TinyBuilder

Reference Guide

Beta

Table of Contents

Introduction 1

The Parser 2
Tokenization 2
Parser Databases 4
Character Encoding 4
Parsers 5

The Build Log 37
The Document Root 37
Connection Information 38
The Job List 39
The Command List 41
The Command Output 42
The Output List 45

TinyBuilder Service 47
The Work Area 47
Security 48
Abstract Servers 49
Error Handling 50
The Path Cache 51
Windows Development Environments 52
Managing the Service on Linux 52
Managing the Service on macOS 54
Managing the Agent on macOS 54
Managing the Service on Windows 55
Managing the Agent on Windows 56

SSH Integration 57
Specifying a User Name 58
Troubleshooting 59

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 Parser

This chapter describes the details of the functionality of the TinyBuilder script parser. The purpose of this chapter is to disambiguate the user guide and document the handling of various corner cases and obscure features.

The TinyBuilder parser consists of a tokenizer along with a set of small parsers. The tokenizer is responsible for segmenting the script into tokens and text. The token is a fixed string that identifies the parser to use; the text is passed to the parser. The parser is a small piece of code that updates structures with what it finds in the text. The parser can supply the tokenizer with a new set of tokens to look for, which identify another set of parsers. When a block ends, the tokenizer signals the parser of the end of the block.

The parser code never handles a new line; the tokenizer uses new lines as delimiters and does not pass the new line code points to any parser. As a result, tokens and text cannot span lines.

Tokenization

The tokenizer is a component of the TinyBuilder script parser that provides structure for the rest of the parser. While part of the parser, it is largely decoupled from the rest of the parser code. The tokenizer determines the block boundaries and signals the parsers when blocks begin and end. The tokenizer divides the script into tokens and text; tokens are used to identify the relevant parser code, while text is processed by the parsers. The text passed to the parser may or may not be empty and may not span a new line. A sequence of spaces (UTF-8 code point 0x20) within a token is consolidated to a single space; spaces within text is interpreted by the parser code.

A root block is a block with no indention. The root block must begin with one of the `project`, `job`, `step`, `import`, `machine`, `data` or `file  list` tokens followed by text. The root block contains zero or more embedded blocks, and an embedded block may contain zero or more embedded blocks. Each root block is finalized at the next root block or at the end of the script. The root block along with its embedded blocks may be considered a tree which the tokenizer traverses in depth first order.

Any new block at the root must start with a known token; children blocks may or may not be identified by tokens. When a line has greater indention than the previous line, a new child block is started. The rules regarding what children blocks a block may have is specified by the parent block; children parsers may be identified by tokens, or the children of a block may be entirely text. A token identifies the parser responsible for the block, which is initialized when the block has started. Text may or may not follow the token; this is specified by the parser identified by the token.

The Parser

A block is finalized when the line has a lesser or equal indention to the parent block; a line with equal or lesser indention signals the tokenizer traversal to go up the tree one or more levels. If the indention is equal to the indention of the previous line, the tokenizer only goes up one level of the tree and only one block is finalized; it is valid for a block to have no content. While the indention is less than the indention of the previous line, the tokenizer moves further up the tree until it finds a block of equal indention to the line. All blocks are finalized as the tokenizer goes up the tree, including the block with equal indention. The parser uses its finalization to check if it is complete and valid; if validation fails, parsing of the script fails. If no block with equal indention can be found as the tokenizer goes up the tree, parsing of the script fails.

Only spaces (UTF-8 code point 0x20) are accepted for indention.

Tokens and text cannot span lines; the tokenizer treats the new line as a delimiter and never passes a new line as text; since parsers never see new lines, they are more platform independent; the variety of methods to define a new line is strictly handled by the tokenizer.

Section 5.8 of the Unicode standard identifies the following code points as new lines:

Acronym	*Name*	*Code Point*
CR	carriage return	000D
LF	line feed	000A
CRLF	carriage return/line feed	000D, 000A
NEL	next line	0085
VT	vertical tab	000B
FF	form feed	000C
LS	line separator	2028
PS	paragraph separator	2029

All of the above are treated equally by the tokenizer as end of lines. The CRLF sequence is treated as a single end of line.

If the first character after a sequence of zero or more spaces is “`#`”, the line is a comment line. All remaining characters in the line are ignored and not parsed. The indention is irrelevant; a comment cannot start or finalize a block. If the “`#`” is preceded by a character other than a space, the “`#`” will be part of the text of the line; it will never begin a comment unless only spaces occur on the line before “`#`”.

The Parser

Parser Databases

The TinyBuilder script parser makes use of three in-memory databases, the object database, the machine database and the import database. All three use a name as the key within the database; the name is a sequence of UTF-8 code points; no case folding, composition or decomposition is done. Duplicate names are not permitted within any database.

The object database is used to store `project`, `job`, `step`, `data` and `file list` blocks as values with a name as the key. These blocks can then be referenced by other blocks.

The key in the machine database is the name of the `machine` block, which is referenced by the names in the job’s machine block. Since the machine database is separate from the object database, an object may have the same name as a machine without problem.

The key in the import database is the absolute path of the imported script; there is no associated value, only names. The purpose of this database is to identify scripts that have already been imported so they are not imported again.

Character Encoding

The script is UTF-8 encoded. Case folding is never done; all string comparisons are case sensitive. Composition and decomposition are never done; it is assumed the editor used to edit the scripts will handle code points consistently. The UTF-8 encoding scheme specifies that some byte sequences are not valid. The tokenizer checks the bytes of the script as it reads them; if it encounters a byte sequence that is not valid according to the UTF-8 encoding, the script parsing will fail. The tokenizer considers the UTF-8 encoding of the Unicode BOM (0xfeff) to be a valid character, but it is ignored regardless of where it appears in the script.

This chapter refers to character literals using double quotes, such as "`#`". These character literals are always the UTF-8 code points with ASCII equivalents; alternatives, i.e. the Arabic comma, are never considered equivalent to the character literals in this chapter. The character sequence "`<0x20>`" is used to represent a sequence of one or more UTF-8 spaces with the value 20 in hexadecimal; in other words, a sequence of one or more ASCII spaces.

Parsers

The Parser

This section lists the parsers used by the tokenizer. Each parser is documented with an initialization, a finalization, a text handling and a token to parser mapping section. The initialization section describes how the parser is initialized and the interpretation of any text appearing after the parser's token. It is possible for initialization to fail. The finalization section describes any checks that occur when the block handled by the parser is finalized. The text handling section describes how the parser treats text within its block; it does not include text on the same line as the token. The token to parser mapping is a list of tokens the parser accepts along with their corresponding parsers.

Tokens are formatted as follows:

```
This<0x20>is<0x20>a<0x20>token<0x20>
```

Spaces within the token are represented by `<0x20>` to highlight the fact that only the ASCII space is accepted; any other valid UTF-8 spaces, like no break space, zero width space, em space, etc are not accepted as a space. Also, `<0x20>` represents a sequence of one or more ASCII space bytes.

The script directory referred to by the parser documentation is the directory containing the script containing the block.

If any parser fails, the TinyBuilder client will fail with a parsing error. Failure can occur at any time.

`root`

The root parser is responsible for the root blocks in the script.

Initialization

The object, machine and import databases are initialized as empty.

Finalization

No checks are performed during `root` finalization.

Text Handling

Only tokens are handled by the root parser; any text will cause the parser to fail.

