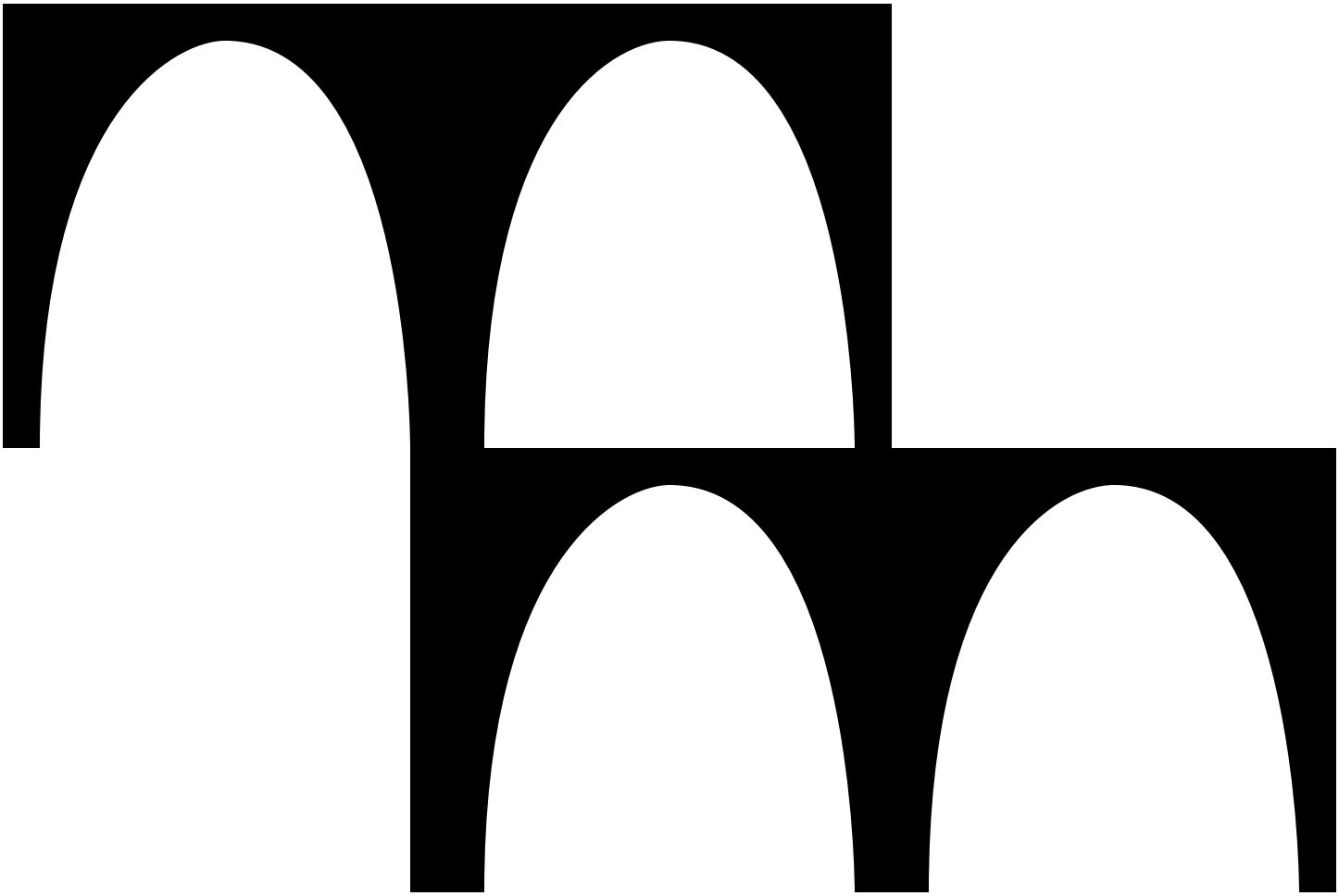

TinyBuilder Tutorial

Or

How to get Building as Soon as Possible

This page intentionally left blank

Contents

This page intentionally left blank

Getting Started

TinyBuilder is run with a script. The following script will use gcc to build hello.exe:

```
step compile source
    commands
        gcc -o hello.exe hello.c

job build hello
    steps
        compile source
    input
        hello.c
    output
        hello.exe
```

The above script specifies a single job, `build hello`, with a single step, `compile source`. Jobs and steps have a block structure. The `commands`, `steps`, `input` and `output` are block headers which specify a type to the data within the block. There are no quotes or shell escapes in TinyBuilder scripts; indention and new-lines take their place.

