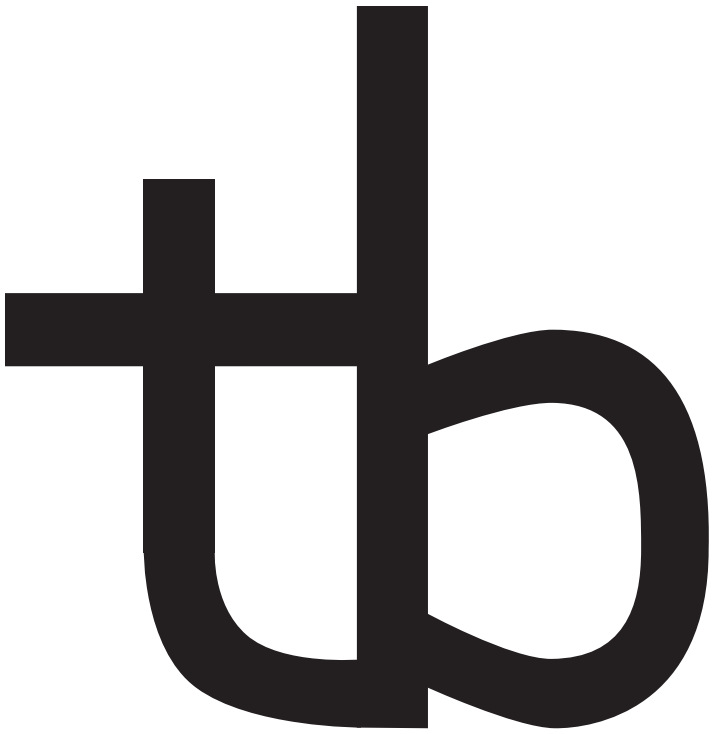

TinyBuilder

User Guide

Beta

Table of Contents

Introduction

The TinyBuilder documentation comes in three parts, a user guide, a reference guide, and for the non-Windows platforms, man pages.

The user guide is intended to be the most used document. It provides the information needed to build and maintain a working build system that makes optimal use of TinyBuilder, without the details that are only relevant for diagnosing problems. The user guide begins with the TinyBuilder script, which is a simple and elegant way to describe a build, followed by the method used to construct command lines. The user guide continues with a brief description of the build log format, an XML based format designed to be both human and machine readable. The client and agent chapters of the user guide describe how to use those components, and finally, the user guide ends with a brief story describing the evolution of a fictional build.

The reference guide is intended to be useful only on the occasions when a difficult problem needs to be solved. The reference guide begins with a detailed explanation of the parser, which is helpful for explaining text encoding issues and error messages, followed by a comprehensive description of the build log format. The next chapter goes into a detailed explanation of the operation of the TinyBuilder service and agent, for use by system administrators who wish to diagnose problems or run tests on exotic platforms. The reference guide ends with a chapter detailing the integration of ssh into TinyBuilder to provide authentication and encryption.

The man pages are intended to be a quick reference where they are available. While not particularly informative, they will provide the exact spellings of environment variables and command line options. They can also serve as a prompt for more complex concepts explained elsewhere. While not normally used, the man pages will provide the command line interface of the service executables on Linux and macOS; the Windows service and agent have no command line interface.

The Script

The foundation of the TinyBuilder script is the job; the job specifies a list of input files, a list of commands and parameters, and a list of output files. The input files are transferred from the client to the machine, the commands are run on the machine, and the output files are transferred from the machine to the client. A job is self contained; it runs within a work area, which is a context that is completely specified by the job; other jobs that happen to be running on the machine have no effect. The current directory used by the commands corresponds to the directory of the script containing the job; the input and output files have the same relative location on the client and in the work area.

The TinyBuilder script is structured as a set of blocks; a block's contents are given a greater indention than the line starting the block. The block ends when a line with a lesser indention is read. The block is a recursive concept; each sub block is terminated when the indention of the next line is equal or smaller.

Each block may be thought of as having a type. The contents of a block depend on the block type; the contents of a block may be strictly content, or they may be keywords identifying inner blocks with their own type. Usually, an inner block of the same type may be specified more than once within the same outer block; subsequent inner blocks add to the original block of the same type.

While input files, commands and output files can be specified explicitly within a job block, doing so does not make creating any other job easier. Instead, lists of files may be specified in a file list block, command line parameters may be specified in data blocks, and sequences of commands may be specified in step blocks. Each job may then include these blocks, so jobs can be updated by changing the blocks they include.

Command line parameters are specified by assigning values to names. Each assignment to a name adds to the name; an assignment never replaces a value. A name becomes a command line parameter by expanding the name. Names may be assigned a data value, which are not interpreted by TinyBuilder, or names may be assigned a path value, which are converted into paths located relative to the job's directory. Expansion options affect how a name is expanded, i.e. each expansion is in its own command line or within a single command line.

Jobs may be included into projects as a build or a test job. Build jobs have output and will be considered up to date after they are run; they do not need to run again unless their input is updated. Test jobs generally do not have output when they succeed; only their exit status is used

instead. Test jobs will run as many times as requested, even if no input to the job has changed.

Each job is assigned one or more machines; a machine is like a server, and probably is one. However, TinyBuilder uses the term machine because while a job may run on a massive build server servicing hundreds of clients, a machine may also be a small deployable device used exclusively for testing. The script may specify a list of machines for a job so the job load balances between the machines, or the job may run on all the machines.

A single build may make use of multiple TinyBuilder scripts, each imported into a single build script. When imported, the build scripts may be located alongside the code they use. File paths remain relative to the script specifying the paths, which can be a different directory than the script specifying the job.

After the jobs have all been run, the client constructs a build log, which contains information pertaining to its connections, jobs and commands. The build log is designed to be at least as useful as running the commands interactively, and is frequently better. Output from commands run by the job is never lost and frequently do not even require the command to block; the pipe between the process and the TinyBuilder server is a wide one. The build log is formatted in a human readable form of XML.

The Job

The fundamental unit of the TinyBuilder script is the job; no work can be done without one. The job reads a set of input files on the client, runs a set of commands on the machine and writes a set of output files on the client. All paths and commands are relative to the directory of the script containing the job block; all files must be specified using relative directories.

The job block is started with the text `job` with no indention. Immediately following the `job` keyword is the name of the job. The name may be any text that can be expressed by UTF-8 except for the new line. The name is matched byte by byte; no case folding, composition or decomposition is done when comparing names.

A minimal TinyBuilder script would look something like this:

```
job minimal job
    input
        main.c
        module1.c
        module2.c
        module1.h
        module2.h
```

The Script

```
values
    compiler options = -O3
    compiler options = -m64
    compiler options = -g
    compiler options = -I.
    link options = -m64
    link options = -g
paths
    source file = main.c
    source file = module1.c
    source file = module2.c
    executable = program
command break on error
    gcc
        <compiler options>
        -o
        <<base name, enumerate>source file>.o
        <<enumerate along>source file>
    gcc
        <link options>
        -o
        <executable>
        <<base name>source file>.o
output
    program
machine
    builder
```

As explained in more detail below, when `minimal job` is executed, `main.c`, `module1.c`, `module2.c`, `module1.h` and `module2.h` are compressed into the input archive and sent to the TinyBuilder server named `builder`. All the input files are must be in the same directory as the TinyBuilder script containing `minimal job` or the job will refuse to start. The current directory is created within a work area on the server and the input archive is extracted relative to that directory.

The following command lines are executed within the current directory within the work area on `builder`:

```
gcc -O3 -m64 -g -I. -o main.o main.c
gcc -O3 -m64 -g -I. -o module1.o module1.c
gcc -O3 -m64 -g -I. -o module2.o module2.c
```

```
gcc -m64 -g -o program main.o module1.o module2.o
```

Each command line starts after the previous command line has completed; no parallel execution is performed within a job. Since the command `gcc` is specified without a directory, it must be found on the server using `PATH`. Assuming all of the `gcc` invocations are successful, a compressed output archive consisting only of `program` is created and transferred to the client. The file `program` is extracted from the archive into the same directory as the script containing `minimal job` and the job is successfully completed.

The `input` block provides a list of input files used by the job. The client produces an input archive from the list of input files and transfers the archive to the machine. All of the files listed in the input block will be available for use on the machine before the first command is started, in the same relative location as the script defining the job.

Each job has a failed flag which is set if a command within a `command break on error` or `command complete with error` block returns a non-zero value. Once the failed flag is set, commands within a `command complete with error` and `command ignore error` block will be executed to the end of the block; a failed command in a `command break on error` block will cause the job to stop immediately. Like other blocks, each new command block adds commands to the job in sequence.

A command has zero or more command line parameters. The parameter list is a block within the command, with each parameter terminated by the new line. Any character may be specified in a parameter except for a new line, '<' or '>'; there is no escaping or quoting. Note that output redirection cannot be done within a command block. While the command block may run a script that redirects the output of a command, it is generally better to allow the output to appear in the build log; no output from a command is lost.

The '<' and '>' characters act as delimiters to specify an expansion. An expansion replaces the sequence '<', the name and '>' with all the values that have been assigned to that name. Each expansion, combined with any text provided, become a command line parameter passed to the command. If the name has no expansions, the text is passed as the parameter, or if there is no text and no expansion, no command line parameter is passed.

A list of output files are provided in the `output` block. The files are specified in a directory relative to the script defining the job; if an output file does not exist in the work area after the last command completes, the job fails. The machine creates an output archive from the files in the work area and transfers the archive to the client. The client extracts the files using paths relative

to the script defining the job.

There are several aspects of `minimal job` to be noted. First, while `minimal job` is building a program, the job has no knowledge that it is compiling anything. All it knows is the job has files for input, the command lines that need to be executed and expected output files. A different job could test `program` on `builder` to ensure `program` successfully runs.

Secondly, adding a source file does not scale nicely; source files must be added to both the input and as a `source file` path value. We will see how to fix that with a `file list` block later.

Thirdly, this job is completely self contained; it is unaffected by any changes to any other jobs in the same script. This is a good thing if building `program` requires unique options. However, this is rarely the case. We will see how to fix that with the `data` block later.

Fourthly, it would seem likely that a similar command line sequence to the one used in `minimal job` could be used for other jobs that are compiling other programs. When specified in a command block in the job, the command line sequence cannot be shared. We will see how to create common command line sequences with the `step` block later.

Fifthly, if building `program` was the only goal, a shell script would work nicely; any powerful build system like TinyBuilder would be overkill. More likely, `program` exists within a large ecosystem of other programs and libraries that also need to be built, along with a set of test programs to build and run automatically. We will see how to handle that with the `project` block later.

Data

Data blocks create name/value pairs within a job; this data is used within command blocks to specify commands and parameters. Within a command block, a name is specified between '<' and '>' which is expanded to the value(s) associated with the name. If the name has not been assigned the command or parameter is skipped. If the name has been assigned more than once, the command or parameter is repeated once per value, in assignment order. Any additional text on the same line as an expansion is part of the command or parameter and is repeated per assignment. Multiple expansions in the same line are permitted. Data assignments are created as the script is parsed and expansions are done as the job runs. As a result, assignments within the script may occur before or after the command blocks within the script and all assignments will still be used.

A name may be assigned data or path values, not both. Within a `job` block, data values are specified within a `values` block; path values are specified within a `path` block. The difference between data and path values are in how they are expanded. Data values are treated strictly

as text; expansion values are literal; no interpretation of the expansion text is made. Path value assignments tell the TinyBuilder client to expand each value as a path; valid path separators on the client are converted into valid path separators on the machine and extensions can be removed and replaced. Unlike data, where adjacent expansions simply append values, adjacent path value expansions are separated using valid path separators on the machine.

A comma separated list of expansion options may be specified within '<' and '>' within the `<' and '>' of an expansion. The following expansion options are supported:

`required`: If the name has no assignments, the command is skipped, not just the parameter. For example, if a compiler is invoked with no source file, it will fail with an invalid command line. If the `required` expansion option is specified in the source file expansion, the compiler will not be invoked

`environment`: The name does not specify a data name, instead, it specifies an environment value on the machine. For example, if the command were `<<environment>CC>` the command would be specifying the default C compiler on the server. An `environment` expansion must be by itself, with no accompanying text; the client does not know how to expand it since it is expanded on the machine; the client can only send the name of the environment variable to use.

`enumerate`: When specified within a parameter, the command is executed once per assignment. When specified within an executable name, the option is ignored. For example, the compiler is typically invoked a source file at a time; its expansion would use the `enumerate` expansion option. The linker, on the other hand, is passed a list of all the source files; its expansion would not use the `enumerate` expansion option.