Token to Parser Mapping

Token	*Parser*
`data<0x20>`data-block-name	`data_block`
`file<0x20>list<0x20>`file-list-block-name	`file_list`
`import<0x20>`import-script-path	`import_parse`
`job<0x20>`job-block-name	`job_parser`
`machine<0x20>`machine-block-name	`machine_block`
`project<0x20>`project-block-name	`project_parser`
`step<0x20>`step-block-name	`step_parser`

`command_argument`

The `command_argument` parser accepts a command during initialization and command line arguments as its children.

Initialization

The line passed to the `command_argument` parser during its initialization is the executable. While this may include a relative or absolute path, the parser treats the text as a literal; path separators will not be adjusted for the machine. If no path separators (as interpreted on the machine side) are present, the machine will search the path for the executable provided. Otherwise, if a relative path is specified, the path is relative to the job directory in the work area.

The Parser

Path separators within command line arguments are not interpreted by the client and will remain as they are in the script, regardless of the path separators used on the machine. The commands running on the machine will interpret the command line arguments, not the client.

The `command_argument` parser initialization will fail if the expansion options parsing fails.

The `command_argument` parser receives a lookup callback from the parent parser, which may be a job or a step parser. The interpretation of names are different for the different parents.

Finalization

The following checks are made:

If the `directory<0x20>name` and `file<0x20>name` expansion options are used on the same expansion, parsing fails.

If the `environment` expansion option is used with any of the `enumerate`, `required`, `base<0x20>name`, `file<0x20>name`, or `directory<0x20>name` expansion options, or if the expansion has text, the TinyBuilder client is unable to expand the command and parsing fails.

If the expansion has the `enumerate<0x20>along<0x20>`name expansion option, another expansion with the name must exist and have an `enumerate` option that is not an `enumerate<0x20>along<0x20>`name or parsing will fail.

All the expansions with `enumerate` and `enumerate<0x20>within` expansion options must form a single chain of expansions with no cycles, or parsing fails.

Text Handling

Commands and arguments may contain zero or more expansions. The start of an expansion is signaled using "`<`" and the end of the expansion is signaled using "`>`". Either character cannot be in a name. An additional "`<`" at the start of the expansion signals the start of an expansion option list, which are delimited by "`,`" and terminated with "`>`". Expansion options are optional; unrecognized expansion options will cause parsing to fail. Zero or more ascii spaces are permitted between any delimiter, expansion option and name; names will not have leading or trailing spaces. Empty expansion option lists or an expansion without a name will cause parsing to fail.

The Parser

The following are accepted as expansion options:

```
base<0x20>name
base<0x20>names
directory<0x20>name
directory<0x20>names
enumerate
enumerates
enumerate<0x20>along
enumerates<0x20>along
enumerate<0x20>within<0x20>name
enumerates<0x20>within<0x20>name
environment
file<0x20>name
file<0x20>names
required
```

Token to Parser Mapping

The `command_argument` parser does not recognize any tokens.

`data_block`

The data_block parser parses root data blocks.

Initialization

Initialization will fail if the token's text is empty, since the data block must have a name. Initialization will fail if another object in the parser's object database has the same name as the token's text. If the checks succeed, a `data_block` parser is added to the object database with the token's text as the key.

The Parser

Finalization

There are no checks when the `data_block` parser is finalized.

Text Handling

No text is permitted by the `data_block` parser.

Token to Parser Mapping

Token	*Parser*
`include`	`data_include_data`
`includes`	`data_include_data`
`include<0x20>with<0x20>name<0x20>name`	`named_data_include_data`
`includes<0x20>with<0x20>name<0x20>name`	`named_data_include_data`
`path`	`data_path`
`paths`	`data_path`
`value`	`data_value`
`values`	`data_value`

`data_include_data`

The `data_include_data` includes other blocks into a data block.

Initialization

There are no checks during `data_include_data` initialization.

Finalization

No checks are performed during `data_include_data` finalization.

The Parser

Text Handling

Each line is interpreted to be a name in the parser's object database. If the name cannot be found, parsing fails. The type of the found block is checked. If the block is not a data block, parsing fails.

The including data block is searched for each name in the included block. If the name is not found, the name and its values are copied to the including data block. If the name is found, the type is checked. If the including value is a data value and the included value is a path value, or if the including value is a path value and the included value is a data value, parsing will fail. If the types are the same, the values associated with the name are appended to the values already in the including block.

Token to Parser Mapping

There are no tokens within a `data_include_data` block.

`data_path`

The `data_path` parser adds path values to a data block.

Initialization

There are no checks during `data_path` initialization.

Finalization

There are no checks during `data_path` finalization.

The Parser

Text Handling

Each line is interpreted as a name value pair. The parser searches for an “`=`” character; if no “`=`” is found, parsing fails. If the first character is “`=`”, parsing fails. If no character follows the “`=`” on the line, parsing fails. Once an “`=`” is found, the trailing spaces are stripped from the name. The data block is searched for the name. If the name is found and the value is not a path value, parsing fails. If the name is not found and the name contains “`<`” or “`>`”, parsing fails. If the name begins with “TB “, parsing fails; names starting with “TB “ are reserved.

The value is added to the list of values associated with the name. If there is a trailing space after the name, the leading spaces are removed from the value. The value is treated as a path relative to the script directory; both “`/`” and “`\`” are considered to be path separators. If the value is an absolute path, parsing fails. The value is appended to the script directory and then normalized to remove any unneeded “`.`” or “`..`” directories.

Token to Parser Mapping

There are no tokens within a data_path block.

`data_value`

The `data_value` parser adds data values to the data block.

Initialization

There are no checks during `data_value` initialization.

Finalization

The Parser

There are no checks during `data_value` finalization.

Text Handling

Each line is interpreted as a name value pair. The parser searches for an “`=`” character; if no “`=`” is found, parsing fails. If the first character is “`=`”, parsing fails. If no character follows the “`=`” on the line, parsing fails. Once an “`=`” is found, the trailing spaces are stripped from the name. The data block is searched for the name. If the name is found and the value is not a data value, parsing fails. If the name is not found and the name contains “`<`” or “`>`”, parsing fails. If the name begins with “TB “, parsing fails; names starting with “TB “ are reserved.

The value is added to the list of values associated with the name. If there is a trailing space after the name, the leading spaces are removed from the value.

Token to Parser Mapping

There are no tokens within a `data_value` block.

`development_environment`

The `development_environment` parser sets the job's development environment.

Initialization

If the development environment was already set for the job, parsing fails. If no development environment name was provided, parsing fails.

Finalization

No checks are made during `development_environment` finalization.

Text Handling

The `development_environment` parser does not allow any text.

Token to Parser Mapping

The `development_environment` parser does not recognize any tokens.

`file_file_list`

The `file_file_list` is responsible for parsing the paths within the `files` block within a `file list` block.

Initialization

There are no checks during `file_file_list` initialization.

Finalization

There are no checks during `file_file_list` finalization.

Text Handling

The text is interpreted as a file path, relative to the script directory. Both "`/`" and "`\`" are accepted as path separators. If the path is an absolute path, parsing fails; otherwise, the file path is converted into a normalized absolute path and is added to the file list.

Token to Parser Mapping

There are no tokens within a `file_file_list` parser block.

`file_list`

The `file_list` parser parses file list blocks.