The `input` block of the job specifies the files used by the job. When the script is run, TinyBuilder creates a file cache containing the files listed in the input block and any scripts or executables run from within the job will have their current directory set within the file cache. By running within the cache, only the explicitly specified files are used, providing consistent results and preventing difficult to diagnose bugs. Since the job works from copies, all files can be safely edited while the job is in progress. It is even possible to submit the same job again before the first job is done.

The `output` block specifies the files that will be returned by a job. The commands run as part of the job create or update the files within the file cache while the job is in progress, and the files are copied back to the build area when the job successfully completes. Only files included in the output block are copied from the file cache.

The step `compile source` runs the command line to build hello.exe and the job runs the steps specified in its `steps` block. The command line in the `compile source` step is in the simple syntax. It looks familiar, but does not work when the executable or any argument contain spaces. The full syntax is needed to describe arguments con-

taining spaces. Following is the equivalent command line using the full syntax:

```
step compile source
    commands
        gcc
            -o
            hello.exe
            hello.c
```

In the full syntax, the executable name starts a block and each command line argument is indented. All the characters on an argument line from the first non-whitespace character to the end of the line will be passed as a single command line parameter. The executable name may also include spaces when the full syntax is used.
To run the script, execute the command:

```
tbuild script.tb
```

In Windows, double-clicking a file with the .tb extension will also work. Once started, `tbuild` will print the following:

```
Parsing scripts...
Generating command lines...
Initializing file cache
File cache built, files may now be safely edited.
Connecting to builders
Connected to 1 builder
Executing command lines
```

An important message is that stating that the file cache is built. Once `tbuild` prints this message, all of the input files have been copied into the cache and may be safely edited. After building the cache, `tbuild` connects to a builder; the file cache is transferred and the commands are run by the builder within its own copy of the cache. The output files listed in the `output` block are transferred back to `tbuild` to replace the corresponding files within its file cache. When `tbuild` completes, the output files are copied from the file cache to the build area.
Since the job is running within a file cache, there is no way to know what the absolute directory will be before the job is started; only relative paths can be used in scripts. It is possible to access files on a different Windows drive using a relative path within the TinyBuilder script. The file cache will always contain the drive letter as part of the path. So if the build script is placed in C:\build and the source is in D:\source, an executable running in C:\build will be able to access the source directory using ..\..\D\source.

Job Log

Executing `tbuild` generates `job_log.xml`, which contains a report of the build. The file is in the XML format to be both human readable and machine pars-able. The example script creates the following `job_log.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<TinyBuilderLog
  xmlns="http://www.tinymanagement.com/TinyBuilder/JobLog/1.0/"
  version="1.0">
  <JobStart time="2014-12-25T11:54:11.624760Z" job="build hello"
    cache="\\?\C:\build\tbuild.tmp\tmpktyvqvca">
    <InputFiles>
      <file>C:\build\hello.c</file>
    </InputFiles>
    <ScriptFiles>
      <file>C:\build\script1.tb</file>
    </ScriptFiles>
  </JobStart>
  <Job name="build hello" status="success"
    StartTime="2014-12-25T11:54:11.811960Z"
    EndTime="2014-12-25T11:54:22.170378Z">
    <InputFiles>
      <files>C:\build\hello.c</files>
    </InputFiles>
    <step command="gcc" type="command"
      directory="C:\build\" time="2014-12-25T11:54:11.952360Z"
      return="0">
      <parameters>
        <parameter>-o</parameter>
        <parameter>hello.exe</parameter>
        <parameter>hello.c</parameter>
      </parameters>
    </step>
    <OutputFiles>
      <files>C:\build\hello.exe</files>
    </OutputFiles>
  </Job>
</TinyBuilderLog>
```

There are two main sections to the file. The data within the `JobStart` element documents the input to all jobs of the script. After the `JobStart` element, there is a se-

quence of `Job` elements that contain the details of each job.
All output to the stdout and stderr is listed within each step element. Running `tbuild` within a script would result in the following:

```
<err>
  <line time="2014-12-25T12:13:35.628643Z">
    Parsing scripts...
  </line>
  <line time="2014-12-25T12:13:35.632125Z">
    Generating command lines...
  </line>
  <line time="2014-12-25T12:13:35.632432Z">
    Initializing file cache
  </line>
  <line time="2014-12-25T12:13:35.632568Z">
    File cache built, files may now be safely edited.
  </line>
  <line time="2014-12-25T12:13:35.632727Z">
    Connecting to builders
  </line>
  <line time="2014-12-25T12:13:35.632805Z">
    Connected to 1 builder
  </line>
  <line time="2014-12-25T12:13:35.632869Z">
    Executing command lines
  </line>
</err>
```

Lines sent to the stdout are contained within an `out` element.
No message is printed to the console to indicate if a job was successful or not. To determine if a job was successful, refer to the `status` attribute of the `Job` element in the job log.
The `-log` command line option can be used to specify a different file name for the job log.

Comments

Comments are lines starting with `#`. For example:

```
# Step to compile the source
step compile source
    commands
        gcc -o hello.exe hello.c

job build hello
    steps
        compile source
    input
        hello.c
    output
        hello.exe
```

Any line with `#` as the first non-whitespace character will be treated as a comment; otherwise, `#` will be treated as data. For example:

```
step compile # source
    commands
        gcc -o hello.exe hello.c
```

The above will create a step named `compile # source`, not a step named `compile` followed by a comment.

Data

The previous script is no better than a batch file. A more useful script would be:

```
step compile source
    parameters
        value source
    commands
        gcc -c -o <source>.obj <source>.c

step link source
    parameters
        list source
        value binary
    commands
        gcc -o <binary>.exe <source>.obj

job build multi-file hello
    data
        source file=hello_no_str
        source file=strings
        binary=hello
    steps
        compile source
            source file
        link source
            source file
            binary
    input
        <source file>.c
    output
        <binary>.exe
```

The `build multi-file hello` job has a `data` block that defines a set of name/value pairs for use by the job. The name consists of every character before `=`. The value assigned to the name is everything past the equal sign to the end of the line. So any character except the new line may be part of a value. Many programmers code assignment statements as `name = value`, but this will not work as intended in a TinyBuilder script. For the above assignment, the name is `name ` and the value is ` value`.

A name may have any number of assignments to it, even zero. All data in TinyBuilder scripts are lists; a second assignment never overwrites a value; it adds a value. So in the above script, the `source file` has two values, `hello_no_str` and `strings`. The

`binary` value has a single value, `hello`. Any name that is not specified in the data block is treated as a list with no members, so it is not an error to use data that is not specified in the `data` block.

Step Parameters

When calling a step, the step parameters are passed to a step positionally, with each data name passing its values to the parameter in the same position within the step's parameter list. The command lines are constructed using the values assigned to the names of the parameters.

The name of the parameter may contain any character and is terminated by the new line. The parameter type, which can be either `value` or `list`, specifies how each value is used to create command lines. If the type of the parameter is `value`, each value in the value list results in a new command line. The values passed to the parameters replace the parameter name delimited within the `<` and `>` characters. In the above script, since the `source` parameter has two values, the `compile source` step will produce a command line for each value:

```
gcc -c -o hello_no_str.obj hello_no_str.c
gcc -c -o strings.obj strings.c
```

If the type of the parameter is `list`, all the values are repeated within a single command line. The `source` is passed to the `link source` step as a list parameter, so the step will generate a single command line containing both values:

```
gcc -o hello.exe hello_no_str.obj strings.obj
```

If `hello2` was added to the `binary` value, then the step would have generated:

```
gcc -o hello.exe hello_no_str.obj strings.obj
gcc -o hello2.exe hello_no_str.obj strings.obj
```

Data names surrounded by `<` and `>` within an input or output block work exactly like a list parameter.

Note that all the adjacent non-whitespace characters and the preceding whitespace to the parameter name are treated as part of each list value. In the `link source` command line, the space before the `<` becomes a prefix and all the non-whitespace characters after `>` (`.obj`) are used as a suffix. Non-whitespace before the parameter name will be treated as part of the value as well. So if the command line in the

`link source` step had been:

```
gcc -o <binary>.exe ..\<source>.obj
```

Then the resulting command line would have been:

```
gcc -o hello.exe ..\hello_no_str.obj ..\strings.obj
```

because `..\` is used as a prefix.

Empty Lists

A TinyBuilder script is permitted to pass an empty list to a step by passing a name with no value assigned. This is not an error as it is in most programming languages:

```
step compile source
    parameters
        value source
        list optimization options
        list compile options
    commands
        gcc <optimization options> <compile options> -c -o <source>.obj <source>.c

step link source
    parameters
        list source
        list link options
        value binary
    commands
        gcc <link options> -o <binary>.exe <source>.obj

job build multi-file hello
    data
        source file=hello_no_str
        source file=strings
        binary=hello
#        optimization option=-O3
#        optimization option=-fomit-frame-pointer
        compile option=-std=c11
        compile option=-fsigned-char
        link option=-shared-libgcc
        link option=-s
    steps
        compile source
            source file
            optimization option
            compile option
        link source
            source file
            link option
            binary
    input
        <source file>.c
    output
        <binary>.exe
```

Since the optimization option is not set when it is passed to the compile source step, the command lines are expanded to:

```
gcc -std=c11 -fsigned-char -c -o hello_no_str.o hello_no_str.c
gcc -std=c11 -fsigned-char -c -o strings.o strings.c
```

The empty list has no impact on the expansion of the command line. The behavior is the same whether the parameter is a `list` or `value` type.

Combining Values

In the previous scripts there are whitespace between the parameter values in the command line. When there is whitespace between the values, every value of every parameter is used exactly once in every command line. If there is no whitespace between `>` and `<`, TinyBuilder will assume that all combinations of the values should be used for each command line as in the following script:

```
step process file
    parameters
        list base
        list extension
    commands
        python process_files.py src\<base>.<extension> -dest
dest\<base>.<extension>

job copy files
    data
        base=file1
        base=file2
        base=file3
        extension=txt
        extension=h
        extension=c
    steps
        process file
            base
            extension
    input
        src\<base>.<extension>
```

Because there is no whitespace between `base` and `extension`, the `process file` step will generate the following command line:

```
python process_files.py src\file1.txt src\file2.txt \
src\file3.txt src\file1.h src\file2.h src\file3.h src\file1.c \
src\file2.c src\file3.c -dest dest\file1.txt dest\file2.txt \
dest\file3.txt dest\file1.h dest\file2.h dest\file3.h \
dest\file1.c dest\file2.c dest\file3.c
```

This feature is intended for use with file name/extension combinations as above or any other case when a set of command line options need to be all possible combination of values.

Job Inheritance

Frequently, different binaries, such as debug binaries, are built in largely the same way as the release version with only a few differences. The easiest and most maintainable way perform similar jobs is using job inheritance. The following script includes a debug version of the multi-hello build:

```
step compile source
    parameters
        value source
        list optimization options
        list compile options
    commands
        gcc <optimization options> <compile options> -c -o <source>.obj <source>.c

step link source
    parameters
        list source
        list link options
        value binary
    commands
        gcc <link options> -o <binary>.exe <source>.obj

job build multi-file hello
    data
        source file=hello_no_str
        source file=strings
        binary=hello
        optimization option=-O3
        optimization option=-fomit-frame-pointer
        compile option=-std=c11
        compile option=-fsigned-char
        link option=-shared-libgcc
        link option=-s
    steps
        compile source
            source file
            optimization option
            compile option
        link source
            source file
            link option
            binary
    input
        <source file>.c
    output
        <binary>.exe

job debug build multi-file hello
    base
        build multi-file hello
    data overrides
        optimization option=-O0
    data
        compile option=-g
        link option=-g
```

The job `debug build multi-file hello` has a `base` block, which specifies that the `debug build multi-file hello` inherits from the job `build multi-file hello`.

The subjob uses all the data and steps of the superjob, but allows adjustments. Any data named in the subjob's `data overrides` block replaces values with the same name in the superjob; any values in the subjob's `data` block are added to the values in the superjob with the same name. The superjob may still be run by itself; when the superjob is run, the subjob has no effect.
When `debug build multi-file hello` is executed, the `compile source` step will generate the following commands:

```
gcc -O0 -g -std=c11 -fsigned-char -c -o hello_no_str.o hello_no_str.c
gcc -O0 -g -std=c11 -fsigned-char -c -o strings.o strings.c
```

Since `optimization option` is in the `data overrides` block, the `-O3` and `-fomit-frame-pointer` command options are discarded and replaced with `-O0` within the subjob. Since the `compile option` is in the subjob's `data` block, the `-g` option is added to the `-std=c11` and `-fsigned-char` options. The `source file` data is not changed by the subjob so there are still two command lines.
The `link source` step will generate the following command line:

```
gcc -g -shared-libgcc -s -o hello.exe hello_no_str.o strings.o
```

Since the subjob has a `link option` in the `data` block, the `-g` command line option is added to the `-shared-libgcc` and `-s`. The other data passed to that step is the same as when the `build multi-file hello` job runs the `link source` step.
All the blocks within the subjob behave in a similar way. The `data`, `steps`, `input` and `output` blocks all add additional information. The `data overrides`, `step overrides`, `input overrides` and `output overrides` remove the information assigned in the superjob and replace it with the data in the subjob.
All of the previous scripts had only one job, so `tbuild` was able to decide which job to run automatically. However, the above script has two, so the `-job` command line option is needed. The following command lines would run each job in the script:

```
tbuild -job 'build multi-file hello' script.tb
tbuild -job 'debug build multi-file hello' script.tb
```

Error Handling

By default, when a command line returns a non-zero value, the job will stop executing and fail. No output files will be copied from the cache when the job fails.
It is possible to specify that a step should ignore the return values of its commands; so the step and job will continue no matter what value each command returns. To

specify the step to ignore return values, an `error handling` block is needed:

```
step compile source
    error handling
        no fail on return value
    commands
        gcc -o hello.exe hello.c
```

The `no fail on return value` is the only accepted error handling directive. The effect of the `error handling` block only applies to its step, so other steps can still fail.

Environment Variables

Many utilities require environment variables to be properly set. For example, the Visual Studio command line utilities require vcvarsall.bat, but executing the batch in a TinyBuilder script will not take effect. Fortunately, it is possible for a step to set environment variables to be passed to subsequent commands within the job.

When python is installed on Windows, it is not added to the path by default. So to run python from TinyBuilder, a step like this is needed:

```
step add python 3.4 to path
    commands
        |PATH=prefix=c:\Python34;
```

The effects of the above step will last for the rest of the job, but will not affect the next job. So like data, setting an environment variable lasts until the job ends, but unlike data, the environment variable is set within the sequence of command lines. When the first character of a command line is `|`, TinyBuilder interprets the command as one that will adjust the environment. The `=prefix=` operator means that the value is prefixed to the current environment value. So in the above script, `c:\Python34` is set to be the first directory within the PATH and the rest of the variable remains unchanged. Note that the path separator is required for the environment variable to be interpreted correctly by the OS.

The following script adds `c:\Python34` to the end of the path:

```
step add python 3.4 to path
    commands
        |PATH=suffix=;c:\Python34
```

To replace the PATH variable so only `c:\Python34` is on the path:

```
step add python 3.4 to path
    commands
        |PATH=c:\Python34
```

And to delete the PATH environment variable:

```
step add python 3.4 to path
    commands
        |PATH=
```

Projects

Any substantial software product involves multiple binaries. While it makes sense for each binary to have its own job to permit developers to build each binary independently, it is useful for a TinyBuilder script to specify how all of the binaries should be built as a group. The following script adds a project to the script that runs the `multi-file hello world` job:

```
step compile source
    parameters
        value source
        list optimization options
        list compile options
    commands
        gcc <optimization options> <compile options> -c -o <source>.obj <source>.c

step link source
    parameters
        list source
        list link options
        value binary
    commands
        gcc <link options> -o <binary>.exe <source>.obj

job build multi-file hello
    data
        source file=hello_no_str
        source file=strings
        binary=hello
        optimization option=-O3
        optimization option=-fomit-frame-pointer
        compile option=-std=c11
        compile option=-fsigned-char
        link option=-shared-libgcc
        link option=-s
    steps
        compile source
            source file
            optimization option
            compile option
        link source
            source file
            link option
            binary
    input
        <source file>.c
    output
        <binary>.exe

job debug build multi-file hello
    base
        build multi-file hello
    data overrides
        optimization option=-O0
    data
        compile option=-g
        link option=-g

project hello world
    jobs
        build multi-file hello
```

Now that the script contains a project, the `-job` command line parameter is no longer necessary to run the `build multi-file hello` job. If there are multiple jobs in a script but only one project, tbuild will run the project by default. If a script contains

multiple projects, the `-job` command line parameter will be needed to specify a project. Since the `debug build multi-file hello` job is not in the project, the `-job` command line parameter would be required to run that job using the above script.

Importing Scripts

It is possible for a TinyBuilder script to import other scripts, which permits one master project file to import a script for each binary. For example:

```
# import the script to build hello.exe
import ..\main\build.tb

project hello world
  jobs
    build multi-file hello
```

Every script used is listed within the JobStart element of the job log.

When a script specifies input files, output files or executables, the location of those files are relative to the directory containing the script and does not change when the script is imported by a script in another directory. The current directory of a subjob is the directory of the script, even if the superjob is contained in a script in a different directory.

Machines

By default, `tbuild` will connect to a builder running on the localhost. However, it is possible to have `tbuild` perform the build on a build machine. To specify a machine, add a `machines` block to the job:

```
step compile source
    commands
        gcc -o hello.exe hello.c

job build hello
    steps
        compile source
    input
        hello.c
    output
        hello.exe
    machines
        build
```

Multiple machines may be specified for a job, in which case, the job may be run on any machine in the list. When a job inherits from a job with a `machines` block, it in-

herits the `machines` block from the superjob. However, inheritance of the `machines` block works differently than the other blocks. If the subjob has a `machines` block, the subjob may only run on machines listed in both the subjob and superjob `machines` blocks. Projects can also have a `machines` block which works in a similar way. As with subjobs, the list of machines specified for a project will limit the machines that the jobs may use. Think of the `machines` block as the list of every machine that can possibly run the job. Presumably, the subjob requires more functionality than the superjob, and so can run on fewer machines.

Path Separators

Since the `\` path separator is valid in Windows but not Linux, and the `/` path separator is valid on both platforms, the `/` path separator should be used in all cases.
The files specified in the `input` and `output` blocks are resolved on the machine running `tbuild`, while the paths specified in the `commands` block within a step are resolved on the builder. The path separators in use must be valid on the machine where the file paths are resolved. So if a Linux builder is added to a working Windows based build system, only the paths in the steps must be changed as long as `tbuild` is started from a Windows system. If a Linux system runs a job containing input files with Windows paths, the input files will not be found and the script will fail to run.

Job Scheduling

A project may run a job at any time when all of its dependencies are fulfilled. If a job has a file in its `input` block and the same file is in another job's `output` block, the first job will wait for the second to succeed. If two jobs may be dispatched to two different machines and have no dependencies between them, then it is possible that the two jobs will be run at the same time on different machines.
It is possible to force the sequencing of jobs using flags. A flag is an arbitrary string that becomes set upon job completion. If the flag is within a `success flags` block, the flag is set when the job succeeds. If the flag is within a `failure flags` block, the flag becomes set when the job fails. A job with a `required flags` block will not start until all the flags within the block are set and all the input files have been built.
The flags are inherited from a superjob. The `success flags` and `failure flags` specified in a superjob become set when the subjob completes as well as the flags specified in the subjob. To prevent the superjob flags from getting set when the subjob completes, use `success flag overrides` and `failure flag overrides` blocks. A subjob requires the flags of the superjob and the flags specified in its `required flags` block. To drop the requirements of the superjob, the `required flag overrides` block should be used.

Best Practices

- If two different scripts have a step or job with the same name, the parser will report duplicate names if those two scripts are imported into the same project. To prevent problems, there are two approaches: 1. All steps are defined in a script that contains only steps, or 2. All steps and jobs are given names that are unique among all TinyBuilder scripts that may be imported. Importing the same script more than once will not result in duplicates.
- By keeping jobs small and making the machines block as large as possible, the job scheduler is given the greatest flexibility to maximize build speed.
- Create separate jobs for building and testing. By doing so, it becomes easy to build without performing a test, or test without performing a build by setting up projects with the appropriate jobs.
- To permit building using a standalone job, create a job inheriting from the standalone job for use within the project. The subjob can then require flags that the standalone job does not require. If a job requires flags, it will not run if the `tbuild -job` parameter specifies that job since the flags will never be set.
- By default, declare a step parameter to be a list rather than a value. Remember that value parameters cause additional command lines to be generated, so a parameter should only be a value if one command invocation per value is the required behavior.
- Minimize the steps using the `no fail on return value` error handling directive. The intent of the directive is to allow TinyBuilder to properly use tools that are sloppy with their return values. The overuse of the directive could make unexpected failures harder to diagnose.
- Always use `/` as the path separator to prevent problems when running the script on multiple platforms.

Appendix A

This script will set up the environment for Visual Studio 2013 Express tools

```
# The vs2013 job sets up the environment for Microsoft compilation tools to
# work.

