments:

build -batch program.bat -script app.bld

Here is a list of the Configuration Builder flags and their functions.

“-All flag,” -A Builds all targets without comparing timestamps.
“-B flag,” Specifies an alternate builtins file.
“-Batch flag,” -C Creates a batch file containing the operation lines from a build script.
“-Close flag,” Causes Configuration Builder to close the Windows output window after a build is completed.
“-Compile flag,” Generates an executable from a build script.
“-Debug flag,”
-D Displays a record of Configuration Builder’s actions.
“-DebugOn, -DebugOff flags,”
Controls debug output issued by Configuration Builder.
“-DirectAll flag,”
-XE+O Directs all output to the specified device

or file.
“-DirectErrors flag,”

-X, -XE Directs error output to the specified
device or file.
“-DirectOutput flag,”

-XO Directs standard output to the specified
device or file.
“-Diag flag,” -# Selects a diagnostic function.
“-Emulate flag,” -M
-Nmake
Selects default, Microsoft MAKE, or
Microsoft NMAKE emulation.
“-Help flag,”
-? Displays a help message.
“-Ignore flag,” -I Ignores non-zero exit codes.

“-Include flag,”
@ Reads a command file for additional
command-line options.
“-Init flag,” Selects an initialization file other than
TOOLS.INI or rejects an initialization file.
“-KeepWorking flag,” -K Ignores serious errors and continues the build process.
“-NoEnvInherit flag,” Prevents the passing of environment
variable values to child processes.
“-NoEnvMacros flag,” Prevents the definition of environment
variables as macros.

“-NoExecute flag,” -N Displays operation lines and suppresses their execution.
“-NoLogo flag,”
Suppresses the sign-on and copyright
messages.
“-NoRedefine flag,” -E Ignores the redefinition of macros
derived from environment variables.
“-NoSilent flag,” Overrides the .Silent directive.
“-Quiet flag,”
-Q Suppresses the display of the sign-on
message, commands, and error
messages.
“-Rebuild flag,” Rebuilds a previous version.
“-Redirect flag,” -X Redirects error messages to a file or to
the screen.
“-Reject flag,”
-R Rejects generalized rules in the
initialization file, the builtins file, and
internal rule set.
“-RejectInternal flag,”
Prevents Configuration Builder from
reading the internal rule set.

“-RejectToolsIni flag,”

Prevents Configuration Builder from
reading information from TOOLS.INI or
other initialization file.
“-Script flag,” -F Specifies a build script.

"-ShowRules flag,"

Shows the internal rule set used by
Configuration Builder.
"-Silent flag," -S Suppresses the display of commands.
"-Summary flag," -P Displays a build script summary.
"-Touch flag," -T Assigns the current timestamp to all
targets that need to be built, but does
not build them.
"-View flag," -V Displays conditional lines during
execution.

Macros

A macro is a symbolic name which Configuration Builder replaces with a defined character string. You can define macros in build scripts, initialization files, and on the command line. The following terms apply to the use of macros:

- A **macro name** is a symbolic name.
- A **macro definition line** is the expression in the build script, initialization file, or command line which assigns a macro name to a macro value.
- A **macro value** is the character string you assign to the macro name.
- A **macro reference** is the macro name used in the build script to represent the macro value.
- **Macro expansion** occurs when Configuration Builder replaces the macro name with its macro value.

A macro definition line is an expression in the build script, initialization file, or on the command line which assigns a macro name to a macro value. By defining a symbolic name for a long string, you can save typing time, prevent errors, and promote flexible build script design. You can use command-line macro definitions to temporarily define conditions that vary from one build process to another, such as compiler flags, debugging options, or variable command-line parameters.

Syntax:

macro_name=["]character_string["]

The following examples define a macro called Objs as a group of object module filenames:
Objs = func1.obj func2.obj func3.obj

build objs = "app.obj subs.obj" -script
prog.bld

A transformation macro begins with a dollar sign ($) and is enclosed in square brackets [ ]. Each transformation macro is identified by the macro name followed by a comma (,). The remainder of the macro lists the character string and other parameters required for the transformation. Configuration Builder replaces the transformation macro with a transformed string.

PROLOG

Use the .Prolog reserved target to specify operations that Configuration Builder performs immediately after reading the build script, but before comparing timestamps. Configuration Builder performs these operations whether or not it finds a target that needs to be built. You can use .Prolog to set environment variables, log on a network, or perform other pre-build tasks.