Initialization

Initialization will fail if the token's text is empty, since the file list must have a name. Initialization will fail if another object in the parser's object database has the same name as the token's text. After the above checks, a `file_list` parser is added to the object database with the token's text as the key.

The Parser

Finalization

There are no checks during `file_list` finalization.

Text Handling

Only tokens are recognized by the `file_list` parser; all unrecognized text will cause parsing to fail.

Token to Parser Mapping

Token	*Parser*
`file`	`file_file_list`
`files`	`file_file_list`
`include`	`include_file_list`
`includes`	`include_file_list`

`import_parse`

The `import_parse` parser is responsible for parsing import blocks.

Initialization

Parsing will fail if the `import<0x20>` token is not followed by any text. The path is normalized based on the script directory. On non-Windows platforms, the absolute path is stored in the import database. On Windows, the file is opened and `GetFileInformationByHandleEx` is called to find identifiers of the file. These identifiers, and not the path, are used as the key into the import database.

Finalization

If the script has already been imported, nothing else is done during finalization. The script path on non-Windows, or the serialized file identifiers on Windows, are used as the key as the import is added to the import database. No valid value is stored in the import database. A new tokenizer is created to parse the imported script. If the parsing of the imported script fails, finalization fails.

Text Handling

No text is permitted within an `import_parse` block.

Token to Parser Mapping

No tokens are permitted within an `import_parse` block.

`include_file_list`

The `include_file_list` is responsible for importing file lists into a file list.

Initialization

There are no checks during `include_file_list` initialization.

Finalization

There are no checks during `include_file_list` finalization.

Text Handling

Each line is interpreted to be a file list block name. If the name is not in the object database, parsing fails. If the name points to a non-file list, parsing fails. Otherwise, each path in the included file list is copied to the including file list. The files in the included file list are already absolute paths; no update of the paths is needed.

Token to Parser Mapping

There are no tokens within a `include_file_list` block.

`job_parser`

The `job_parser` parser is responsible for parsing job blocks.

Initialization

Initialization will fail if the token's text is empty, since the job block must have a name. Initialization will fail if another object in the parser's object database has the same name as the token's text. If the checks succeed, a `job_parser` is added to the object database with the token's text as the key.

Finalization

If no machine has been assigned to the job, parsing fails. If the job concurrency has not been set, it is set to medium. The script directory is stored as the job's home directory.

Text Handling

No text is permitted within a `job_parser`.

Token to Parser Mapping

When the `job_parser` delegates a child block to the `step_command` parser, the lookup callback records the mapping from the parameter position to the parameter name. If the expansion names a data or path value that does not exist when the command is expanded, parsing will not fail. See the Command Generation chapter in the user guide for more details.

Token	*Parser*
`command<0x20>break<0x20>on<0x20>error`	`step_command`
`commands<0x20>break<0x20>on<0x20>error`	`step_command`
`command<0x20>complete<0x20>with<0x20>error`	`step_command`
`commands<0x20>complete<0x20>with<0x20>error`	`step_command`
`command<0x20>ignore<0x20>error`	`step_command`
`commands<0x20>ignore<0x20>error`	`step_command`
`concurrency<0x20>high`	`job_parser_concurrency`
`concurrency<0x20>low`	`job_parser_concurrency`

Token	*Parser*
concurrency<0x20>maximum	job_parser_concurrency
concurrency<0x20>medium	job_parser_concurrency
concurrency<0x20>minimum	job_parser_concurrency
developement<0x20>environment<0x20>name	development_environment
environment<0x20>prefix	data_value
environment<0x20>replace	data_value
environment<0x20>suffix	data_value
failed<0x20>output	file_file_list
include<0x20>data	data_include_data
includes<0x20>data	data_include_data
include<0x20>data<0x20>with<0x20>name<0x20>name	named_data_include_data
includes<0x20>data<0x20>with<0x20>name<0x20>name	named_data_include_data
include<0x20>failed<0x20>output	include_file_list
includes<0x20>failed<0x20>output	include_file_list
include<0x20>input	include_file_list
includes<0x20>input	include_file_list
include<0x20>output	include_file_list
includes<0x20>output	include_file_list
include<0x20>step<0x20>name	step_include_parser
includes<0x20>step<0x209>name	step_include_parser
include<0x20>steps<0x20>name	step_include_parser
includes<0x20>steps<0x20>name	step_include_parser
input	file_file_list
inputs	file_file_list
machine	job_parser_machine
machines	job_parser_machine
output	file_file_list
outputs	file_file_list
path	data_path
paths	data_path
value	data_value
values	data_value

`job_parser_concurrency`

The `job_parser_concurrency` parser sets a job's concurrency.

Initialization

If one of `high`, `low`, `maximum`, `medium` or `minimum` are not following the `concurrency<0x20>` token, parsing fails. If the concurrency of the job was already set, parsing fails.

Finalization

No checks are done during `job_parser_concurrency` finalization.

Text Handling

The `job_parser_concurrency` parser does not permit any text.

Token to Parser Mapping

The `job_parser_concurrency` parser does not recognize any tokens.

`job_parser_machine`

The `job_parser_machine` parser assigns a machine to the job.

Initialization

No checks are performed during `job_parser_machine` initialization.

Finalization

No checks are performed during `job_parser_machine` finalization.

Text Handling

Each line is interpreted as a machine name. The machine database is searched for a matching machine block name. If none is found, a machine block using the TCP protocol to the machine named by the text is added to the machine database. Note that the `machine_block` parser will fail during initialization if an attempt is made to add a machine with the same name later. The machine block is linked to the job.

Token to Parser Mapping

The `job_parser_machine` parser does not recognize any tokens.

`machine_block`

The `machine_block` parser adds machine blocks to the parser's machine database.

Initialization

If there is no name associated with the machine block, parsing fails. The parser's machine database is searched for the name. If the name is found, parsing fails.

Finalization

The machine name is added to the parser's machine database.

Text Handling

The `machine_block` parser does not permit any text.

Token to Parser Mapping

Token	*Parser*
`path`	`path`
`path<0x20>list`	`path_list`

`named_data_include_data`

The `named_data_include_data` includes a data or file list block into a data block and assigns all values to a single name.

Initialization

If there is no text after the token, parsing fails. The text after the token specifies the name accepting all the values.

Finalization

No checks are performed during `named_data_include_data` finalization.

Text Handling

Each line is interpreted to be a name in the parser's object database. If the name cannot be found, parsing fails. The type of the found block is checked. If the block is not a data block or a file list block, parsing fails. The including data block is searched for the name assigned to all included values. If the name is not found, it is added to the data block. If the included block is a data block, the names in the data block are enumerated. If the including name is a data value and the included name is a path value, or if the including name is a path value and the included name is a data value, parsing will fail. All the values are copied to the including name. If the included block is a file list block, the including name is checked to ensure it is a path value. If it is not, parsing will fail. Otherwise, all paths are copied to the name.

Token to Parser Mapping

There are no tokens within a `named_data_include_data` block.

`path`

The `path` parser specifies a list of hops to reach the machine.

Initialization

If there is any additional text after the token, parsing fails.

Finalization

No checks are performed during `path` finalization.

Text Handling

The `path` parser does not permit any text.

Token to Parser Mapping

Token	*Parser*
`tb://`	`tb_url`
`tbi://`	`tbi_url`
`tbs://`	`tbs_url`

`path_list`

The `path_list` parser specifies a list of single hop servers.