`enumerate within` *`name`*: When enumerating more than one name, an order must be provided. All enumerations must be in a nesting order; there may not be any cycles in the order. The outermost enumeration is specified with the `enumerate` option. Each enumeration within the outermost enumeration must specify the expansion to expand within. For example, if `name1`, `name2` and `name3` each have two assignments, the command block:

```
command
    <<enumerate>name1>
    <<enumerate within name1>name2>
    <<enumerate within name2>name3>
```

which would result in the following command line sequence:

```
command name1-value1 name2-value1 name3-value1
command name1-value1 name2-value1 name3-value2
```

```
command name1-value1 name2-value2 name3-value1
command name1-value1 name2-value2 name3-value2
command name1-value2 name2-value1 name3-value1
command name1-value2 name2-value1 name3-value2
command name1-value2 name2-value2 name3-value1
command name1-value2 name2-value2 name3-value2
```

The `enumerate within` expansion option ensures clarity and control regarding how multiple `enumerate` expansions are expanded.

`enumerate along` *name*: When the same name is enumerated within the same command multiple times, the first expansion of the name must be have the `enumerate` expansion option and all subsequent expansions of the same name must be `enumerate along`. For example, if `name1` has two assignments, the command block:

```
command
    <<enumerate>name1>
    <<enumerate along>name1>
```

would result in the following command line sequence:

```
command name1-value1 name1-value1
command name1-value2 name1-value2
```

`directory name`: For path values, the file name value is stripped from the value's expansion. If the resulting path is empty, the assignment is discarded. The expansion option is ignored for data values.

`file name`: For path values, the directory part of the path is discarded. If the resulting path is empty, the assignment is discarded. The expansion option is ignored for the data values.

`base name`: For path values, the extension is stripped from the file. The expansion option is ignored for data values. New extensions may be specified as text after the expansion. The '.' is dropped along with the extension, so the text needs to specify a new '.'.

Assigning Data and Paths

The `data` block is used to share data and path values among multiple jobs. The `data` block is started with the text `data` with no indention. Immediately following the `data` term is the name of the data block. The name of the data block may be any text that can be expressed by UTF-8 except for the new line. The name is matched byte by byte; no case folding, composition or decomposition is done when comparing names.

To update the `minimal job` example earlier to use `data` blocks, the job could be:


```
data compiler options
    values
        optimization = -O3
        option = -m64
        option = -g
        option = -I.

data linker options
    values
        link options = -m64
        link options = -g

data compiled files
    paths
        source file = main.c
        source file = module1.c
        source file = module2.c

job data block job
    input
        main.c
        module1.c
        module2.c
        module1.h
        module2.h
    include data with name compiler options
        compiler options
    include data
        linker options
        compiled files
    paths
        executable = program
```

```
command break on error
    gcc
        <compiler options>
        -o
        <<base name, enumerate>source file>.o
        <<enumerate along>source file>
    gcc
        <link options>
        -o
        <executable>
        <<base name>source file>.o
output
    program
machine
    builder
```

In the above example, the settings have been moved into data blocks and so may be used by other jobs using `include data` blocks. When the `compiler options` data block is included into `data block job`, the assignments to `optimization` and `option` are changed to `compiler options` by the `with name` modifier. When the `linker options` and `compiled files` data blocks are included into the job, the names as well as the values are preserved. Within the `command break on error` block, the names between '`<`' and '`>`' are expanded, replacing the names with their assignments.

Just as within a `job` block, the `values` block within the `data` block is used to assign values to data names. An assignment within a `values` block may not assign to a name with existing path assignments. The name may include any character except '`=`' or the new line. The name is matched byte by byte; no case folding, composition or decomposition is done. The values assigned to a data value are not interpreted and always remain the same text as the assignment when they are expanded.

Just as within a `job` block, the `paths` block within the `data` block is used to assign path values to names. An assignment within a `paths` block may not assign to a name with existing data assignments. The name may include any character except '`=`' or the new line. The name is matched byte by byte; no case folding, composition or decomposition is done. Any path separators in a path value are changed to reflect the path separator used by the machine, but the path separator in the script must be valid for the client. At present, the '`/`' character is a valid path separator for all platforms supported by the TinyBuilder client.

A job always has a data block with zero or more names with assignments. It is possible to include the assignments of a `data` or a `file list` using an `include` block. The contents

of the `include` block is a list of `data` or `file list` block names. The assignments in the included data block are added to the names already in the `data` block of the job. The `with name` modifier may be used to assign all assignments in the included `data` block to the same name within the job data block. A `file list` may be included into a `data` block as a series of assignments to a path value when the file list is included using the `with name` modifier.

The assignments maintain their order, so when a name is expanded, it will be expanded in the same order as the assignments to the name. When a `data` or `file list` block is included, its assignments are made in the same position as the `include` block. Any number of `values`, `paths` and `include` blocks may be specified within a job; the assignments will be added and expanded in order.

The File List

Frequently, a set of files is used more than once within a TinyBuilder script. To allow one change to affect all users of the set of files, the `file list` block may be used. The file list is started with the text `file list` with no indention. Immediately following the `file list` term is the name of the file list. The name of the file list may be any text that can be expressed by UTF-8 except for the new line. The name is matched byte by byte; no case folding, composition or decomposition is done when comparing names.

To update the `minimal job` example earlier to use file lists, the job could be:

```
file list source files
    files
        main.c
        module1.c
        module2.c

file list header files
    files
        module1.h
        module2.h

job file list job
    include input
        source files
        header files
    values
        compiler options = -O3
        compiler options = -m64
```

```
        compiler options = -g
        compiler options = -I.
        link options = -m64
        link options = -g
    include data with name source files
        source files
    paths
        executable = program
    command break on error
        gcc
            <compiler options>
            -o
            <<base name, enumerate>source files>.o
            <<enumerate along>source files>
        gcc
            <link options>
            -o
            <executable>
            <<base name>source files>.o
    output
        program
    machine
        builder
```

In the above example, the source files have been moved to a file list and the headers have been moved to their own file list. Two lists are needed because the source files are both required as input to the job and are used as command line arguments to the compiler, while the header files are needed as input, and are not passed as compiler arguments. The `include input` block does not accept file names like the `input` block, instead, the contents of the `include input` block is a list of file list names. The `include data` block may include either a `file list` or a `data` block; in this case, it includes a file list into the job and assigns each path in the file list to the `source files` name.

The `file list` block may have zero or more `files` or `include` blocks. The contents of the `files` sub block is a list of files. The path of the files is relative to the directory of the script containing the `file list` block. The contents of the `includes` block is a list of `file list` block names. The files in the included file list are incorporated into the `file list` block in the position they were included. The path of the included files remain the same; inclusion does not change the path of files within the list; they remain relative to the script containing

the included block.

When included into a `data` block, the file list maintains its order, so when the path value is expanded, it will be expanded in the same order as the `file list`. When a file list is included, its files are inserted into the including file list in the same position as its `includes` block. Any number of `files` and `includes` blocks may be specified within a `file list` block; the files will still be added in order.

Path Concatenation

Path values will be concatenated into a single valid path when two adjacent path values are expanded. Each expansion is considered to be a member of the path, with path separators placed between the expansions, when there is no text between the expansions. For example:

```
job example job
    paths
        dir  = ../dir
        file = ../other_dir/file
    command break on error
        program
            --file=<dir><file>
```

However, the above example may not work as intended because of the way path values are expanded by TinyBuilder. In this case:

```
<dir><file>
```

becomes:

```
../dir/../other_dir/file
```

which becomes

```
../other_dir/file
```

The way to fix the expansion is to add the `file name` expansion option, so the line reads `--file=<dir><<file name>file>`. In this case, `dir` is still expanded to `../dir`, but

the directory is stripped from `file` so the expansion is `../dir/file` after concatenation.

Note that the problem will also occur in this example:

`../other_dir/files.tb:`

```
file list files
    files
        file
```

`main.tb:`

The Script

```
import ../other_dir/files.tb

job example job
    paths
        dir = ../dir
    include data with name file
        files
    command break on error
        program
            --file=<dir><file>
```

The problem occurs because the file list is in the `../other_dir` directory, so when the path is included into the job, the `files` path value is assigned `../other_dir/file`.

Defining Common Commands

The `step` block specifies a sequence of commands that may be shared among jobs. The step block is started with the text `step` with no indention. Immediately following the `step` term is the name of the step. The name may be any text that can be expressed by UTF-8 except for the new line. The name is matched byte by byte; no case folding, composition or decomposition is done when comparing names.

To update the `minimal job` example earlier to use a step, the job would be:

```
step compile with gcc
    parameters
        source files
        compiler options
```

```
        link options
        executable
    command break on error
        gcc
            <compiler options>
            -o
            <<base name, enumerate>source files>.o
            <<enumerate along>source files>
        gcc
            <link options>
            -o
            <executable>
            <<base name>source files>.o

job step job
    input
        main.c
        module1.c
        module2.c
        module1.h
        module2.h
    values
        compiler options = -O3
        compiler options = -m64
        compiler options = -g
        compiler options = -I.
        link options = -m64
        link options = -g
    paths
        source file = main.c
        source file = module1.c
        source file = module2.c
        executable = program
    include step compile with gcc
        source file
        compiler options
        link options
        executable
    output
        program
    machine
        builder
```

In the above example, the commands have been moved to a step where other jobs can make use of the same command sequence. The named values are passed as a list of parameters to the `compile with gcc step`. Each parameter in the step becomes an alias for the name in the same position in the `include step` block. For example, the `source files` parameter takes on the assignments of the `source file` path value in the job, the `compiler options` parameter takes on the assignments of the data value `compiler options`, and so on. The expansions within the step behave exactly as they do in a command block. Commands within the step are inserted into the command sequence of the job where the step is included.

The Script

The command block within the script specifies a series of commands that are to be executed in sequence with the same error handling. Each command block adds to the list of commands within the job; after parsing is complete, the job contains a single command sequence. The parser flattens the job structure as it parses the script; each command is given the error handling from its block. Commands that originated from steps are also copied into the job; those commands also receive their error handling from their block; the fact that the commands were included into the job is irrelevant. As a result, the error handling of any failed command is specified only by the owning block. The error handling of the block is specified after `command`; the possible values are as follows:

Flag	*Meaning*
`break on error`	If the return value of the command line is non-zero, the job is marked as failed and no other commands are executed.
`complete with error`	If the return value of the command line is non-zero, execution continues until the end of the block is reached; at which point the job fails, regardless of any subsequent return values.
`ignore error`	The return value of the command line is ignored; it is impossible for the job to fail based on a command return value.

The `step` block may also make use of the `include step` block. Just as when used in a `job` block, the step name is specified after the `include step` text. Parameters passed to the including step may be passed to the included step; only parameters to the including step may be passed to the included step. The parameters in the included step are aliased to the same job data names in the job including the including step.

Environment Blocks

Normally, each job inherits its environment variables directly from the service. Some tools, however, require that environment variables be setup in a particular way. To allow job specific

environment changes, environment blocks may be specified. All the changes to the environment are local to the job only; they do not affect the service or any other job. The value assigned to a name in the environment block may replace, prefix or suffix the environment value with that name. The environment changes are applied in replace, prefix, suffix order, regardless of the placement of the blocks or the order of the assignments.

To prefix a value to an environment variable, an `environment prefix` block is used. For example, to add `/root` to the beginning of the `PATH` variable, add to the job:

```
    environment prefix
        PATH = /root:
```


Note that in this case, the '`:`' is needed to separate the directory from the start of the current value of `PATH`.

To replace the value of an environment variable, an `environment replace` block is used. For example, to set the `PATH` variable to `/root`, add to the job:

```
    environment replace
        PATH = /root
```

Note no '`:`' is needed since no other directory will be in the `PATH` after the value is replaced.

To suffix a value to an environment variable, an `environment suffix` block is used. For example, to add `/root` to the end of the `PATH` variable, add to the job:

```
    environment suffix
        PATH = :/root
```

Note the '`:`' needs to be at the beginning of the suffixed value to separate the directory from the last directory in the path.

Using Visual Studio

The Visual Studio tools require a development command prompt to run. The difference between the development command prompt and a normal command prompt is a set of environment variables have been set to allow the tools to run correctly. If `cl.exe` is invoked without specifying

a `development environment` block, the executable will not be found.

A `development environment` block has the same effects as the `environment` blocks, in that the environment variables used within a job using a `development environment` block are different from other jobs. The difference is the environment variable changes are specified on the server side, not the client side. The Visual Studio installations may be in different locations from server to server, but the `development environment` block will have the same effect.

The Script

The Windows service install uses `vswhere` to find installations of Visual Studio and create a set of development environments that can be expected to work on the server. The name of each development environment follows the convention `VS.<version>.<architecture>`, with `version` set to the Visual Studio version and `architecture` set to the Visual Studio architectures that may be supported. At present, the version/Visual Studio mapping is as follows:

Version	*Visual Studio Product*
7	Visual Studio .NET 2003
8	Visual Studio 2005
9	Visual Studio 2008
10	Visual Studio 2010
11	Visual Studio 2012
12	Visual Studio 2013
14	Visual Studio 2015
15	Visual Studio 2017
16	Visual Studio 2022

The installation will only create development environments for the versions `vswhere` finds on the server.