Syntax

.Prolog:
list_of_operations

Example
The following example uses .Prolog to perform various pre-build tasks:

.Prolog:
pkunzip -o \arcs\c *.c
login stephf

EPILOG

Use the .Epilog reserved target to specify operations that Configuration Builder performs at the end of the build session, whether or not it finds a target that needs to be built. You can use .Epilog to delete unwanted files, log off a network, or perform other tasks needed to restore your environment.
Syntax:

.Epilog:
list_of_operations

Example: The following example logs off a network after a build session:

.Epilog:
logout stephf

Special Considerations
When terminating as a result of the %Exit built-in operation, Configuration Builder executes operations specified with the .Epilog reserved target. It does not execute these operations when terminating as a result of the %Abort built-in operation.

Rules

Rules provide instructions. Rules are lines of text that contain information about file relationships and the instructions that Configuration Builder follows to perform its tasks.
A rule consists of a dependency line and one or more operation lines. A dependency line defines a target and may also document the relationship of the target to the files used to create it. Operation lines contain the commands that build the target.

Explicit Rules

Explicit rules contain project-specific instructions for building a particular file or files. You define explicit rules directly in the build script. The following examples provide specific instructions for building the file PROGRAM.EXE

program.exe : prog1.obj prog2.obj prog3.obj
tlink lib\c0m $(_Sources), $(_TargRoot), $(_TargRoot),

\
lib\emu lib\mathm lib\cm

Generalized Rules

Generalized (or implicit) rules specify instructions for building files with a certain extension. Configuration Builder can use generalized rules specified in the build script, builtins file, initialization file, or internal rule set. The rule below provides general instructions for building files with the extension .C from files with the extension .OBJ

.c.obj :
cl /Fo$(_Target) /c $(_Source)

Special Considerations

The precedence of rules is determined by the order that Configuration Builder reads them. Because Configuration Builder reads an initialization file before reading the build script, generalized rules and macro definitions defined in the build script have precedence over rules and macros defined in the build script.

Path

Here is the chronological hierarchy by which CB looks for files it needs to build your app. It starts in the repository, and works its way down searching for the files it needs.
Repository...
Main Directory...
Micellaneous directory list...
Sub directory list...
Makefile list

**** .Path** Directive

Use the .Path directive to specify search paths for files with a certain extension. Place the .Path directive in the build script or TOOLS.INI file.
Syntax
.Path. ext = path[; path...]
Specify the extension to which the search applies with the .ext parameter. Use the path parameter to define the directory or path name to search. Separate multiple paths with semicolons (;). Configuration Builder searches the paths in the order they are listed. Neither the .ext nor the path parameter is case-sensitive.
Configuration Builder automatically defines macros containing directory locations specified with the .Path directive. You can reference these macros anywhere in the build script. The case of the macro reference must match the case of the macro name exactly.

Read-time and run-time use
Read-time only. The .Path directive is in effect for the entire build process.
Examples • The following examples direct Configuration Builder to look for files with a .C extension in the \BIN\SOURCES directory:
.Path.c = \bin\sources

• The following examples use the .Path directive to define a drive and directory for .C files and reference the definition in a dependency line:

```
.Path.c=d:\apps
test.obj : $(.Path.c)\test.c
```

Because macro expansion is case-sensitive, Configuration Builder would not expand the $(.Path.c) macro reference if the corresponding directive line read .PATH.C=d:\apps (for UNIX, .PATH.C=/apps).

Special Considerations

• Configuration Builder's internal rule set contains .Path directives that specify the current working directory as the default location of files with certain extensions. To override these defaults, specify a .Path directive in TOOLS.INI or the build script, or use the -Reject flag or .RejectInitFile directive.

• If you leave a trailing backslash (\) on the path name following the .Path directive, Configuration Builder processes it as a line continuation character. To prevent Configuration Builder from treating the backslash as a line continuation character, either follow it with a space or a semicolon (;), or precede it with a caret (^).

• If Configuration Builder cannot locate the file in any of the directories specified by .Path, it looks for instructions for creating it. If it cannot find any instructions, it uses the .Default reserved target. If the .Default reserved target does not exist, Configuration Builder terminates the build process.

• If a file referenced by the _Target context macro does not exist, Configuration Builder expands _Target to the first path name specified by an applicable .Path directive. If the file does exist, Configuration Builder expands _Target to the directory where it located the file, which is not necessarily the first directory in the list. If multiple files with the same filename exist in different .Path directories, Configuration Builder may overwrite the wrong one when it expands _Target.