Initialization

If there is any additional text after the token, parsing fails.

Finalization

No checks are performed during `path_list` finalization.

The Parser

Text Handling

The `path_list` parser does not permit any text.

Token to Parser Mapping

Token	*Parser*
`tb://`	`tb_url`
`tbi://`	`tbi_url`
`tbs://`	`tbs_url`

`project_build_parser`

The `project_build_parser` adds projects and jobs to a project for make scheduling. Make scheduling may allow the job to be skipped, depending on the modification times of the input and output. See The Client chapter in the user guide for more details.

Initialization

If any text follows the token, parsing fails.

Finalization

The `project_build_parser` has no checks during finalization.

Text Handling

The object database is searched for the text. If the text is not found, parsing fails. If the matching object is not a job or a project, parsing fails. If the project is including itself, parsing fails.

Token to Parser Mapping

The `project_build_parser` does not recognize any tokens.

`project_parser`

The `project_parser` parses project blocks.

Initialization

If there is no name after the token, parsing fails. The parser’s object database is searched for the name. If a match is found, parsing fails.

Finalization

No checks are performed during `project_parser` finalization.

The Parser

Text Handling

The `project_parser` parser does not permit text.

Token to Parser Mapping

Token	*Parser*
`build`	`project_build_parser`
`builds`	`project_build_parser`
`test`	`project_test_parser`
`tests`	`project_test_parser`

`project_test_parser`

The `project_test_parser` adds projects and jobs to a project for test scheduling. When the project is run, all jobs and projects assigned test scheduling are run regardless of modification times. See The Client chapter of the user guide for more details.

Initialization

If any text follows the token, parsing fails.

Finalization

The `project_test_parser` has no checks during finalization.

Text Handling

The object database is searched for the text. If the text is not found, parsing fails. If the matching object is not a job or a project, parsing fails. If the project is including itself, parsing fails.

Token to Parser Mapping

The `project_test_parser` does not recognize any tokens.

step_command

The `step_command` parser parses command blocks.

Initialization

No checks are performed during `step_command` initialization. The `step_command` parser receives a lookup callback from the delegating parser to look up expansion names. Expansion names have a different meaning if the parent parser is a step or a job.

Finalization

If no commands were added to the block, parsing will fail.

Text Handling

Unlike the other parsers, which decide to delegate to child parsers based on tokens, the `step_command` parser delegates the child block to the `command_argument` parser unconditionally. The `command_argument` parser is passed the command as text which may include expansions. If the `command_argument` parser fails, parsing will fail.

Token to Parser Mapping

There is no mapping; the child block is delegated to the `command_argument` parser regardless of the line's content.

`step_include_parser`

The `step_include_parser` is used to include steps.

Initialization

If no name was specified in the text, parsing fails. If the name cannot be found in the parser's object database, parsing fails. If the object corresponding to the name in the object database is not a step, parsing fails. If a step is including itself, parsing fails.

Finalization

The commands are copied from the included step to the including step.

Text Handling

Each line is matched to a parameter in the included step. If too many parameters have been specified, parsing fails. If the parameter in the included step cannot be matched to a parameter in the including step, parsing fails.

Token to Parser Mapping

The `step_include_parser` does not recognize any tokens.

`step_parameter`

The `step_parameter` parser adds parameters to a step.

Initialization

If there is any text after the token, parsing fails. If a parameter block has already been added to the step, parsing fails.

Finalization

The `step_parameter` parser does not perform any checks during finalization.

Text Handling

If the text contains “`<`” or”`>`”, parsing fails. If the text matches another parameter for the step, parsing fails.

Token to Parser Mapping

The `step_parameter` parser does not recognize any tokens.

`step_parser`

The `step_parser` parses step blocks.

Initialization

If there is no text following the token, parsing fails. The object database is searched for the text. If there is a matching object, parsing fails.

Finalization

The `step_parser` does not have any checks during finalization.

Text Handling

The `step_parser` does not permit any text.

Token to Parser Mapping

Token	*Parser*
`parameter`	`step_parameter`
`parameters`	`step_parameter`
`command<0x20>break<0x20>on<0x20>error`	`step_command`
`commands<0x20>break<0x20>on<0x20>error`	`step_command`
`command<0x20>break<0x20>on<0x20>errors`	`step_command`
`commands<0x20>break<0x20>on<0x20>errors`	`step_command`
`command<0x20>complete<0x20>with<0x20>error`	`step_command`
`commands<0x20>complete<0x20>with<0x20>error`	`step_command`
`command<0x20>complete<0x20>with<0x20>errors`	`step_command`
`commands<0x20>complete<0x20>with<0x20>errors`	`step_command`
`command<0x20>ignore<0x20>error`	`step_command`
`commands<0x20>ignore<0x20>error`	`step_command`
`command<0x20>ignore<0x20>errors`	`step_command`
`commands<0x20>ignore<0x20>errors`	`step_command`

Token	*Parser*
`include<0x20>step`	`step_include_parser`
`includes<0x20>step`	`step_include_parser`
`include<0x20>steps`	`step_include_parser`
`includes<0x20>steps`	`step_include_parser`

The Parser

`tb_url`

The `tb_url` parser is responsible for parsing default, insecure hops within a `machine` block's `path` block.

Initialization

There are no checks during `tb_url` initialization.

Finalization

There are no checks during `tb_url` finalization.

Text Handling

Only a host name is permitted; a port is optional and is separated from the host name with the "`:`" character. A final "`/`" is permitted; no additional path is allowed. The parser makes no assumptions about the rules regarding valid host names, except to assume "`:`" and "`/`" are invalid.

Token to Parser Mapping

There are no tokens within a `tb_url` block.

`tbi_url`

The `tbi_url` parser is responsible for parsing a hop to an agent within a `machine` block's path block.

Initialization

There are no checks during `tbi_url` initialization.

Finalization

There are no checks during `tbi_url` finalization.

Text Handling

Only the host name `localhost` is permitted; a port is not permitted. A final "`/`" is permitted; no additional path is allowed.

Token to Parser Mapping

There are no tokens within a `tbi_url` block.

`tbs_url`

The `tbs_url` parser is responsible for parsing secure hops within a `machine` block's `path` block.

Initialization

There are no checks during `tbs_url` initialization.

Finalization

The Parser

There are no checks during `tbs_url` finalization.

Text Handling

Only a host name is permitted; a port is optional and is separated from the host name with the "`:`" character. A final "`/`" is permitted; no additional path is allowed. The parser makes no assumptions about the rules regarding valid host names, except to assume "`:`" and "`/`" are invalid.

The Build Log

When the TinyBuilder client completes all the scheduled jobs, it uses the output from the commands to produce the build log, named `build_log.xml`. The log is a UTF-8 encoded XML document with indention to be both human and machine readable. All elements are in the name space:

`http://www.tinymanagement.com/TinyBuilder/BuildLog/1.0/`

All attributes have no name space. The name space is given the "`tb`" prefix; there is no default name space in the document.

The root of the document is the `BuildLog` element, which contains the version of the client running the build along with time and machine information. The first children of the root element document the machine connections made, including IP addresses and `ssh` command lines. Following the connection information is a list of `job` elements, one per job.

Each `job` element contains machine and timing information. Within each `job` element are elements describing the environment of the job, followed by a list of `command` elements, followed by elements describing the output.