The `architecture` will be one of:

Architecture		
x86	x86.store	x86.uwp
x64	x64.store	x64.uwp
arm	arm.store	arm.uwp
arm64	arm64.store	arm64.uwp
amd64		

The installation does not check that the Visual Studio installation can support all the possible

architectures; all will be added unconditionally.

The `development environment` block is specified by the text `development environment` followed by the development environment name. Only one `development environment` block may be specified in a job. For example, to build a 64 bit executable with Visual Studio 2019:

```
job 64 bit Windows program
    development environment VS.16.x64
```


The development environment is only supported on Windows; the job will fail if a development environment is set in a job sent to a non-Windows machine or if the development environment is not setup by the service installation. To make use of a new Visual Studio installation on the Windows server, the TinyBuilder service will need to be reinstalled.

Retrieving Files from Failed Jobs

Sometimes when a job failed, files produced by the job are needed to debug the job failure, e.g. intermediate test data or core files. Such files can be listed in the `failed output` block.

When the commands of a job all succeed, the job will attempt to download all files listed in the `output` block and no attempt is made to obtain any of the files in the `failed output` block. If any file listed in the `output` block is missing from the work area, the job will fail.

When a job failed, no attempt is made to download the files in the `output` block, but an attempt is made to obtain each file in the `failed output` block. If a file in the `failed output` block is missing, the file is skipped and the rest of the list will be transferred to the client

Specifying Common Machines

The machine block is started with the text `machine` with no indention. Immediately following the `machine` term is the name of the machine; the name is treated distinctly by TinyBuilder; a `machine` block name may be the same as the name of another block such as a `job` or `data` block, but not the same as the name of another `machine` block. The name may be any text that can be expressed by UTF-8 except for the new line. The name is matched byte by byte; no case folding, composition or decomposition is done when comparing names. The name has no relation to any name/IP address resolution; it is never passed to the operating system. If the name of the `machine` block is the same as a host name, the `machine` block will be used, not the host

name, when the host name is specified within a `machine` block within a `job` block.

Unlike the `machine` block within a `job` block, where the name is simply a host name or `machine` block name, each machine within the `machine` block is specified as a URL. The scheme specifies one of the protocols supported by TinyBuilder. The host name of the URL is used to determine the IP address of the machine; a port may be specified after the host name using the '`:`' character. If no port is specified, the default TCP port 5017 is used. None of the protocols support authentication or a path in their URL's.

The `machine` block contains a list of target machines, each specified by a URL; TinyBuilder will load balance among the list. Each target machine is reached by a path, and each path consists of one or more hops. Each `path` block within the `machine` block contains a path of hops; the last hop in the list is the target machine. Requests and responses will pass through each machine in the list, to be serviced by the target machine. For example, to reach the agent running on `buildserver1`, the `interactive builder` machine block would use the following:

```
machine interactive builder
    path
        tb://buildserver1
        tbi://localhost
```

Most paths, however, are single hop paths. The `path list` block may be used to list a set of single hop paths. Within the `path list` block, all the URL's are target machines acting as part of a server pool for load balancing. For example, the following `builder` block would randomly assign jobs to one of two servers, `buildserver1` or `buildserver2`:

```
machine builder
    path list
        tb://buildserver1
        tb://buildserver2
```

It is possible to specify a single hop path using a `path` block. For example:

```
machine builder
    path list
        tb://buildserver1
        tb://buildserver2
```

would be equivalent to:

```
machine builder
    path
        tb://buildserver1
    path
        tb://buildserver2
```

Either a `path` block or a `path list` block may be used to specify one single hop path; the `path list` block is simply a more compact alternative when specifying more than one single hop path for load balancing.

It is useful to know exactly which machine ran a job in the build log; the script does not specify which path will be used when more than one path is provided in the `machine` block. Within the build log, each job is given a path ID, which together with the machine block name, uniquely identifies each path in the script. The path ID is an integer, starting with zero, and incremented for each path specified in the block. In the last example, `tb://buildserver1` would be given a path ID of zero, and `tb://buildserver2` would be given a path ID of one.

The URL's may be one of three schemes:

Scheme	*Protocol*
tb	The TinyBuilder protocol is used to directly connect to the TinyBuilder machine. The communication is in plain text; packet capturing could be used to recover all source in the input archives and all the binaries in the output archives, as well as all output from any commands executed. A man in the middle attack could silently change requests to the machine and responses from the machine. No authentication is done. The connection relies completely on network security. This scheme is used by default.
tbs	SSH port forwarding is used to connect to the TinyBuilder server over an SSH tunnel. The connection is as secure as SSH port forwarding. Authentication is provided by SSH. This requires `sshd` to be installed and configured on the server and `ssh` installed on the client. The TinyBuilder client will generate an `ssh` command line to establish a tunnel between itself and `sshd` on the server.
tbi	The URL specifies an agent; the host name of the URL must be localhost. No port may be specified. The agent runs in the UI context of the user logged into the machine specified by the previous hop in the path. This scheme may not be used as the first hop in any path. See The Agent chapter of this user guide for more details.

Specifying a Machine for a Job

The `machines` block in the `job` block specifies the TinyBuilder machine(s) to run the job. TinyBuilder uses the term machine instead of server because while the client will connect to computers that could reasonably be called servers, it may also connect to small devices for testing. The term machine emphasizes the broad range of connection targets.

The `machine` block is required within the `job` block; every job must have at least one machine path. The simplest way to specify the machine running the job is to specify a host name or IP address in the `machine` block within the `job` block. When specified this way, the job will run over an insecure, default protocol connection with a single hop to the address specified within the block. If the `machine` block within the `job` block specifies multiple machines, the job will run on all the specified machines.

The `machine` block within the `job` block may also specify the name of a `machine` block, which is a block that may be shared among jobs. A `machine` block permits the specification of the protocol used to connect along with any hops that may be necessary to reach the target machine. When an IP address or host name is provided within the `job` block's `machine` block, the parser automatically creates a default `machine` block with the same name as the address. If the default protocol is no longer desired, a `machine` block may be specified explicitly with the same name as the host name, and it will be used instead.

The address specified in the `machines` block is resolved to an address using the operating system. For example:

```
machine test machine 1
    paths
        tb://test1

job test job
    machines
        test machine 1
        test2
```

In the above example, `test job` will be run on both servers with the DNS names `test1` and `test2`.

Effectively, a `machine` block name overrides a host name when used within a `job` block. For example, to redirect the `builder` server used in `minimal job` to use the loopback, provide

this `machine` block before the job:

```
machine builder
    path
        tb://localhost
```

Note that the job block does not change when it is redirected to another server using a `machine` block.

To have `minimal job` run on one of two servers using SSH port forwarding, the following `machine` block may be used:


```
machine builder
    path list
        tbs://builder
        tbs://builder-backup
```

Controlling Machine Utilization

The TinyBuilder service is a finite resource and if it is over utilized, various scripts and programs used as part of the build will fail due to lack of resources. Frequently, the result is a mysterious, intermittent build failure with strange or non-existent error messages. To prevent these failures, the TinyBuilder service protects itself against usage spikes by limiting the number of concurrent jobs that may run on the machine.

Some jobs are more resource intensive than others. For those jobs that are unusually resource intensive or those that are unusually quick, a `concurrency` block may be specified. The `concurrency` block is a single line block with no sub block. For a typical compilation job, no `concurrency` block is necessary. A `job` block may have zero or one `concurrency` blocks, which specifies how much of the machine should be consumed by a job. If the job, along with the other jobs already running on the server, would consume more than the entire machine, the job is delayed until the server has the capacity to start the job.

For example, a highly resource intensive test job could look like this:

```
job program test
    concurrency minimum
    input
        program
```

```
commands break on error
    program
        test-mode
machine
    builder
```

In the concurrency context, the machines utilized are actually called abstract servers; the service installation configures each core on the machine to act as an abstract server. Agent utilization is separate from the service utilization. See the TinyBuilder Service chapter of the reference guide for more details.

The valid concurrency values are:

Value	*Abstract Server Percentage Consumed*
`minimum`	100%
`low`	50%
`medium` (default)	9.8%
`high`	3.9%
`maximum`	0.4%

Projects

The `project` block permits jobs and projects to be consolidated so that when a project is run, all of the jobs contained within the project may be run. The project block is started with the text `project` with no indention. Immediately following the `project` term is the name of the project. The name may be any text that can be expressed by UTF-8 except for the new line. The name is matched byte by byte; no case folding, composition or decomposition is done when comparing names.

Unlike the other blocks, the order of jobs within the project are not taken into account; all jobs that can run are run immediately. If an input file to the job is the output of another job within the project, the job requiring output from a job will wait until the output is generated. If the job creating the output failed, the job requiring the input file will not run, even if an older version of the input file is available. Do not create jobs where an input file and an output file have the same path; the job will never run in this case. That includes larger cycles, such as when job A creates a file needed by job B and job B creates a file needed by job A; neither job A nor B will ever run.

By default, the TinyBuilder client observes what is referred to as make scheduling. Like the venerable utility `make`, when using make scheduling, if all input is older than the oldest output of a job, the job is considered up to date and is not run; such a job is called make complete. When a

job does not need to run, any output is considered to be up to date, so any jobs depending on the output will be considered ready. However, make scheduling does not take the modification date of any TinyBuilder script into account, so any script changes will require the job to be rebuilt.

By default, the TinyBuilder client will run the project or job named `main`. If this is not preferred, a job name can be specified on the TinyBuilder client command line, in which case, the job will run, even if it is up to date according to make scheduling. If a project is specified on the command line, the associated jobs will be run according to make scheduling. If a project is specified on the command line and the `--rebuild` command line option is specified, all jobs in the project are run, even if they are up-to-date. The `--rebuild` command line option will not force a job to run if its input could not be built, but specifying the job explicitly on the command line will run the job if all input is present, regardless of the input file modification time.

The `project` block contains zero or more `builds` or `tests` blocks. Jobs listed within the `builds` block are executed according to make scheduling. The jobs listed in the `tests` block are never considered to be up to date when make scheduling is in use. The intent of the `tests` block is to specify jobs that are to be run even if nothing is to be built, such as jobs running automated tests. If a project specified in the `tests` block has its own `builds` block, the jobs in the included block will behave as if they are in a `tests` block.

Importing Other Scripts

A script may be imported into another script. When this is done, it is as if the text from the imported script brought into the importing script, similar to the C `#include` preprocessing directive.

The `import` block is used to import scripts. The `import` block is started with the text `import` with no indention, followed by a file path. The file path is relative to the script containing the `import` block. There is no content in the `import` block. If a script is imported more than once, only the first import attempt will be done; all subsequent attempts to import the same script will be silently ignored.

Comments

If the first non-whitespace character in a line is the `#` character, then the line is a comment. For example:

```
data data block
# This is a comment
    # This is also a comment
```

```
    values
        name = value # This is part of the data value
# This comment does not terminate the values block
        name = value2
```

Command Generation

This chapter describes how command lines are generated while jobs are running. A command is a member of a command block in the script. A command line is an executable name and command line arguments passed to the server and executed. Each command line is derived from the command along with data values and path values just before the command is run. While the command is running, no other command may run within the job. After a command is complete, its return value is used for any error handling.

When the conversion of a command to a set of command lines starts, each name is searched in the job's data assignment database; each name is replaced by the assignments to that name. If the name is not found among the assignments in the job, the name is considered to have no assignments, and no failure occurs. For example, the following job:

```
job example 1
    commands break on error
        command
            <unset name>
    machine
        localhost
```

would execute the command line:

```
command
```

The `required` expansion option will cause the command to be skipped if there are no expansions for the name. For example:

```
job example 1a
    commands break on error
        command
            <<required> unset name>
    machine
        localhost
```

would succeed without executing any commands. The `enumerate` expansion option has the same results, but for a different reason. The `required` expansion option results in one or zero command lines. The `enumerate` expansion option creates one command line per expansion, zero being a valid number of expansions.

A command line parameter may also contain text. The text is replicated per expansion, and used once if there is no expansion. For example:


```
job example 2
    commands break on error
        command
            --file=<unset name>
    machine
        localhost
```

would execute the command line:

```
command --file=
```

While the job:

```
job example 2a
    paths
        file = file1
    commands break on error
        command
            --file=<file>
    machine
        localhost
```

would execute the command line:

```
command --file=file1
```

and the job:

```
job example 2b
    paths
```

```
        file = file1
        file = file2
    command break on error
        command
            --file=<file>
    machine
        localhost
```

would result in the command line:

```
command --file=file1 --file=file2
```


It is permissible for a command line parameter to only be text. For example:

```
job example 3
    commands break on error
        command
            --file
            <unset name>
    machine
        localhost
```

would execute the command line:

```
command --file
```

While the job:

```
job example 3a
    paths
        file = file1
    commands break on error
        command
            --file
            <file>
    machine
        localhost
```

would execute the command line:

```
command --file file1
```

and the job:

```
job example 3b
    paths
        file = file1
        file = file2
    command break on error
        command
            --file
            <file>
    machine
        localhost
```


would result in the command line:

```
command --file file1 file2
```

The last example demonstrates that there is no relationship between the command line parameters; the previous parameter is not replicated per expansion.