# Use the step set vs2013 environment to set the environment of the job to run
# the VS2013 tools.
# The vs2013 job is used to provide reasonable defaults for the parameters
# passed to set vs2013 environment; the value names are the same as the
# parameter names.
step set vs2013 environment
    parameters
        value vs2013 path
        value sdk path
        value microsoft sdk
        value windows sdk
        value framework path
        value framework64 path
        # Must be x86 or x64 (case sensitive)
        value cpu
    commands
        |DevEnvDir=<vs2013 path>Common7\IDE
        |ExtensionSdkDir=<microsoft sdk>Windows\v8.1\ExtensionSDKs
        |Framework40Version=v4.0
        |FrameworkDir=<framework64 path>
        |FrameworkDir32=<framework path>
        |FrameworkVersion=v4.0.30319
        |FrameworkVersion32=v4.0.30319
        |INCLUDE=<sdk path>INCLUDE;<windows sdk>include\shared;<windows sdk>include\um\;<windows
sdk>include\winrt
        |LIB=<sdk path>LIB;<windows sdk>lib\winv6.3\um\<cpu>
        |LIBPATH=<framework path>;<sdk path>LIB;<windows sdk>References\CommonConfiguration\Neu-
tral;<microsoft sdk>Windows\v8.1\ExtensionSDKs\Microsoft.VCLibs\12.0\References\CommonConfiguration\
neutral
        |PATH=prefix=<vs2013 path>Common7\IDE\CommonExtensions\Microsoft\TestWindow;<microsoft sd-
k>TypeScript\1.0;c:\Program Files\MSBuild\12.0\bin;<vs2013 path>Common7\IDE;<sdk path>BIN;<vs2013
path>Tools;<framework path>v4.0.30319;<sdk path>VCPackages;<windows sdk>bin\<cpu>;<microsoft sd-
k>Windows\v8.1A\bin\NETFX 4.5.1 Tools;
        |VCINSTALLDIR=<sdk path>
        |VisualStudioVersion=12.0
        |VS120COMNTOOLS=<vs2013 path>Common7\Tools\
        |VSINSTALLDIR=<vs2013 path>
        |WindowsSdkDir=<windows sdk>
        |WindowsSDK ExecutablePath x86=<microsoft sdk>Windows\v8.1A\bin\NETFX 4.5.1 Tools
        |WindowsSDK ExecutablePath x64=<microsoft sdk>Windows\v8.1A\bin\NETFX 4.5.1 Tools

job vs2013
    data
        # If VS2013 is installed in another location, override this job and
        # use data overrides to correct the vs2013 path and sdk path variables.
        vs2013 path=c:\Program Files\Microsoft Visual Studio 12.0\
        sdk path=c:\Program Files\Microsoft Visual Studio 12.0\VC\
        # If using an SDK installed in a non-default location, override this
        # job and use data overrides to correct the microsoft sdk variable
        microsoft sdk=c:\Program Files\Microsoft SDKs\
        windows sdk=c:\Program Files\Windows Kits\8.1\
        framework path=c:\Windows\Microsoft.NET\Framework\
        framework64 path=c:\Windows\Microsoft.NET\Framework\
```

The above script was written by running vcvarsall.bat in a command prompt and com-

paring the output of the `set` command from another command prompt.
The following example script makes use of the above script. Since the imported script does not specify any files, the script may be correctly imported by a script in a different directory.

```
import ../build/vs2013.tb

step compile source
    parameters
        value source
    commands
        cl /c /Fo<source>.obj /c <source>.c

step link source
    parameters
        list source
        value binary
    commands
        link /OUT:<binary>.exe <source>.obj

job build multi-file hello
    data
        source file=hello_no_str
        source file=strings
        binary=hello
    steps
        set vs2013 environment
            vs2013 path
            sdk path
            microsoft sdk
            windows sdk
            framework path
            framework64 path
            cpu
        compile source
            source file
        link source
            source file
            binary
    input
        <source file>.c
    output
        <binary>.exe
```

Appendix B

The contents of the example source files are as follows:

hello.c:

```
#include <stdio.h>

int main(int argc,char **argv) {
    printf("hello world\n");
}
```

hello_no_str.c:

```
#include <stdio.h>

extern char *hello_str;

int main(int argc,char **argv) {
    printf("%s\n",hello_str);
}
```

strings.c:

```
char *hello_str="Hello world";
```