Each `command` element contains a list of elements documenting the parameters, the command output and the exit status of the command. Precise timings are provided as tags within the output and attributes in the exit status element.

The build log is designed to provide at least as much information as would be observable if the commands were performed interactively. Every action of the build is traceable on the machine and the client.

The Document Root

The `BuildLog` element is the root element of the XML document. The build log is a valid XML document; there is only one root. The element contains the following mandatory attributes:

`version`: The version of the client that produced the log.

`StartTime`: The time and day the job was started on the client in the standard XML format.

`BuildHost`: The name of the machine running the client.

`RunningTime`: The number of seconds between the first attempt to connect to a TinyBuilder machine and disconnection from the last machine. Script parsing time and log building time are not included.

The Build Log

Connection Information

The connections to each machine are documented at the start of the XML document. Each connection is recorded as a `machine` element. The order of the `machine` elements are not specified, but none will occur after the first `job` element. The `machine` element contains the following mandatory attributes:

`name`: The name of the corresponding `machine` block in the script.

`PathID`: The index of the path within the `machine` block, starting at zero. If there is no `machine` block, the value will be zero.

The `machine` element has a series of children `hop` elements, one hop element per hop in the `path` block. The `hop` element has the following attributes:

`url`: The URL specified for the hop. This is the same as the hop in the script. This attribute is mandatory.

`to`: The IP address corresponding to the host name in the `url` attribute. The port is included. This attribute is optional; it will not be specified if the host name could not be resolved.

`MajorVersion`, `MinorVersion`, `build`: Specifies the version of the service processing the hop. This attribute will not be specified if a connection could not be established.

`SshCommandLine`: The `ssh` command line used to tunnel the connection. This attribute will not be specified if the host name could not be resolved or the connection is not secured.

A `hop` element may have zero or one `error` elements. An `error` element has the following attributes:

`type`: This may be `connection`, `protocol` or `license`. This attribute is mandatory.

`time`: The time of the failure. This attribute is mandatory.

`code`: An OS specific error code. This attribute is optional.

The `error` element has a text node as a child containing the description of the error.

The Build Log

If the `type` of the error is `connection`, the `code` attribute will be set to a connection error code appropriate for the client's operating system. The text will be from the client. The commands in progress, if any, will still appear in the build log, but there will be no exit status.

If `ssh` cannot connect to the server, the `code` attribute will be set to the exit status of `ssh`. The text will contain the output of `ssh`. If the connection is lost after `ssh` connects, the failure will be reported the same as a connection failure without port forwarding.

If the TCP connection is established, but the initial connection process fails, it is assumed that the server is not a TinyBuilder machine; the `type` will be set to `connection`, not `protocol`.

If the `type` of the error is `protocol`, the text will come from the client; there is no `code` attribute. This error only occurs after a successful connection was made to the machine. It is assumed that a non-TinyBuilder server will not accept the connection sequence, so the error is due to a bug on the client or the machine.

If the `type` of the error is `license`, a premium feature was requested of the machine, but the license on the machine does not permit the feature. There is no `code` attribute; the text will describe the feature requested and the reason for the error.

The Job List

Each job started as part of the build has a `job` element in the build log. Projects do not appear in the build log. The jobs within the build log have no specified order. The `job` element has the following attributes:

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

`machine`: The name of the `machine` block used to run the job. If there is no `machine` block, the name is the host name of the machine used.

`PathID`: The zero based index of the path used within the `machine` block. If there is no `machine` block, the `PathID` is zero.

The Build Log

`status`: The outcome of the job. This may be set to one of `succeeded`, `failed` or `error`. The job has failed if the attribute is set to either `failed` or `error`. The `failed` value indicates that a command returned a non-zero exit status that was not ignored. The `error` value indicates that a failure occurred when running the job.

`RunningTime`: The running time of the job. The start time of the job is when the job was started on the machine; time waiting for the machine to have capacity for the job is not included in the running time. The end time is the time the job failed or immediately after the output archive was written to the client.

`concurrency`: This is the value of the job's concurrency. This is the same as the concurrency value specified in the script.

`DelayTime`: If the job was delayed due to machine capacity, this attribute is set to the number of seconds the job was delayed. The attribute is not set if there was no delay.

`DevelopmentEnvironment`: If the job has a development environment, this attribute is set to its name. If there was no development environment, this attribute is not set.

`ErrorCode`: If the `status` is set to `error` and the failure has an associated error code, this attribute will be set to that error code. If there was no error code, the attribute will not be set.

`ErrorReason`: If the `status` is set to `error` and the failure has an associated error message, this attribute will be set to that error message.

`ErrorPath`: If the `status` is set to `error` and the failure has an associated file, this attribute will be set to the path to that file. If there was no file associated with the error, the attribute will not be set.

If a failure to create the input archive occurs, the job will fail. The `ErrorReason` `ErrorCode` and `ErrorPath` attributes will be set to document the error. However, the archive code has dependencies, and that failure would also be reported, somewhat like an exception. In that case, additional information will be provided using a series of children elements of the `job` element:

`ClientInputError`: A sequence of these elements reports the state of the input archive code, from the top layer to the bottom layer of the code; similar to an exception. This element will not occur if the entire error can be documented using the `ErrorReason` attribute.

`ClientInputErrorMessage`: This element contains an error message from the client. It documents what the client was attempting when the failure occurred, in addition to the string from the `ErrorReason` attribute

If the job updates the environment to be used by commands in the job, the changes to the environment are logged using the following elements:

`PrefixEnvironment`: The value has been prefixed to an existing environment variable. If the environment variable did not exist, it was added.

`ReplaceEnvironment`: The value has replaced an existing environment variable. If the environment variable did not exist, it was added.

`SuffixEnvironment`: The value has been suffixed to an existing environment variable. If the environment variable did not exist, it was added.

The elements have the following attributes:

`name`: The name of the updated environment variable.

`value`: The value used to update the environment variable.

The Command List

Each command line run has a `command` element as a child of the `job` element. The order of the `command` elements is the same order the commands run within the job. The `command` element has the following attributes:

`directory`: The directory the command ran in. The path is relative to the root directory of the job. The root of the job is the lowest directory in the file system that includes all the files in the input, all the path value assignments in the job and all the files in the output.

ExecutableFromEnvironment: It is possible to specify a command by using a value with the environment expansion option; the value specifies an environment variable on the machine in this case. If a command is started this way, the environment variable named is in the ExecutableFromEnvironment attribute; otherwise the attribute is not set.

executable: The name of the executable run if the executable was not from the environment. If the executable was derived from the machine environment, the executable attribute is not set.

ErrorMessage: If the fork failed on a non-Windows machine, or CreateProcess failed on a Windows server, this attribute will contain an associated message.

ErrorCode: If the fork failed on a non-Windows machine, or CreateProcess failed on a Windows server, this attribute will specify the machine operating system error code.

The command element has zero or more parameter elements as children. The order of the parameter elements is the same order the parameters were passed to the command as command line options. They appear before the output and exit status elements. Each parameter element has the following attributes:

environment: If the name was expanded using the environment expansion option, the machine environment variable used for the command line value is specified as an environment attribute. Otherwise, the parameter element has no environment attribute.

value: If the command line parameter was specified without an environment expansion option, the parameter element has a value attribute; otherwise, there is no value attribute. The contents of the value attribute is the text of the command line parameter.