Note that the space is simply text within the command line parameter; it has no special significance. A mistake that is easy to do is the following:

```
job example 3c
    paths
        file = file1
        file = file2
    command break on error
        command
            --file <file>
    machine
        localhost
```

which would result in the command line:

```
command '--file file1' '--file file2'
```

when the intent was to make `--file` and the file name separate parameters like `example 3b`.

The `enumerate` expansion option causes TinyBuilder to create a command line per expansion of the name; zero expansions is an acceptable number of expansions. Other expansions within the same command line are unaffected by the `enumerate` expansion option and are repeated for each command line. For example:


```
job create archive 0
    paths
        files = file1
        files = file2
        files = file3
    command break on error
        tar
            -czvf
            <<enumerate> archive>
            <files>
    machine
        localhost
```

The `create archive 0` job would succeed without executing any commands since `archive` has no assignments.

```
job create archive 1
    paths
        files   = file1
        files   = file2
        files   = file3
        archive = output.tgz
    command break on error
        tar
            -czvf
            <<enumerate> archive>
            <files>
    machine
        localhost
```

Would execute one command line:

```
tar -czvf output.tgz file1 file2 file3
```

and the following:

```
job create archive 2
    paths
        files   = file1
        files   = file2
        files   = file3
        archive = output1.tgz
        archive = output2.tgz
    command break on error
        tar
            -czvf
            <<enumerate> archive>
            <files>
    machine
        localhost
```


would execute two command lines:

```
tar -czvf output1.tgz file1 file2 file3
tar -czvf output2.tgz file1 file2 file3
```

It is possible for a single command line to have multiple `enumerate` expansions, but one must be nested within the other using the `enumerate within` expansion option. The first command line will use the first expansion of the outer and inner `enumerate` expansions, then the first outer and second inner expansions, continuing until all the inner expansions are exhausted. Then the second outer and first inner expansions are used. This continues until all the possible expansions are used. If any name has no expansions, no command lines are executed. Embedding `enumerate` expansions within other `enumerate` expansions may be done to any depth.

For example, the following job will not execute any command lines:

```
job create archive directories 0
    paths
        files   = file1
        files   = file2
        files   = file3
```

```
        archive = output1.tgz
        archive = output2.tgz
    command break on error
        tar
            -C
            <<enumerate> directory>
            -czvf
            <<enumerate within directory> archive>
            <files>
    machine
        localhost
```


The following will execute two command lines:

```
job create archive directories 2
    paths
        files     = file1
        files     = file2
        files     = file3
        archive   = output1.tgz
        archive   = output2.tgz
        directory = subdir1
    command break on error
        tar
            -C
            <<enumerate> directory>
            -czvf
            <<enumerate within directory> archive>
            <files>
    machine
        localhost
```

and the command lines are:

```
tar -C subdir1 -czvf output1.tgz file1 file2 file3
tar -C subdir1 -czvf output2.tgz file1 file2 file3
```

and the following will execute four command lines:

```
job create archive directories 4
    paths
        files     = file1
```

```
        files     = file2
        files     = file3
        archive   = output1.tgz
        archive   = output2.tgz
        directory = subdir1
        directory = subdir2
    command break on error
        tar
            -C
            <<enumerate> directory>
            -czvf
            <<enumerate within directory> archive>
            <files>
    machine
        localhost
```

and the commands will be:

```
tar -C subdir1 -czvf output1.tgz file1 file2 file3
tar -C subdir1 -czvf output2.tgz file1 file2 file3
tar -C subdir2 -czvf output1.tgz file1 file2 file3
tar -C subdir2 -czvf output2.tgz file1 file2 file3
```

and we can reverse the `enumerate within` option to make the outer enumeration the inner enumeration:

```
job create directory archives 4
    paths
        files     = file1
        files     = file2
        files     = file3
        archive   = output1.tgz
        archive   = output2.tgz
        directory = subdir1
        directory = subdir2
    command break on error
        tar
            -C
            <<enumerate within archive> directory>
            -czvf
            <<enumerate> archive>
            <files>
```

```
    machine
        localhost
```

which will execute:

```
tar -C subdir1 -czvf output1.tgz file1 file2 file3
tar -C subdir2 -czvf output1.tgz file1 file2 file3
tar -C subdir1 -czvf output2.tgz file1 file2 file3
tar -C subdir2 -czvf output2.tgz file1 file2 file3
```


Sometimes, the `enumerate` expansion needs to happen more than once within the same command line. To accomplish this, the `enumerate along` expansion option is used. For example:

```
job enumerate along example
    paths
        files     = file1
        files     = file2
        files     = file3
        directory = subdir1
        directory = subdir2
    commands break on error
        cat
            <<enumerate>directory><files>
            <<enumerate along>directory><files>
    machine
        localhost
```

Note that the script has no path separator between the `directory` and `files` expansions. This is because they are path values and TinyBuilder will insert the correct path separator for the machine running the job. This is discussed in more detail later in this chapter. The above job will execute:

```
cat subdir1/file1 subdir1/file2 subdir1/file3 subdir1/file1\
subdir1/file2 subdir1/file3
cat subdir2/file1 subdir2/file2 subdir2/file3 subdir2/file1\
subdir2/file2 subdir2/file3
```

When using `enumerate along`, exactly one expansion must have the `enumerate`

expansion option, all of the rest must have the `enumerate along` expansion option.

When a non-`enumerate` expansion is expanded a command line option is added per expansion. So in the above example, each `files` assignment is expanded as part of the `<<enumerate>directory><files>` command line options before the next set of command line options are added. The `files` expansion is repeated for each command line option that uses the value; the same `directory` expansion is used due to the `enumerate along` expansion option.

If we remove the `enumerate` expansion option from the previous example, we get:


```
job multiple expansions example
   paths
        files     = file1
        files     = file2
        files     = file3
        directory = subdir1
        directory = subdir2
    commands break on error
        cat
            <directory><files>
            <directory><files>
    machine
        localhost
```

A single command line with the following command line parameters will be executed:

```
subdir1/file1    # expansions from the first line
subdir1/file2
subdir1/file3
subdir2/file1
subdir2/file2
subdir2/file3
subdir1/file1    # expansions from the second line
subdir1/file2
subdir1/file3
subdir2/file1
subdir2/file2
subdir2/file3
```

When multiple non-`enumerate` expansions are done within the same command line parameter, the leftmost expansion is the outer expansion, and the rightmost expansion is the inner expansion. All possible expansions are done. Unlike `enumerate` expansions, command line parameters are created when one of the names has no expansions. For example:


```
job multiple expansions example 2
   paths
        files     = file1
        files     = file2
        files     = file3
    commands break on error
        cat
            <directory><files>
            <directory><files>
    machine
        localhost
```

would execute the following command:

```
cat file1 file2 file3 file1 file2 file3
```

TinyBuilder makes a distinction between path values and data values because the expansions have different meanings. For example, if the `multiple expansions example` is changed to make `files` and `directory` data values instead of path values, the script would look like this:

```
job multiple values expansions example
   values
        files     = file1
        files     = file2
        files     = file3
        directory = subdir1
        directory = subdir2
    commands break on error
        cat
            <directory>/<files>
            <directory>/<files>
    machine
        localhost
```

Now, a path separator is necessary or the expansions will be similar to `subdir1file1`. However, while TinyBuilder knows the correct file separator to use, the script may not. It is possible for a single job to be sent to both a Windows and a Linux server. If path values are used, the paths sent to the servers would be correct. If data values are used, the '/' string will be used as a path separator on Windows, which can cause problems on some command lines.

Furthermore, if the directory assignments are removed, the command line would become:


```
cat /file1 /file2 /file3 /file1 /file2 /file3
```

which is obviously incorrect; the command line would have been correct if path values were used.

A small adjustment to the previous example creates a different problem:

```
job multiple parent values expansions example
   values
        files     = file1
        files     = file2
        files     = file3
        directory = ../subdir1
        directory = ../subdir2
    commands break on error
        cat
            <directory>/<files>
            <directory>/<files>
    machine
        localhost
```

Since `directory` is a data value, TinyBuilder knows nothing about its meaning. The current directory for `cat` will be the root of the work area. When the command lines are expanded, they will point to paths that are outside of the job's work area. This is incorrect. However, if path values are used, TinyBuilder knows that the path value `../subdir1` is used and the parent of the current directory must be the root of the work area.

In other words, when `directory` was set to `subdir1`, TinyBuilder knew that it was safe to make the current directory the root of the work area, since no path value, `input` block or `output` block referred to a parent directory. When `directory` was a path value with the assignment `../subdir1`, TinyBuilder knew that the job referred to the parent directory, and so made it the root of the work area. However, when `directory` is a data value, TinyBuilder

makes no assumptions about the assignments to that name, and as a result, the work area will be constructed incorrectly.

When the parser parses a path value, each path value becomes a sequence of path segments; each path segment is separated by the path separator; a path segment never includes a path separator. The parser doesn't know what kind of machine will be ultimately using the value within the job, so it uses the client concept of what may be used as a path separator. For example, the '\' character will function as a path separator when parsed by a Windows client, but the Linux client will treat the character as part of the path segment.

When path values are concatenated, the sequences of path segments are concatenated. The resulting sequence can then be normalized. For example:


```
job path concatenation example
    paths
        directory = bin
        archive   = ../archive.tgz
        files     = file1
        files     = file2
        files     = file3
    commands break on error
        tar
            -czvf
            <directory><archive>
            <files>
    machine
        localhost
```

The command line would be:

```
tar -czvf archive.tgz file1 file2 file3
```

When `directory` and `archive` are concatenated, the sequence of path segments are `bin`, `..` and `archive.tgz`. When the sequence is normalized, the path segment `..` is dropped along with the previous segment, `bin`; only `archive.tgz` remains.

The `directory name` expansion option allows a command line option to use the same directory as another command line option, without using the file. For example:

```
job archive binaries example
    paths
```

```
        archive     = ../bin/binaries.tgz
        executables = util1
        executables = util2
        executables = util3
    command break on error
        tar
            -C
            <<directory name> archive>
            -czvf
            <<enumerate>archive>
            <executables>
    machine
        localhost
```

would result in the command line:

```
tar -C ../bin -czvf ../bin/binaries.tgz util1 util2 util3
```

The `directory name` expansion option works by removing the last path segment from the expansion. In the above example, the `-C` command line option will always be given the directory of the `archive` expansion. However, care must be taken to specify a directory within any assignment when the `directory name` expansion option is used. For example:

```
job bad archive binaries example
    paths
        archive     = binaries.tgz
        executables = util1
        executables = util2
        executables = util3
    command break on error
        tar
            -C
            <<directory name> archive>
            -czvf
            <<enumerate>archive>
            <executables>
    machine
        localhost
```

would result in an invalid command line:

```
tar -C -czvf ../bin/binaries.tgz util1 util2 util3
```

since an empty expansion is not an expansion. To fix this, set `archive` to `./binaries.tgz`. Alternatively, adding the `required` expansion option would cause the job to succeed without executing any commands:

```
job tar not run example
    paths
        archive     = binaries.tgz
        executables = util1
        executables = util2
        executables = util3
    command break on error
        tar
            -C
            <<required, directory name> archive>
            -czvf
            <<enumerate>archive>
            <executables>
    machine
        localhost
```


In the above example, the `archive` expansion is discarded due to the `directory name` expansion option. Since the expansion is discarded, the `required` option triggers and the command is discarded, even though an assignment was made to `archive`.

The `file name` expansion option is the opposite of the `directory name` expansion option; while the `directory name` expansion option drops the last path segment, the `file name` expansion option drops all but the last path segment. It is used to repeat a file name in another directory. For example:

```
job compile source 1
    paths
        src = ../src/hello.c
    command break on error
        gcc
            -c
            -o
            ../obj/<<enumerate, file name> src>.o
```

```
            <<enumerate along> src>
    machines
        localhost
```

The job will execute:

```
gcc -c -o ../obj/hello.c.o ../src/hello.c
```

Command Generation

The assignment to `source files` creates a sequence of three path segments, `..`, `src` and `hello.c`. Its first expansion leaves on the path segment `hello.c`. The second expansion converts the entire path segment sequence into `../src/hello.c`.

However, the object file's name is `hello.c.o`; while still usable, the `.c` is redundant. To remove it, the `base name` expansion option can be used. This expansion option acts upon the last path segment and removes the last '`.`' along with all subsequent text. This example is the same as the previous, but makes use of the `base name` expansion option:

```
job compile source 2
    paths
        src = ../src/hello.c
    command break on error
        gcc
            -c
            -o
            ../obj/<<enumerate, file name, base name> src>.o
            <<enumerate along> src>
    machines
        localhost
```

This job will execute:

```
gcc -c -o ../obj/hello.o ../src/hello.c
```

When used together, the `directory name`, `file name` and `base name` expansion options all work on the same path segment sequence; effectively, there is no order of operations. When `directory name` and `file name` are used together, the sequence would always be empty; the parser considers this situation to be invalid and will fail. When `directory name` and `base name` are used together, the changes made by `base name` are lost. As a result, if both options are used on the path `x86.opt/file.c`, the expansion will be `x86.opt`, never `x86`. If there is no extension, the `base name` option has no effect. So `x86.opt/file` will

be expanded as `x86.opt/file` when the `base name` option is used. If there is no directory, the `file name` option has no effect. So `file` will remain `file` in its expansion.

The Build Log

When the client runs, it generates a build log named `build_log.xml`. The log is an XML document that is designed to be human and machine readable with the following objectives:

Everything pertaining to a build is in a single file and may be easily found.

Everything is traceable. If anything failed, the root cause of the failure should be obvious.

All failures are recorded in the build log.

Everything that is observable when running a command line program is observable when the command line is run through TinyBuilder.

The entire output of a command may be reconstructed from the log, byte for byte, regardless of encoding.

The root element of the build log is `BuildLog`. The XML is UTF-8 encoded; while output from other encodings is accurately encoded, UTF-8 text is the easiest to read. The structure of the log is as follows:

```
root
   connection list
      connection
         hop
            error
   job list
      job
         command list
            command
               parameter list
                  parameter
               stdout
               stderr
               return
         output list
            output
            output error
```

The connection list, job list, command list, parameter list and output list are conceptual; they do not have corresponding elements in the XML. Rather, they are a list of zero or more elements of the same type. The elements in the connection and job lists have no particular order, even relative to each other. The command list is in execution order within each job. The parameter list is in the same sequence as the command line for each command. The directories containing the files in the output list of the job appear before the files; the files have no particular order.

The Connection List

The connection list documents each connection made by the client. Each connection is specified by a `machine` block in the script, which is reflected in the log by the `machine` element. The `name` attribute of the `machine` element names the `machine` block. The block allows multiple paths to be specified for load balancing, so the `PathID` attribute is used to identify the path within the block. The first path has the value zero, the second path has the value one and so on. If there is no `machine` block corresponding to the machine specified in the `job` block, the `name` attribute will be the name of the machine in the `job` block and the `PathID` attribute will be zero.

Since each path is a series of hops, each hop is documented using the `hop` element. The `hop` element contains attributes to specify the URL, IP address and service version information of each hop. If the connection to the hop fails, the `hop` element is given an `error` element as a child. The `error` element contains attributes to specify the type of the error, the time of the error and possibly any operating system error code. The `error` element has the error message as its contents.

There is no significance to where each `machine` element appears in the log.

The Job List

A `job` element is created in the build log for every job executed; jobs will not appear in the log for the following reasons:

The job is make complete.

The job is not part of a scheduled project.

A failure occurs while connecting to the job's machine.

The job building a file used as input to the missing job did not run or failed.

The `job` element may have the following attributes:

`name`: The name of the job from the script.

`machine`: Set to the value of the `name` attribute of the `machine` element documenting the connection used by this job.

`PathID`: Set to the `PathID` of the `machine` element documenting the connection used by this job.

`status`: Will be set to either `succeeded`, `failed` or `error`. If the value is `succeeded`, the output of the job was transferred to the client and other jobs that depend on that output may be scheduled. If the value is `failed`, no output from an `output` block was transferred and no jobs depending on the failed job's output may run. If the status is `error`, a TinyBuilder error occurred while running the job and no jobs depending on the job's output may run.

`RunningTime`: The number of seconds the job was running, which includes the input archive transfer, executing all commands, and the output archive transfer. The `RunningTime` does not include the `DelayTime`.

`concurrency`: Set to the concurrency of the job as specified by the script. If no concurrency was specified in the script, this attribute is set to the default, `medium`.

`DelayTime`: The number of seconds the job was delayed because the server was too busy. This attribute is not present if the job was not delayed. This time is not included in `RunningTime`.

`DevelopmentEnvironment`: If the job is using a development environment, this attribute is set to the name in the script. Otherwise, the attribute is not set.

If the job specifies changes to its environment, the first children of the `job` element describe the changes made. Each of the elements have a `name` and `value` attribute and are called `PrefixEnvironment`, `SuffixEnvironment` and `ReplaceEnvironment`. The elements appear in the log in the same order as the changes were applied.

The Command List

Each job has a list of commands it executed; each stored in a `command` element. Each command has three sets of children, the parameter list, the output and the return value. The parameter list describes each parameter passed to the command, in order. The output records the exact stdout and stderr; it is assumed to be UTF-8 text, but every byte can be retrieved from the log, regardless of encoding or if it is even text. The return value records the process return value along with a

precise execution time. If the machine running the command supports signals, it will record a signal value if the process was terminated due to a signal.

The `command` element has the following attributes:

`executable`: The executable used for the command. If a relative path is specified, the path is relative to the work area directory. If no path is specified, the `PATH` environment value is used.

`directory`: The current directory used by the command. This will always be a path relative to the work area directory.

`ExecutableFromEnvironment`: The executable may be specified using an environment variable on the machine. When this is done, the environment variable name is documented in the log as this attribute. Otherwise, this attribute is not set.

The first children of the `command` element are the `parameter` elements, which specify each command line option in their `value` attribute. Each element is in the same order as the command line parameters passed.

The command output is recorded in a sequence of `out` elements for stdout and a sequence of `err` elements for stderr. While the order of `out` elements is chronological and the order of `err` elements is chronological, the ordering of `out` elements relative to `err` elements is not specified. The number of UTF-8 code points is limited to help maintain human readability of the log. The attributes of the `out` elements and `err` elements are:

`offset`: The number of bytes of output from the command before this element.

`EOL`: The Unicode name of the code point that terminated this element. This attribute is not present if the element exceeded its code point limit.

The last child the `command` element is the `return` element or the `signal` element. The purpose of these elements is to document how the command terminated. If the job failed with the status `error`, the element may be missing. These elements may have the following attributes:

`value`: The return value of the process if the element is a `return` element, or the signal number if the element is a `signal` element. If the `value` attribute is non-zero, the command has failed. Whether this causes the job to fail depends on the error handling specified by the command's block.

`elapsed`: The number of seconds between the start of the command and its termination.

Output Tags

While TinyBuilder assumes all output is UTF-8, output is never lost, regardless of its encoding, or if it is even text. Since the build log still must always contain valid XML, TinyBuilder is constrained by the requirements of the format.

First, a valid UTF-8 encoded XML file must only contain valid UTF-8 code points. Unicode specifies bit patterns that must be adhered to and some bit patterns cannot be coaxed into utf-8. TinyBuilder is designed to never lose data, so it will never convert an invalid code point to the REPLACEMENT CHARACTER, drop the byte, or fail a command. Instead, it will embed an `InvalidByte` tag into the text with a `value` attribute set to the hexadecimal value of the byte.

Second, the XML standard contains restrictions on the code points the document may contain, such as the NUL code point must be excluded. TinyBuilder converts these code points into `CodePoint` tags with a `value` attribute set to the hexadecimal value of the code point. While the `InvalidByte` tag is limited to 0x00 to 0xff, the `value` attribute of the `CodePoint` tag is limited to the range of valid Unicode code points.

TinyBuilder is very efficient when gathering the output of commands running on the machine. As a result, the stdout/stderr output becomes a wide pipe and a viable logging method when running within a job. When used this way, the output time becomes important. To track the output time, each block of bytes received by the TinyBuilder machine is assigned an `elapsed` tag with the following attributes:

`seconds`: The number of seconds elapsed since the beginning of the command when the following output was received.

`offset`: The number of bytes of output received before this tag.

`EOF`: The bytes following this tag are the final bytes emitted by the command before it closed the output.

Effectively, the `elapsed` tag span the `out` and `err` elements containing the tags; if the text of an `out` element has no `elapsed` tags, the entire text of the `out` element was written at the time specified by the last `elapsed` tag occurring in a previous `out` element.

The `offset` attribute only counts the bytes received within the output stream. The `offset` attribute of an `elapsed` tag within an `out` element does not reflect any stderr bytes

received at that time.

While the TinyBuilder machine has substantial buffering to ensure high efficiency, the buffering is not infinite. If the TinyBuilder machine would run out of buffer space, it will prioritize getting every byte over never blocking the command. The command will then block if it attempts to write more output and the subsequent `elapsed` tags will be affected. When the machine runs out of buffer space, it will insert a `throttle` tag into the output. The `throttle` tag has a `ThrottleOnElapsed` attribute which records the seconds after the command started when the machine began to throttle the output. The time throttling ended is recorded by the next `elapsed` tag.

The Output List

The output files transferred from the machine is documented at the end of the job with a series of `output` elements. These elements have no attributes; their contents are the path of the output relative to the work area directory. If an output file could not be transferred, an `OutputError` element will document the failure. In this case, the job fails and the client attempts to transfer the files in the job's failed output list. If a file from the failed output list cannot be transferred, the error is ignored.

The Client

The client is the application used by each developer to perform their builds from their own machines. It has no setup; all of its state is stored in the scripts it runs. It is designed to work with the scripts consistently, regardless of the platform. While it only has a command line interface, it has a UI to display its status when a terminal is available. The client produces a build log, `build_log.xml` in the current directory; see The Build Log chapter of this guide for information regarding the log's contents.

This chapter describes the use of the TinyBuilder client, `tbuild` or `tbuild.exe`. This includes an explanation of its command line interface, its user interface, and how it integrates with `ssh`.

The Command Line Interface

The command line options are as follows:

Name	*Parameter*	*Description*
`--job`	A job or project name	Specifies the name of the job or project to run. If not specified, the job or project named `main` will run. If the name specifies a project, the jobs in the `build` block will run according to make scheduling. If the name specifies a job, the job will be run regardless of its scheduling status.
`--rebuild`	None	When building a project, the jobs in the `build` block will be run regardless of their scheduling state. When building a job, the parameter has no effect.
`--automated`	None	When specified and no job completes within a half hour, the client will consider all jobs hung and will abort all running jobs.

Name	*Parameter*	*Description*
`--server-user-list`	SSH destination list	Specifies a list of what would be destination parameters on the `ssh(1)` command line. If the `ssh(1)` command line is configured to not require a user to be specified, this option is not necessary. Each destination is in the form `user@hostname`, separated by either the '`:`' or the '`;`' character. To take effect, the host name provided in the list must exactly match the host name of a URL in a `machine` block.

Job Scheduling

By default, the project `main` is run with make scheduling. When a job is run with make scheduling, the earliest modification date of the output of the job is compared with the latest modification date of the input to the job. If the newest input file to a job is newer than the oldest output to a job, then the job is not up to date and is run as part of the project. If the newest input is older than the oldest output to the job, the job is up to date and is skipped. Jobs listed in the project's `test` block are always run when the project is run, regardless of if they are up to date or not.

When a job is specified by the `--job` command line parameter, the file modification times are ignored and the job is always run. When a project is specified by the `--job` parameter, the jobs listed in the `test` block are always run; jobs listed in the `build` block are run according to make scheduling. When the `--rebuild` command line option is specified, the file modification times are ignored and all jobs listed in the project are run.

If the output of a job within a project is input to a job in the same project, the job is delayed until after the input file is generated by the other job. If the job producing input to another job fails, the jobs depending on the input are not run.

If a project with a `build` block is included into a project's `test` block, the jobs in the included project's `build` block are scheduled as if they are in a `test` block. If a project with a `test` block is included in another project's `build` block, the included project's `test` block remains scheduled regardless of file modification times.

SSH Integration

TinyBuilder has no built in security; it does no encryption and no authentication. It would be easy to sniff traffic to derive all the source and binaries involved with the build. It would be easy to intercept communications between the client and the service to record or alter the communication. By design, TinyBuilder permits files to be transferred to the service and can execute anything the user running the service can execute; anything that can establish a connection to TCP port 5017 would be able to utilize that service without authentication. To solve all these problems, TinyBuilder provides integration with `ssh` port forwarding.

The Client

The `machine` block will specify when a service must be connected through `ssh`. If `ssh` is properly configured, no other knowledge is needed by the client to complete the connection. However, if a user is required on the `ssh` command line, one cannot be specified in the script. For testing, the command line option `--server-user-list` may be used to specify the user using the syntax `user@hostname`, which will be used as the destination on the `ssh` command line. The `hostname` specified in the command line option must exactly match the host name specified in the `machine` block. Multiple destinations may be specified in the command line option by separating them with either a '`;`' or a '`:`'. Once the command line is working, the same destination list can be assigned to the environment variable `TB_SSH_SERVER_LIST`. If a list is specified on the command line, the environment variable will not be used.