The Command Output

The output from the command is stored in a series of out and err elements, which record stdout and stderr respectively. To maintain readability, the number of Unicode code points within the elements is limited; if the output line continues to the next element, the element will not have an EOL attribute. Like the XML and the TinyBuilder script, the contents of the command output elements are UTF-8 encoded; there are additional elements to allow non-UTF-8 or XML incompatible code points to be stored.

The `out` and `err` elements are each in the output sequence, but their order relative to each other is not specified. The `elapsed` tags can be used to order then relative to each other. The `out` and `err` elements have the following attributes:

`offset`: The number of output bytes logged before this tag. The stdout and stderr offsets are tracked separately.

The Build Log

`EOL`: The name of the EOL character after the end of the tag. This attribute is not specified if the content doesn't end with an EOL. The valid names are:

Name	*Code Point*
NL	0xA
NLCR	0xA, 0xD
LINE TABULATION	0xB
FORM FEED	0xC
CR	0xD
CRNL	0xD, 0xA
NEXT LINE	0x85
LINE SEPARATOR	0x2028
PARAGRAPH SEPARATOR	0x2029

Output bytes are preserved exactly within the build log, but not all bytes are permitted in the UTF-8 encoded XML document. These bytes are recorded using the `InvalidByte` or `CodePoint` tags within the output.

If an output byte would create a byte sequence that is not valid UTF-8, the byte is stored using an `InvalidByte` element. The `InvalidByte` element has a single attribute, `value`. The `value` attribute contains the value of the byte using two hex digits without a `0x` prefix. If the output includes bytes that are valid UTF-8 code points, but they are not valid to place in the XML, such as the NUL code point, the code point is stored in a `CodePoint` element. The `value` attribute records the hex value of the code point, without a `0x` prefix. The `value` attribute records all the bytes in the code point, not byte by byte.

The TinyBuilder machine receives output from the child process in blocks of bytes. Since the time these blocks arrive may significantly help with debugging, an `elapsed` element is placed at the beginning of each byte sequence received by the service. The `elapsed` element has the following attributes:

`EOF`: If the output received an EOF, the attribute is set to `true`. If no EOF was received, the attribute is not present.

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

`seconds`: The number of seconds between the start of the command and the receipt of the following output.

If the program emits more output than the TinyBuilder service can buffer, the service adds a throttle response to the output, which becomes a `throttle` element. By throttling, the service will stop reading output from the process; no output is lost. It can be assumed that the kernel buffers will fill and the process will block soon after it is throttled; the `elapsed` tags afterward will reflect the time the process was blocked. The `throttle` element has the `ThrottleOnElapsed` attribute, which is the number of seconds between the start of the command and when the output throttling began.

When the command completes on the machine, the exit status is recorded and is added to the log as a child element of the `command` element. The exit status is recorded in one of `return`, `signal`, `StartupFailed`, `ClockError`, `ChildCommError`, `ChildRequestError`, `TerminateFailure` or `StatusCheckFailure`. All the elements have an `elapsed` attribute, which is the number of seconds between the start of the command and the program's termination.

When the process completion is detected normally, the exit status of the completed command is recorded using either a `return` or a `signal` element. The `return` element is used when the program completes normally and the `signal` element is used when a non-Windows process crashed. Both elements have a `value` attribute, which is set to the return value in the case of a `return` element, or the signal value in the case of a `signal` element. The `signal` exit status is treated the same as a non-zero return value; the command will be considered a failed command.

If the TinyBuilder service encounters an error while monitoring a child process, the process is killed and the error is reported as one of the failure exit status elements. The failure statuses are all treated as a non-zero return value by the command; the command will be considered a failed command. These exit status elements are:

`StartupFailed`: A failure occurred while starting the child process. On non-Windows platforms, this is the exit status used when the `fork` succeeded but the `exec` failed, which most likely means the executable cannot be found. The element has an `ErrorCode` attribute to provide the machine operating system error code.

`ClockError`: The service could not get the current time when handling a response from the child process. The `value` attribute is set to the machine operating system error code.

`ChildCommError`: The service encountered an error reading output from the child process. The `value` attribute is set to the machine operating system error code.

`ChildRequestError`: This error means the service was unable to request output from the child process. The `value` attribute is set to the machine operating system error code. The `message` attribute is set to a description of the error.

`TerminateFailure`: The service was unable to terminate the child process. The `value` attribute is set to the machine operating system error code. The `message` attribute is set to a description of the error.

`StatusCheckFailure`: The service was unable to determine the exit status of the child process. The `value` attribute is set to the machine operating system error code. The `message` attribute is set to a description of the error.

When the connection failed, the job will fail, regardless of the command's error handling, and there will be no exit status element.

The Output List

The last children elements of a successful job are a list of `output` elements. The `output` element contains the path of each directory and file requested to be in the output archive. This element has no attributes. The paths in the `output` element are relative to the root directory of the job. Each directory needed to store the output will be included in the list.

If an error occurred retrieving the output from a job, the job will fail. After an output error occurs, an attempt will be made to retrieve files from the failed output list, if any. A failure to retrieve any file from the failed output list will be ignored.

When an error occurred while the service is building the output archive, an `OutputError` element is stored in the XML to document the error. The element has an `error` attribute that describes the error. If the output error is due to a missing file, the `error` attribute is set to "missing file" and the `OutputError` element has an additional attribute, `path`, which is set to the path of the missing output file, relative to the job's root directory.

The Build Log

If the service is able to construct the output archive, but an error occurred while transferring the archive to the client, an `OutputDownloadError` element is added to the build log. The `error` attribute is set to a message describing the error.

When any error occurred transferring the output archive to the client, the output list is reset to allow the output archive to be created from the failed output list. If the reset failed, an `OutputResetError` element is added to the build log. An error message may be found in the `error` attribute of this element.

If the client could not store any of the contents of the output archive, the `ErrorCode`, `ErrorReason` and `ErrorPath` attributes in the `job` element will be set to the error and the job will fail. However, the archive code has dependencies, and that failure would also be reported, somewhat like an exception in top to bottom order. In that case, additional information will be provided using a series of `ClientOutputError` elements as children of the `job` element. The text of the `ClientOutputError` element contains additional information regarding the error. The element has no attributes.

TinyBuilder Service

The TinyBuilder service is divided into two processes. The process that connects to clients and processes their requests is `tbuilder`. The process that interfaces with the operating system to manage `tbuilder` is `tbuilderd`. The operating system starts `tbuilderd`, which constructs a command line based on operating system settings and launches `tbuilder`. The `tbuilder` process uses interprocess communication to send messages to `tbuilderd`, which are forwarded to the operating system logs. If `tbuilderd` detects that `tbuilder` terminated unexpectedly, it will automatically restart the process. When the operating system tells `tbuilderd` to shutdown, `tbuilderd` abruptly kills `tbuilder` and shuts itself down. When `tbuilderd` starts again, it cleans up after the killed `tbuilder`.

This chapter describes how `tbuilder` and `tbuilderd` work together to act as a TinyBuilder service that automatically starts and cleanly shuts down as directed by the operating system. On Linux, there are additional options to run the service on non-systemd environments, e.g. BusyBox. This chapter will describe how this can be done.