If `ssh-agent` is used, no password will be necessary. If `ssh-agent` is not used for the destination, the password will need to be entered during the build. If the `--automated` command line parameter is specified or if the client is used without a terminal, any connection requiring a password will fail. If the client is used interactively, the password may be entered. On Windows, each `ssh` command launched by the client will have its own command prompt where the password may be entered. The title of the command prompt will specify the server. On the other platforms, a window will pop up in the UI to accept the password. The client can only connect to one server at a time; it will wait for up to two minutes for the password to be entered before the connection fails. Since all other connections will wait for the password, entering passwords can make the build take longer.

The Interactive User Interface

If the stdout, stderr, or stdin of the client are redirected or otherwise not connected to the terminal, then client will execute in batch mode. When running in batch mode, The client will print messages as each job completes, but stdin will be ignored. Since stdin is ignored, ssh port forwarding will fail if a password is required.

When `tbuild` is attached to a terminal, it will run in interactive mode. While in interactive mode, a simple UI will be displayed as the jobs run. The UI displays the commands as they execute

along with the time they have been running. If backspace is pressed, the UI will exit and any jobs in progress will abort. After the UI exits, the same messages printed in batch mode will be printed to the terminal.

The left most column in the UI is the URL of the connection. The next column is the job name. After the job name is the seconds between the current time and when the job started; this time will include time the job was delayed due to concurrency restrictions. The third column is the command that is being run, without its arguments. If an input or output transfer is in progress, the number of K transferred will be displayed.

The last column, the command time, is the time since the command or transfer was started. The client may not check if a command has completed for some time, so the command may have completed on the machine while the command time is still increasing on the client.

Batch Use

If stdin, stdout or stderr are redirected, the interactive user interface will be disabled. Instead, the client will only print a line when a job completes; stderr is not used. It is impossible for a password to be entered without a terminal, so ssh integration can only work with `ssh-agent`. In a similar way, if the `--automated` command line parameter is specified, the terminal, if any, will be ignored. In addition, when the `--automated` command line parameter is passed, the client will automatically abort if a job does not complete within a half hour. This behavior is intended to prevent a build from running forever when something goes wrong during an automated build.

Error Handling

When an error occurs, the error will be reported to the `build_log.xml` file. See The Build Log chapter of the user guide for additional details.

If the TinyBuilder client encounters an internal error that indicates it has a bug, `tbuild` will panic and create a `panic.bin` file in the current directory. If the `tbuild` process is hung, pressing `ctrl-c` or sending a `SIGINT` will cause `tbuild` to panic and produce a `panic.bin` file. The `panic.bin` file can be sent to `support@tinymanagement.com` for diagnosis. The `panic.bin` file will contain the same contents of the `build_log.xml` file, such as IP addresses and file names. It will not contain file contents.

The Agent

In Windows and macOS, there is a distinction between foreground processes and background processes; foreground processes may interact with the GUI, while background processes cannot. In Windows, any executable that makes use of USER32.DLL can be expected to hang if it is run as a background process. In macOS, a process that attempts to interact with the desktop will encounter strange errors. The TinyBuilder service always runs as a background process and all commands it spawns also run as a background process.

To support automation of foreground processes, TinyBuilder provides the agent. When the Windows or macOS TinyBuilder service is installed, the installation also installs an agent. The agent runs whenever the installing user logs in; when the user logs out, the agent is terminated. Since it is running as a foreground process as the installing user, the agent will launch foreground processes. Since Linux does not normally provide a strong distinction between foreground and background processes, there is no Linux agent.

When the agent starts, it allocates an ephemeral port for its use and stores the port number in a file. When the service is asked to connect to the agent, the service reads the file and connects to the specified port over the loopback interface. The service then acts as a tunnel to the agent.

To specify a connection to an agent, a `path` block must be used within a `machine` block. The first member of the path must be a URL to the service on the server running the agent; both the `tb:` and `tbs:` schemes are valid. The second member of the path must be the URL `tbi://localhost`. Doing so instructs the client to first connect to the service normally, then establish a tunnel to the second URL. Since the URL explicitly specifies the agent, the tunnel connects to the agent. For example:

```
machine interactive builder
    path
        tbs://build-server
        tbi://localhost
```

A connection to the agent is treated by the client as a separate connection from the connection to the service. So if one or more jobs are making use of the service and one or more jobs are making use of the agent in the same project, two connections will be established. If the connection to the service is using `ssh` and `ssh` requires a password, the password will need to be entered

twice.

Since the agent terminates when the installing user logs out, the user must remain logged in for TinyBuilder to make use of it. While an automated reboot will restart the service, it will not restart the agent. The user will need to log in after any reboot for the agent to stay running. Any macOS key stores used by the agent will need to be manually unlocked by the user after logging in; the agent does not have the user's password. Password changes are irrelevant to the agent itself, but may be important if the password change disconnects the logged in user from any remote resources used by processes launched by the agent.

Maintenance Story

This chapter describes a fictional situation to demonstrate a possible evolution of a build system. It is intended to show various TinyBuilder script styles with functional examples.

Maintenance Story

Bob was asked to port a `make` script to TinyBuilder. To minimize the disruption he did this a program at a time by replicating the command lines generated by `make`. He prepared a dedicated server to build the product, called `builder`. After the TinyBuilder script was completed and working, Bob was required for another project and Alice was asked to improve the script to be easier to maintain.

Alice inherited a single script to build two programs and single library. The script looked like this:

Naming conventions are very important. A build script can be expected to contain a large number of blocks and it is important to use key terms in a specified order to ensure a common understanding of the meaning of each block.

```
job build debug utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    commands break on error
        mkdir
            -p
            obj
        mkdir
            -p
            libs/debug
        gcc
            -c
            -O0
            -m64
            -g
            -Iinclude
            -o
            obj/file.utils.o
            utils/file.utils.c
        gcc
            -c
```

```
            -O0
            -m64
            -g
            -Iinclude
            -o
            obj/screen.utils.o
            utils/screen.utils.c
        ar
            q
            libs/debug/libutils.a
            obj/file.utils.o
            obj/screen.utils.o
    output
        libs/debug/libutils.a
    machine
        builder

job build production utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    commands break on error
        mkdir
            -p
            obj
        mkdir
            -p
            libs/production
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -o
            obj/file.utils.o
            utils/file.utils.c
        gcc
            -c
            -O3
            -m64
```

Maintenance Story

Block names can and should be very long, descriptive, and use spaces for readability. Typically, build scripts are read much more frequently than they are written, so it is useful to put in the typing up front.

Maintenance Story

```
            -Iinclude
            -o
            obj/screen.utils.o
            utils/screen.utils.c
        ar
            q
            libs/production/libutils.a
            obj/file.utils.o
            obj/screen.utils.o
    output
        libs/production/libutils.a
    machine
        builder

job build debug program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
        program1/src/module2.c
        program1/include/module2.h
        libs/debug/libutils.a
    commands break on error
        mkdir
            -p
            obj
        mkdir
            -p
            bin/debug
        gcc
            -c
            -O0
            -m64
            -g
            -Iinclude
            -Iprogram1/include
            -o
            obj/main.o
            program1/src/main.c
        gcc
```

There is no limitation to the kinds of commands that can be run. While impossible as input or output, absolute paths are permissible within commands. The work area is not a sandbox; it is possible to affect the server and other jobs with a carefully constructed command. Protocol port spaces are shared between jobs.

```
            -c
            -O0
            -m64
            -g
            -Iinclude
            -Iprogram1/include
            -o
            obj/module1.o
            program1/src/module1.c
        gcc
            -c
            -O0
            -m64
            -g
            -Iinclude
            -Iprogram1/include
            -o
            obj/module2.o
            program1/src/module2.c
        gcc
            -g
            -o
            bin/debug/program1
            obj/main.o
            obj/module1.o
            obj/module2.o
            -Llibs/debug
            -lutils
    output
        bin/debug/program1
    machine
        builder

job build production program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
        program1/src/module2.c
        program1/include/module2.h
```

If a directory is not specified as part of the input, it will not exist in the work area at the start of the job. Any other directories need to be created by a script or command as done here.

Maintenance Story

```
        libs/production/libutils.a
    commands break on error
        mkdir
            -p
            obj
        mkdir
            -p
            bin/production
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -Iprogram1/include
            -o
            obj/main.o
            program1/src/main.c
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -Iprogram1/include
            -o
            obj/module1.o
            program1/src/module1.c
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -Iprogram1/include
            -o
            obj/module2.o
            program1/src/module2.c
        gcc
            -o
            bin/production/program1
            obj/main.o
            obj/module1.o
            obj/module2.o
            -Llibs/production
```

```
            -lutils
    output
        bin/production/program1
    machine
        builder

job build debug program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/debug/libutils.a
    commands break on error
        mkdir
            -p
            obj
        mkdir
            -p
            bin/debug
        gcc
            -c
            -O0
            -m64
            -g
            -Iinclude
            -Iprogram2/include
            -o
            obj/main.o
            program2/src/main.c
        gcc
            -c
            -O0
            -m64
            -g
            -Iinclude
            -Iprogram2/include
            -o
            obj/module3.o
```


It is very easy with this script to compile different source files differently. It is possible to accomplish the same thing more elegantly using multiple file lists for source files.

Maintenance Story

```
            program2/src/module3.c
        gcc
            -c
            -O0
            -m64
            -g
            -Iinclude
            -Iprogram2/include
            -o
            obj/module4.o
            program2/src/module4.c
        gcc
            -g
            -o
            bin/debug/program2
            obj/main.o
            obj/module3.o
            obj/module4.o
            -Llibs/debug
            -lutils
    output
        bin/debug/program2
    machine
        builder

job build production program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/production/libutils.a
    commands break on error
        mkdir
            -p
            obj
        mkdir
            -p
            bin/production
```

The script lacks comments, which is obviously not best practice. However, comments are the least interesting part of TinyBuilder's scripting language, so they are not shown here.

```
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -Iprogram2/include
            -o
            obj/main.o
            program2/src/main.c
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -Iprogram2/include
            -o
            obj/module3.o
            program2/src/module3.c
        gcc
            -c
            -O3
            -m64
            -Iinclude
            -Iprogram2/include
            -o
            obj/module4.o
            program2/src/module4.c
        gcc
            -o
            bin/production/program2
            obj/main.o
            obj/module3.o
            obj/module4.o
            -Llibs/production
            -lutils
    output
        bin/production/program2
    machine
        builder

project main
    builds
```

```
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
```

Maintenance Story

The first thing Alice noticed about the script is the number of lines describing commands; lines that could be consolidated by adding steps for compiling the programs and libraries. There were four basic operations in the script, creating the output directories, compiling the C files, building the archive and linking the programs. Each could be consolidated as steps. The updated script was as follows:

```
step create output directory
    parameters
        directory
    commands break on error
        mkdir
            -p
            <<enumerate>directory>

step compile C files
    parameters
        compiler options
        source files
        output directory
    commands break on error
        gcc
            -c
            <compiler options>
            -o
            <output directory><<enumerate, base name, file
name> source files>.o
            <<enumerate along> source files>

step archive library
    parameters
        source files
        output
    commands break on error
        ar
            q
            <output>
            obj/<<base name, file name>source files>.o

step link program
    parameters
        link options
        source files
```

The enumerate expansion option causes mkdir to be run once per assignment. If omitted, the directories would all be created by a single run.

The -c and -o command line options are hard coded here because the command line always needs them. They could be specified by assignments.

The line continued here since there was insufficient space on the page. This cannot be done in an actual script

The link and archive steps are their own steps because they are used differently and require different parameters. Steps can have multiple commands.

```
        executable
        library paths
        libraries
    commands break on error
        gcc
            <link options>
            -o
            <executable>
            obj/<<base name, file name>source files>.o
            -L<library paths>
            -l<libraries>

job build debug utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    paths
        output directory = obj
        output directory = libs/debug
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g
        compiler options = -Iinclude
    paths
        source files           = utils/file.utils.c
        source files           = utils/screen.utils.c
        intermediate directory = obj
        archive name           = libs/debug/libutils.a
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
        intermediate directory
    include step archive library
        source files
        archive name
    output
        libs/debug/libutils.a
```

```
    machine
        builder

job build production utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    paths
        output directory = obj
        output directory = libs/production
    values
        compiler options = -O3
        compiler options = -m64
        compiler options = -Iinclude
    paths
        source files           = utils/file.utils.c
        source files           = utils/screen.utils.c
        intermediate directory = obj
        archive name           = libs/production/libutils.a
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
        intermediate directory
    include step archive library
        source files
        archive name
    output
        libs/production/libutils.a
    machine
        builder

job build debug program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
```


Now, the part of the job specifying commands has become quite repetitive. Repetitive job structures are useful; a new job is easier to copy/paste; especially when naming conventions are done well; a new job should be exactly like the old job; only the block names are changed.

```
        program1/src/module2.c
        program1/include/module2.h
        libs/debug/libutils.a
    paths
        output directory = obj
        output directory = bin/debug
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g
        compiler options = -Iinclude
        compiler options = -Iprogram1/include
    paths
        source files           = program1/src/main.c
        source files           = program1/src/module1.c
        source files           = program1/src/module2.c
        intermediate directory = obj
        executable             = bin/debug/program1
        library paths          = libs/debug
    values
        link options = -g
        libraries    = utils
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
        intermediate directory
    include step link program
        link options
        source files
        executable
        library paths
        libraries
    output
        bin/debug/program1
    machine
        builder

job build production program 1
    input
      include/file.utils.h
```

```
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
        program1/src/module2.c
        program1/include/module2.h
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    values
        compiler options = -O3
        compiler options = -m64
        compiler options = -Iinclude
        compiler options = -Iprogram1/include
    paths
        source files           = program1/src/main.c
        source files           = program1/src/module1.c
        source files           = program1/src/module2.c
        intermediate directory = obj
        executable             = bin/production/program1
        library paths          = libs/production
    values
        libraries = utils
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
        intermediate directory
    include step link program
        link options
        source files
        executable
        library paths
        libraries
    output
        bin/production/program1
    machine
        builder

job build debug program 2
```

```
input
    include/file.utils.h
    include/screen.utils.h
    program2/src/main.c
    program2/src/module3.c
    program2/include/module3.h
    program2/src/module4.c
    program2/include/module4.h
    libs/debug/libutils.a
paths
    output directory = obj
    output directory = bin/debug
values
    compiler options = -O0
    compiler options = -m64
    compiler options = -g
    compiler options = -Iinclude
    compiler options = -Iprogram2/include
paths
    source files           = program2/src/main.c
    source files           = program2/src/module3.c
    source files           = program2/src/module4.c
    intermediate directory = obj
    executable             = bin/debug/program2
    library paths          = libs/debug
values
    link options = -g
    libraries    = utils
include step create output directory
    output directory
include step compile C files
    compiler options
    source files
    intermediate directory
include step link program
    link options
    source files
    executable
    library paths
    libraries
output
    bin/debug/program2
```

```
    machine
        builder

job build production program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    values
        compiler options = -O3
        compiler options = -m64
        compiler options = -Iinclude
        compiler options = -Iprogram2/include
    paths
        source files           = program2/src/main.c
        source files           = program2/src/module3.c
        source files           = program2/src/module4.c
        intermediate directory = obj
        executable             = bin/production/program2
        library paths          = libs/production
    values
        libraries = utils
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
        intermediate directory
    include step link program
        link options
        source files
        executable
        library paths
        libraries
```

Thanks to the library built by another job, this job will always run after `build production utilities library` *and only if the the job succeeds. If run by itself,* `build production program 2` *will always run as long as the library file exists.*

```
    output
        bin/production/program2
    machine
        builder

project main
    builds
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
```

While progress was made, more needed to be done. Every library build and every program build had exactly the same sequence of steps. Therefore, it was possible to consolidate the sequences into two steps, one for a library and one for a program. The resulting script looked like this:

```
step create output directory
    parameters
        directory
    commands break on error
        mkdir
            -p
            <<enumerate>directory>

step compile C files
    parameters
        compiler options
        source files
    commands break on error
        gcc
            -c
            <compiler options>
            -o
            obj/<<enumerate, base name, file name> source
files>.o
            <<enumerate along> source files>

step archive library
    parameters
        source files
        output
    commands break on error
        ar
            q
            <output>
            obj/<<base name, file name>source files>.o

step link program
    parameters
        link options
        source files
        executable
        library paths
        libraries
```

Maintenance Story

Data assignments cannot change after the script is parsed, so passing parameters by reference or value has no meaning. Instead, parameters are just aliases into data assignments used by the job including the step; even when a step includes another step; the aliasing is as deep as necessary.

Maintenance Story

```
    commands break on error
        gcc
            <link options>
            -o
            <executable>
            obj/<<base name, file name>source files>.o
            -L<library paths>
            -l<libraries>

step build library
    parameters
        output directory
        compiler options
        source files
        archive name
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step archive library
        source files
        archive name

step build program
    parameters
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step link program
        link options
        source files
        executable
```

When consolidating steps like this, the parameter list of the step is the union of the parameter lists of the included steps.

```
        library paths
        libraries

job build debug utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    paths
        output directory = obj
        output directory = libs/debug
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g
        compiler options = -Iinclude
    paths
        source files = utils/file.utils.c
        source files = utils/screen.utils.c
        archive name = libs/debug/libutils.a
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        libs/debug/libutils.a
    machine
        builder

job build production utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    paths
        output directory = obj
        output directory = libs/production
    values
        compiler options = -O3
```

```
        compiler options = -m64
        compiler options = -Iinclude
    paths
        source files = utils/file.utils.c
        source files = utils/screen.utils.c
        archive name = libs/production/libutils.a
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        libs/production/libutils.a
    machine
        builder

job build debug program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
        program1/src/module2.c
        program1/include/module2.h
        libs/debug/libutils.a
    paths
        output directory = obj
        output directory = bin/debug
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g
        compiler options = -Iinclude
        compiler options = -Iprogram1/include
    paths
        source files  = program1/src/main.c
        source files  = program1/src/module1.c
        source files  = program1/src/module2.c
        executable    = bin/debug/program1
        library paths = libs/debug
    values
```

```
        link options = -g
        libraries    = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/debug/program1
    machine
        builder

job build production program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
        program1/src/module2.c
        program1/include/module2.h
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    values
        compiler options = -O3
        compiler options = -m64
        compiler options = -Iinclude
        compiler options = -Iprogram1/include
    paths
        source files  = program1/src/main.c
        source files  = program1/src/module1.c
        source files  = program1/src/module2.c
        executable    = bin/production/program1
        library paths = libs/production
    values
        libraries = utils
    include step build program
```

```
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/production/program1
    machine
        builder

job build debug program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/debug/libutils.a
    paths
        output directory = obj
        output directory = bin/debug
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g
        compiler options = -Iinclude
        compiler options = -Iprogram2/include
    paths
        source files  = program2/src/main.c
        source files  = program2/src/module3.c
        source files  = program2/src/module4.c
        executable    = bin/debug/program2
        library paths = libs/debug
    values
        link options = -g
        libraries    = utils
    include step build program
        output directory
```

```
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/debug/program2
    machine
        builder

job build production program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    values
        compiler options = -O3
        compiler options = -m64
        compiler options = -Iinclude
        compiler options = -Iprogram2/include
    paths
        source files  = program2/src/main.c
        source files  = program2/src/module3.c
        source files  = program2/src/module4.c
        executable    = bin/production/program2
        library paths = libs/production
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
```

```
        executable
        library paths
        libraries
    output
        bin/production/program2
    machine
        builder

project main
    builds
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
```


Alice confirmed that this script produced exactly the same command line sequence as Bob's script.

Now that the command line sequences were consolidated, Alice noticed that a lot of lines of the script were consumed with command line options. She decided that the next step to take was to consolidate those options into data blocks that could be shared among the jobs. She confirmed the new script resulted in the same command line sequence as Bob's script:

```
step create output directory
    parameters
        directory
    commands break on error
        mkdir
            -p
            <<enumerate>directory>

step compile C files
    parameters
        compiler options
        source files
    commands break on error
        gcc
            -c
            <compiler options>
            -o
            obj/<<enumerate, base name, file name> source
files>.o
            <<enumerate along> source files>

step archive library
    parameters
        source files
        output
    commands break on error
        ar
            q
            <output>
            obj/<<base name, file name>source files>.o

step link program
    parameters
        link options
        source files
        executable
        library paths
```

Maintenance Story

When compiling, the command is run once per source file due to the enumerate option. The same source file is used in two arguments, which necessitates the enumerate along option. The base name and file name expansion options allow the object files to be placed in a different directory from the source file and removes the original source file extension.

The base name and file name expansion options work together to place a file in a different directory with a new extension cleanly. The enumerate expansion option is not desired here.

Maintenance Story

```
        libraries
    commands break on error
        gcc
            <link options>
            -o
            <executable>
            obj/<<base name, file name>source files>.o
            -L<library paths>
            -l<libraries>

step build library
    parameters
        output directory
        compiler options
        source files
        archive name
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step archive library
        source files
        archive name

step build program
    parameters
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step link program
        link options
        source files
```

```
        executable
        library paths
        libraries

data debug compile options
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g

data production compile options
    values
        compiler options = -O3
        compiler options = -m64

data utils library compile options
    values
        compiler options = -Iinclude

data debug link options
    values
        link options = -g

data production link options

job build debug utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    paths
        output directory = obj
        output directory = libs/debug
    include data
        debug compile options
        utils library compile options
    paths
        source files = utils/file.utils.c
        source files = utils/screen.utils.c
        archive name = libs/debug/libutils.a
    include step build library
```


An empty block is perfectly valid; it is just a block with no assignments. With this block, production jobs can include it just like debug jobs include theirs. If assignments are added, they will affect the production jobs without needing to change them.

```
        output directory
        compiler options
        source files
        archive name
    output
        libs/debug/libutils.a
    machine
        builder

job build production utilities library
    input
        utils/file.utils.c
        include/file.utils.h
        utils/screen.utils.c
        include/screen.utils.h
    paths
        output directory = obj
        output directory = libs/production
    include data
        production compile options
        utils library compile options
    paths
        source files = utils/file.utils.c
        source files = utils/screen.utils.c
        archive name = libs/production/libutils.a
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        libs/production/libutils.a
    machine
        builder

job build debug program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
```

```
        program1/src/module2.c
        program1/include/module2.h
        libs/debug/libutils.a
    paths
        output directory = obj
        output directory = bin/debug
    include data
        debug compile options
        utils library compile options
        debug link options
    values
        compiler options = -Iprogram1/include
    paths
        source files  = program1/src/main.c
        source files  = program1/src/module1.c
        source files  = program1/src/module2.c
        executable    = bin/debug/program1
        library paths = libs/debug
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/debug/program1
    machine
        builder

job build production program 1
    input
        include/file.utils.h
        include/screen.utils.h
        program1/src/main.c
        program1/src/module1.c
        program1/include/module1.h
        program1/src/module2.c
        program1/include/module2.h
```

```
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    include data
        production compile options
        utils library compile options
        production link options
    values
        compiler options = -Iprogram1/include
    paths
        source files  = program1/src/main.c
        source files  = program1/src/module1.c
        source files  = program1/src/module2.c
        executable    = bin/production/program1
        library paths = libs/production
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/production/program1
    machine
        builder

job build debug program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/debug/libutils.a
    paths
```

```
        output directory = obj
        output directory = bin/debug
    include data
        debug compile options
        utils library compile options
        debug link options
    values
        compiler options = -Iprogram2/include
    paths
        source files  = program2/src/main.c
        source files  = program2/src/module3.c
        source files  = program2/src/module4.c
        executable    = bin/debug/program2
        library paths = libs/debug
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/debug/program2
    machine
        builder

job build production program 2
    input
        include/file.utils.h
        include/screen.utils.h
        program2/src/main.c
        program2/src/module3.c
        program2/include/module3.h
        program2/src/module4.c
        program2/include/module4.h
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
```

```
    include data
        production compile options
        utils library compile options
        production link options
    values
        compiler options = -Iprogram2/include
    paths
        source files  = program2/src/main.c
        source files  = program2/src/module3.c
        source files  = program2/src/module4.c
        executable    = bin/production/program2
        library paths = libs/production
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/production/program2
    machine
        builder

project main
    builds
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
```

Now that the command line options settings were consolidated, Alice decided to simplify including the utils library headers and the job `input` blocks by adding file lists. She confirmed that this script produced the same command line sequence as Bob's script:


```
step create output directory
    parameters
        directory
    commands break on error
        mkdir
            -p
            <<enumerate>directory>

step compile C files
    parameters
        compiler options
        source files
    commands break on error
        gcc
            -c
            <compiler options>
            -o
            obj/<<enumerate, base name, file name> source
files>.o
            <<enumerate along> source files>

step archive library
    parameters
        source files
        output
    commands break on error
        ar
            q
            <output>
            obj/<<base name, file name>source files>.o

step link program
    parameters
        link options
        source files
        executable
        library paths
        libraries
```

```
    commands break on error
        gcc
            <link options>
            -o
            <executable>
            obj/<<base name, file name>source files>.o
            -L<library paths>
            -l<libraries>

step build library
    parameters
        output directory
        compiler options
        source files
        archive name
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step archive library
        source files
        archive name

step build program
    parameters
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step link program
        link options
        source files
        executable
```

```
        library paths
        libraries

data debug compile options
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g

data production compile options
    values
        compiler options = -O3
        compiler options = -m64

data utils library compile options
    values
        compiler options = -Iinclude

data debug link options
    values
        link options = -g

data production link options

file list utils library headers
    files
        include/file.utils.h
        include/screen.utils.h

file list utils library source
    files
        utils/file.utils.c
        utils/screen.utils.c

job build debug utilities library
    include input
        utils library source
        utils library headers
    paths
        output directory = obj
        output directory = libs/debug
    include data
```


The file lists and the data blocks conform to naming conventions as well as the jobs to make the meaning of the block clear and to help ease job creation when copy/pasting.