This chapter also describes the TinyBuilder agent. The agent runs as a foreground process as the user who installed the service; it only runs while the user is logged into the console of the server. The agent is installed along with the Windows and macOS service; it is not supported on Linux. The process that accepts connections and executes their requests is the same `tbuilder` executable running as the service; in this case, it is running as a foreground process. On macOS, the same executable `tbuilderd` interfaces with `launchd`. On Windows, the process managing `tbuilder` is `tbagent.exe`.

The Work Area

When a job is started, a directory is created to contain that job. The parent directory of all of these directories is the work area. The TinyBuilder service has complete ownership of the work area; any files or directories placed there are subject to deletion. The directory used by the job is called the job directory in this chapter; it is frequently referred to as the job’s work area elsewhere. There is no ambiguity since each job behaves as if it has its own work area, regardless of other jobs that may be running. It is only when viewing the TinyBuilder service that it is apparent that there are multiple, independent directories.

After creating the job directory, `tbuilder` will extract the input archive into the directory. After extraction, the job directory will reflect the client side directory structure. The current directory of all the processes created by the job is the directory corresponding to the directory containing the script of the job. Since the script is generally not used as an input file, it is possible for that directory to be empty. As output files are created, they are placed within the job directory in the same directories corresponding to the directories on the client. All output files will be in the job directory; the client will not request any files from anywhere else.

TinyBuilderService

After the job is complete, the job directory is placed on a queue for cleanup by another thread. The thread wakes every thirty seconds and clears every job directory from the queue. When the operating system asks `tbuilderd` to shutdown, it immediately kills `tbuilder`, so any jobs in the queue will not be cleaned and will be left in the work area directory. When `tbuilderd` is started by the operating system again, it will delete all the job directories from the work area before starting `tbuilder`. This will prevent any job files from being left behind indefinitely.

The agent makes use of its own work area; it is separate from the service work area. After cleaning up its work area, the agent creates the `agent-port` file in the root of the work area. This file contains the port number used by the agent in the ASCII format. The first time a connection to the agent is requested in the service, it reads the file to connect to the agent. The Windows agent also creates an `agent-tid` file in the root of its work area; this file is used by the installation to control the agent.

Security

The TinyBuilder service does not do encryption or authentication. Anything that can connect to the service has full access to its capabilities, which include transferring files to and from the machine and executing anything the user running the service can run. Normally, the service install will install the service so that it listens to TCP port 5017 on all the IP interfaces and will accept connections from them. The secure install will setup the service so that it only accepts connections to TCP port 5017 from the loopback interface, making direct remote connections to `tbuilder` impossible. Another process, such as `sshd`, can be used to tunnel a remote connection to the service over the loopback.

The service will be able to perform any processing that the user running `tbuilder` is able to perform. In macOS and Linux, the service runs as the user that ran the installation. In Windows, the service runs as the SYSTEM user. It is possible to setup the “TinyBuilder Job Server” to run as another user using the Windows Services Manager; though a user password will be required to do this.

The agent running on Windows and macOS will run as a foreground process as the user that ran the installation. The agent only runs while the user is logged in. It listens to an ephemeral port on the loopback interface; remote connections are impossible. Since an ephemeral port is used, it would be challenging to tunnel to the port, except through the TinyBuilder service as intended.

TinyBuilderService

Abstract Servers

Each TinyBuilder service installation is considered to be a set of abstract servers for calculating concurrency. A single abstract server is permitted to run a single job with minimum concurrency. The installation sets the number of abstract servers to the number of cores.

The way abstract servers are implemented in `tbuilder` is by assigning the server a number of concurrency slots. The number of concurrency slots that may be used by `tbuilder` is 256 multiplied by the number passed as the `--server-count` command line parameter. The command line parameter is passed a floating point number, so the number of concurrency slots may be specified more precisely.

When the client attempts to start a job, it passes the concurrency slots required by the job. If the number of available concurrency slots are less than the number of slots requested, the job start attempt fails. A connection flag is set so that the client will get the number of available concurrency slots with each poll. Once the poll result indicates the server has enough concurrency slots, the client attempts to start the job again.

The concurrency to concurrency slot mapping is as follows:

Job Concurrency	*Slots Used*
minimum	256
low	128
medium	25

high	10
maximum	1

The agent uses the same `tbuilder` as the service, so abstract servers work the same way. The agent and service do not coordinate their utilization, so the total utilization of the CPU, memory and disk will be higher if the service and agent are used simultaneously.

Error Handling

TinyBuilderService

Errors that occur within `tbuilder` that cannot or should not be reported to clients are reported by using IPC to transfer messages from `tbuilder` to `tbuilderd`. If messages cannot be sent to `tbuilderd` without blocking, the messages will be queued in memory by `tbuilder`. If the memory queue fills, the `tbuilder` process will panic. As part of panic processing, it will attempt to dump all queued error messages into the `panic.txt` file in the current directory. After the dump attempt completes or fails, the process will exit. The exit status will help explain why a `panic.txt` file could not be written if the attempt failed.

Before starting `tbuilder`, `tbuilderd` will detect the presence of a `panic.txt` and copy its contents to the operating system log. If `tbuilder` cannot start, the error is reported by the exit status. The `tbuilderd` process will translate the exit status into an appropriate operating system log message. If `tbuilder` terminates five times within two minutes, `tbuilderd` will stop attempting to restart `tbuilder` and will terminate.

The possible tbuilder exit codes are:

0: The `tbuilder` process only exits when there is an error; there is no path to exit successfully.

1: The error queue filled or an error occurred writing sending messages to `tbuilderd`, and a panic.txt file was created.

2: An error occurred while dumping messages to the panic.txt file.

3: The process could not listen for connections on an interface.

4: Windows Only: The CRT experienced a run time error and there was insufficient context to handle the error.

5: A bad command line option was passed

6: Cannot setup the error pipe.

7: Cannot setup error handling.

8: Cannot use the network interfaces.

9: Windows Only: Cannot initialize WinSock.

10 - 12: Reserved.

13: Windows Only: A CRT error occurred.

14 - 15: Reserved.

TinyBuilderService

On Linux, `tbuilderd` will send messages to the syslog using the `com.tinymanagement.tbuilder` identity. The messages are transferred to `tbuilderd` when `tbuilder` writes to stderr. The stdin and stdout are not used by either process.

On macOS, `tbuilderd` determines if it is an agent by counting the number of sockets passed to it by `launchd`; launchd will not pass any sockets to the agent. When running as a service, `tbuilderd` will use the `com.tinymanagement.tbuilder` identity for logging. When running as an agent, it will use `com.tinymanagement.tbagent` identity. The messages are transferred to `tbuilderd` when `tbuilder` writes to stderr. The stdin and stdout are not used by either process.

In Windows, `tbuilderd.exe` and `tbagent.exe` will send messages to the event log using the ETW interface; the messages are transferred from `tbuilder.exe` using a named pipe. The messages from `tbuilderd.exe` are sent to the `TinyManagement-TinyBuilder-JobService/Admin` application log. The messages from `tbagent.exe` are sent to the `TinyManagement-TinyBuilder-Agent/Admin` application log.

The Path Cache

Whenever `tbuilder` uses the `PATH` environment variable to find an executable, the absolute path to the executable is added to the path cache. The path cache speeds the startup of job processes since the file system no longer needs to be searched after the first invocation of a tool chain executable. The price of the path cache is `tbuilder` may need to be cycled if any change is made to the tool chain; any executables that have been relocated will not be found.