Note that the paths in the file lists are the same as the paths that were repeated in each job. The path is relative to the directory of the script in both in the job and in the file list.

```
        debug compile options
        utils library compile options
    include data with name source files
        utils library source
    paths
        archive name = libs/debug/libutils.a
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        libs/debug/libutils.a
    machine
        builder

job build production utilities library
    include input
        utils library source
        utils library headers
    paths
        output directory = obj
        output directory = libs/production
    include data
        production compile options
        utils library compile options
    include data with name source files
        utils library source
    paths
        archive name = libs/production/libutils.a
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        libs/production/libutils.a
    machine
        builder

file list program 1 headers
    files
```

```
        program1/include/module1.h
        program1/include/module2.h

file list program 1 source
    files
        program1/src/main.c
        program1/src/module1.c
        program1/src/module2.c

job build debug program 1
    include input
        utils library headers
        program 1 headers
        program 1 source
    input
        libs/debug/libutils.a
    paths
        output directory = obj
        output directory = bin/debug
    include data
        debug compile options
        utils library compile options
        debug link options
    values
        compiler options = -Iprogram1/include
    include data with name source files
        program 1 source
    paths
        executable    = bin/debug/program1
        library paths = libs/debug
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/debug/program1
```

The file lists are not together in this script, and that is acceptable. It is only important that the block occurs before any block refers to it. There is no concept of forward declaration in TinyBuilder.

Maintenance Story

```
    machine
        builder

job build production program 1
    include input
        utils library headers
        program 1 headers
        program 1 source
    input
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    include data
        production compile options
        utils library compile options
        production link options
    values
        compiler options = -Iprogram1/include
    include data with name source files
        program 1 source
    paths
        executable    = bin/production/program1
        library paths = libs/production
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/production/program1
    machine
        builder

file list program 2 headers
    files
        program2/include/module3.h
```

```
        program2/include/module4.h

file list program 2 source
    files
        program2/src/main.c
        program2/src/module3.c
        program2/src/module4.c

job build debug program 2
    include input
        utils library headers
        program 2 headers
        program 2 source
    input
        libs/debug/libutils.a
    paths
        output directory = obj
        output directory = bin/debug
    include data
        debug compile options
        utils library compile options
        debug link options
    values
        compiler options = -Iprogram2/include
    include data with name source files
        program 2 source
    paths
        executable    = bin/debug/program2
        library paths = libs/debug
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/debug/program2
    machine
```

```
        builder

job build production program 2
    include input
        utils library headers
        program 2 headers
        program 2 source
    input
        libs/production/libutils.a
    paths
        output directory = obj
        output directory = bin/production
    include data
        production compile options
        utils library compile options
        production link options
    values
        compiler options = -Iprogram2/include
    include data with name source files
        program 2 source
    paths
        executable    = bin/production/program2
        library paths = libs/production
    values
        libraries = utils
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        bin/production/program2
    machine
        builder

project main
    builds
        build debug utilities library
        build production utilities library
```

```
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
```

Maintenance Story

Now that the patterns in Bob's script were consolidated, Alice realized that the script had the following problems:

Jobs were including blocks that were far from the job blocks, so a lot of skipping around within the script was necessary when updating the script.

Any changes made by different developers would require a merge, even when the changes affected different binaries.

The file paths always included directories and could be shortened.

To address the remaining problems, Alice decided to make use of the `import` block. To make script components easy to find, she decided to place `file list` blocks, `data` blocks and `job` blocks into their own files and place those files in the same directory as the source. The data and steps used throughout the build would be placed in a build directory. Once complete, the build scripts would look like:

The convention is to create a main.tb in the build directory and import the entire build into that file. Doing so simplifies the use of the client, though the client will not enforce this convention. Another convention is to import the jobs.tb scripts in the main.tb. Doing so brings in everything needed for the jobs referred to in the project. A third convention is to name the project that builds the product main, which is the default name. Another root project may be desired to run the automated tests, if running the product tests as part of the product build is not desired. Obviously, it is a best practice to have automated tests; this example script doesn't have any.

build/data.tb:

```
data debug compile options
    values
        compiler options = -O0
        compiler options = -m64
        compiler options = -g

data production compile options
    values
        compiler options = -O3
        compiler options = -m64

data debug link options
    values
        link options = -g

data production link options
```

build/main.tb:

```
import steps.tb
import data.tb
import ../utils/jobs.tb
```

```
import ../program1/src/jobs.tb
import ../program2/src/jobs.tb

project main
    builds
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
```

Maintenance Story

build/steps.tb:

```
step create output directory
    parameters
        directory
    commands break on error
        mkdir
            -p
            <<enumerate>directory>

step compile C files
    parameters
        compiler options
        source files
    commands break on error
        gcc
            -c
            <compiler options>
            -o
            obj/<<enumerate, base name, file name> source
files>.o
            <<enumerate along> source files>

step archive library
    parameters
        source files
        output
    commands break on error
        ar
```

Note how no change has been made to the blocks as they are moved to different files. This is correct since there are no paths specified in these blocks.

```
            q
            <output>
            obj/<<base name, file name>source files>.o

step link program
    parameters
        link options
        source files
        executable
        library paths
        libraries
    commands break on error
        gcc
            <link options>
            -o
            <executable>
            obj/<<base name, file name>source files>.o
            -L<library paths>
            -l<libraries>

step build library
    parameters
        output directory
        compiler options
        source files
        archive name
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step archive library
        source files
        archive name

step build program
    parameters
        output directory
        compiler options
        source files
        link options
        executable
```

```
        library paths
        libraries
    include step create output directory
        output directory
    include step compile C files
        compiler options
        source files
    include step link program
        link options
        source files
        executable
        library paths
        libraries
```

Maintenance Story

include/files.tb:

```
file list utils library headers
    files
        file.utils.h
        screen.utils.h
```

Each files.tb specifies all of the file lists needed to use all the files in the same directory. Note how all paths no longer require directories; a copy from the original script would point to files that don't exist.

program1/include/files.tb:

```
file list program 1 headers
    files
        module1.h
        module2.h
```

program1/src/data.tb:

```
data build debug program 1 settings
    paths
        output directory = obj
        output directory = ../../bin/debug
    include
        debug compile options
        debug link options
    values
        compiler options = -I../../include
```

Since most paths and settings have been removed from the job, it is easy to create another similar job using copy/paste. The new job should be exactly like the old job, but with different block names. A good naming convention will make it easy to change the block names.

```
        compiler options = -I../include
    include with name source files
        program 1 source
    paths
        executable    = ../../bin/debug/program1
        library paths = ../../libs/debug
    values
        libraries = utils

data build production program 1 settings
    paths
        output directory = obj
        output directory = ../../bin/production
    include
        production compile options
        production link options
    values
        compiler options = -I../../include
        compiler options = -I../include
    include with name source files
        program 1 source
    paths
        executable    = ../../bin/production/program1
        library paths = ../../libs/production
    values
        libraries = utils
```

Maintenance Story

program1/src/files.tb:

```
file list program 1 source
    files
        main.c
        module1.c
        module2.c
```

program1/src/jobs.tb:

```
import ../include/files.tb
import files.tb
import data.tb
```

By convention, the jobs.tb imports the files it needs for the jobs in the file. This convention is not enforced.

```
job build debug program 1
    include input
        utils library headers
        program 1 headers
        program 1 source
    input
        ../../libs/debug/libutils.a
    include data
        build debug program 1 settings
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        ../../bin/debug/program1
    machine
        builder

job build production program 1
    include input
        utils library headers
        program 1 headers
        program 1 source
    input
        ../../libs/production/libutils.a
    include data
        build production program 1 settings
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        ../../bin/production/program1
```

```
    machine
        builder
```

program2/include/files.tb:

```
file list program 2 headers
    files
        module3.h
        module4.h
```


program2/src/data.tb:

```
data build debug program 2 settings
    paths
        output directory = obj
        output directory = ../../bin/debug
    include
        debug compile options
        debug link options
    values
        compiler options = -I../../include
        compiler options = -I../include
    include with name source files
        program 2 source
    paths
        executable    = ../../bin/debug/program2
        library paths = ../../libs/debug
    values
        libraries = utils

data build production program 2 settings
    paths
        output directory = obj
        output directory = ../../bin/production
    include
        production compile options
        production link options
    values
        compiler options = -I../../include
        compiler options = -I../include
```

```
    include with name source files
        program 2 source
    paths
        executable    = ../../bin/production/program2
        library paths = ../../libs/production
    values
        libraries = utils
```

program2/src/files.tb:


```
file list program 2 source
    files
        main.c
        module3.c
        module4.c
```

program2/src/jobs.tb:

```
import ../include/files.tb
import files.tb
import data.tb

job build debug program 2
    include input
        utils library headers
        program 2 headers
        program 2 source
    input
        ../../libs/debug/libutils.a
    include data
        build debug program 2 settings
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
```

```
        ../../bin/debug/program2
    machine
        builder

job build production program 2
    include input
        utils library headers
        program 2 headers
        program 2 source
    input
        ../../libs/production/libutils.a
    include data
        build production program 2 settings
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        ../../bin/production/program2
    machine
        builder
```

utils/data.tb:

```
data build debug utilities library settings
    paths
        output directory = obj
        output directory = ../libs/debug
    include
        debug compile options
    values
        compiler options = -I../include
    include with name source files
        utils library source
    paths
        archive name = ../libs/debug/libutils.a
```

```
data build production utilities library settings
    paths
        output directory = obj
        output directory = ../libs/production
    include
        production compile options
    values
        compiler options = -I../include
    include with name source files
        utils library source
    paths
        archive name = ../libs/production/libutils.a
```

utils/files.tb:

```
file list utils library source
    files
        file.utils.c
        screen.utils.c
```

utils/jobs.tb:

```
import ../include/files.tb
import files.tb
import data.tb

job build debug utilities library
    include input
        utils library source
        utils library headers
    include data
        build debug utilities library settings
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        ../libs/debug/libutils.a
    machine
```

```
        builder

job build production utilities library
    include input
        utils library source
        utils library headers
    include data
        build production utilities library settings
    include step build library
        output directory
        compiler options
        source files
        archive name
    output
        ../libs/production/libutils.a
    machine
        builder
```

After the script was complete, a third program, `program3`, was added to the project. The program had two modules in addition to a main module; the utils library was not used by this program. The source would be placed in the `program3` directory with an `include` and a `src` directory.

`build/main.tb:`

The change to main.tb consists of importing another jobs.tb file and adding the two new jobs to the project.

Maintenance Story

```
import steps.tb
import data.tb
import ../utils/jobs.tb
import ../program1/src/jobs.tb
import ../program2/src/jobs.tb
import ../program3/src/jobs.tb

project main
    builds
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
        build debug program 3
        build production program 3
```

`program3/include/files.tb:`

This file and the corresponding file for program2 are very similar. Basically, the string program 2 was replaced with the string program 3 to complete the data.tb file. That is the value of a naming convention.

```
file list program 3 headers
    files
        module5.h
        module6.h
```

`program3/src/data.tb:`

```
data build debug program 3 settings
    paths
        output directory = obj
        output directory = ../../bin/debug
    include
        debug compile options
```

```
        debug link options
    values
        compiler options = -I../include
    include with name source files
        program 3 source
    paths
        executable    = ../../bin/debug/program3

data build production program 3 settings
    paths
        output directory = obj
        output directory = ../../bin/production
    include
        production compile options
        production link options
    values
        compiler options = -I../include
    include with name source files
        program 3 source
    paths
        executable    = ../../bin/production/program3
```

program3/src/files.tb:

```
file list program 3 source
    files
        main.c
        module5.c
        module6.c
```

program3/src/jobs.tb:

```
import ../include/files.tb
import files.tb
import data.tb

job build debug program 3
    include input
        program 3 headers
        program 3 source
```

```
    include data
        build debug program 3 settings
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        ../../bin/debug/program3
    machine
        builder

job build production program 3
    include input
        program 3 headers
        program 3 source
    include data
        build production program 3 settings
    include step build program
        output directory
        compiler options
        source files
        link options
        executable
        library paths
        libraries
    output
        ../../bin/production/program3
    machine
        builder
```

Maintenance Story

Alice's company added more developers and the `builder` server started to slow. Her IT department added a new server with the same tool chain and called it `builder2`. To allow the build to use either one of the two servers at random, she added a `machines` block to `build/main.tb` as follows:

`build/machines.tb:`

```
machine builder
    path
        tb://builder
        tb://builder2
```

`build/main.tb:`

Note that the machine block was imported before other scripts. This was done to prevent a default machine block from being constructed by the first job in the script.

```
import machines.tb
import steps.tb
import data.tb
import ../utils/jobs.tb
import ../program1/src/jobs.tb
import ../program2/src/jobs.tb
import ../program3/src/jobs.tb

project main
    builds
        build debug utilities library
        build production utilities library
        build debug program 1
        build production program 1
        build debug program 2
        build production program 2
        build debug program 3
        build production program 3
```

Index

113