Windows Development Environments

TinyBuilderService

The installation uses the Microsoft utility `vswhere.exe` to find all of the Visual Studio installations. The install adds the development environments to the registry; with the name as the development environment name and the value is the batch to run to setup the development environment. The key used is `DevelopmentEnvironments` key in the key:

```
HKLM\SYSTEM\CurrentControlSet\services\tbuilder\Parameters
```

During its initialization, `tbuilder.exe` executes each command and extracts the environment variables `vcvarsall.bat` has set. The same key is used when `tbuilder.exe` is started by `tbagent.exe`.

The development environments are rebuilt whenever the TinyBuilder service is installed. To make use of a new Visual Studio installation, a re-installation of the service will add the needed values to the registry.

Managing the Service on Linux

At system startup `systemd` will automatically use the configuration file:

```
/etc/systemd/system/tbuilder.service
```

to start `tbuilderd`; the `ExecStart` field specifies the command line. When the system is shutting down, `systemd` will send a `SIGTERM` to `tbuilderd`. The `SIGTERM` handler sends a `SIGKILL` to `tbuilder` and terminates itself. If any jobs have not been cleaned up, `tbuilderd` will clean them up before starting `tbuilder` again.

On systems without `systemd`, `tbuilderd` can be started and stopped using other code. The command line options will define how `tbuilderd` will function; see the `tbuilderd(1)` man page for details. To properly stop `tbuilderd`, send

`SIGTERM` to the process.

Note that `tbuilderd` depends on `glibc` version 2.19, so it cannot run in an environment with an older `glibc` or with no `glibc` available. On those platforms, `tbuilder` must be run in some other fashion.

Running `tbuilder` directly in the foreground is supported on Linux, but the functionality provided by `systemd` and `tbuilderd` will be lost. The benefit of running `tbuilder` directly is that it is a statically linked executable and does not have any dependencies, including `glibc`; so `tbuilder` is capable of running in environments that `tbuilderd` cannot. It is possible for `tbuilder` to run on any machine with a 3.0 Linux kernel or later, including configurations like BusyBox.

TinyBuilderService

When `tbuilder` is run without `tbuilderd`, other code will be needed to replace the functionality provided by `tbuilderd`. This code would need to do the following:

The work area directory must exist before `tbuilder` is started.

If a `panic.txt` is present in the current directory of the `tbuilder` process, its contents should be copied somewhere to help diagnose whatever problem caused the panic. The file should be deleted after it is copied.

Clear the work area directory before starting `tbuilder`. This is needed to ensure that useless files do not occupy file system space indefinitely.

Read `tbuilder`'s stderr; `tbuilder` assumes it can write to stderr asynchronously, so stderr cannot be redirected to a regular file. If the stderr kernel buffer fills, `tbuilder` will queue errors in memory and eventually panic. The messages printed to stderr are the same messages sent to the operating system logs, so it's useful to send the messages somewhere. The messages are NL delimited.

If `tbuilder` crashes, it should be restarted. However, if it crashes too frequently, it should not be restarted. It is likely that a useful error message will be written to stderr when this happens if the execution environment is set up correctly.

The `tbuilder` process has no normal way to shutdown. When `tbuilderd` is requested to shutdown, it kills `tbuilder` using `SIGKILL`.

While `tbuilder` will clean up the work area, the cleanup is asynchronous to the job execution and it is possible for `tbuilder` to terminate without completing its cleanup. Before `tbuilderd` starts `tbuilder`, it will delete all job data from the work area. If `tbuilderd` is not used, job files will gradually leak as `tbuilder` starts and stops unless some other process cleans up the work area.

Managing the Service on macOS

The `tbuilderd` process is started by `launchd` as a launch-on-demand daemon when a connection occurs. `tbuilderd` uses the XPC API to obtain the sockets it will pass to `tbuilder`. The command line parameters passed to `tbuilderd` is specified by the `ProgramArguments` array in the configuration file:

TinyBuilderService

```
/Library/Application Support/com.tinymanagement.tbuilder
```

When the system is shutting down, `launchd` will send a `SIGTERM` to `tbuilderd`. The `SIGTERM` handler of `tbuilderd` sends a `SIGKILL` to `tbuilder` and terminates itself. If any jobs have not been cleaned up, `tbuilderd` will clean them up before starting `tbuilder` again.

Running `tbuilderd` or `tbuilder` outside of `launchd` is not supported on macOS.

Managing the Agent on macOS

The `tbuilderd` process is started by `launchd` as a launch agent when the user logs into the console. `tbuilderd` uses the XPC API to obtain the sockets it will pass to `tbuilder`, but when run as a launch agent, there are no sockets to receive. In this case, `launchd` creates a socket on an ephemeral port, and saves to port to:

```
$HOME/Library/Application Support/com.tinymanagement.tbagent
/workarea/agent-port
```

The command line parameters passed to `tbuilderd` is specified by the `ProgramArguments` array in the configuration file:

```
$HOME/Library/LaunchAgents/com.tinymanagement.tbagent.plist
```

When the system is shutting down, `launchd` will send a `SIGTERM` to `tbuilderd`. The `SIGTERM` handler of `tbuilderd` sends a `SIGKILL` to `tbuilder` and terminates itself. If any jobs have not been cleaned up, `tbuilderd` will clean them up before starting tbuilder again.

Running tbuilderd or tbuilder outside of launchd is not supported on macOS.

Managing the Service on Windows

At system startup, the Windows Service Control Manager will start the "TinyBuilder Job Service" by executing tbuilderd.exe as a service. The command line provided to tbuilder.exe is specified by values in the registry.

TinyBuilderService

When the system is shutting down, tbuilderd.exe is notified by the Windows Service Control Manager to shutdown. The tbuilderd.exe process calls TerminateProcess to kill tbuilder.exe and shuts itself down. If any jobs have not been cleaned up, tbuilderd.exe will clean them up during its startup before starting tbuilder.exe again.

The tbuilderd.exe executable has no command line interface; it is configured using the registry. All registry values used by tbuilderd.exe are in the key:

```
HKLM\SYSTEM\CurrentControlSet\services\tbuilder\Parameters
```

The values are as follows:

workarea: The absolute path to the root of the work area directory. The directory must exist before tbuilderd.exe starts tbuilder.exe.

installation: The directory containing the tbuilder.exe and tbuilderd.exe executables.

servers: A string specifying the number of abstract servers provided by the service. The format is expected to be floating point formatted with two integers separated by ".". For example, "1.5" means that the process may service one job with a minimum concurrency and one job with a low concurrency at the same time. The recommended value is the number of cores on the server as a floating point number.

secure: If set to a DWORD zero, tbuilder.exe will listen to all interfaces for connections. The firewall must be setup to permit connections first; the installation will setup a firewall rule permitting remote connections. If non-zero, `tbuilder.exe` will only listen to the loopback interface, making remote connections impossible. Another program, such as `sshd.exe` may be used to connect to `tbuilder.exe` over the loopback.

Running `tbuilderd.exe` or `tbuilder.exe` outside of the Windows Service Control Manager is not supported.

Managing the Agent on Windows

TinyBuilderService

To ensure the agent starts when the user logs in, the TinyBuilder service install adds the `TinyBuilderAgent` string value to the:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
```

key. The agent setup is placed in the:

```
HKCU\SOFTWARE\TinyManagement\TinyBuilder\Agent
```

key. The values are as follows:

`workarea`: The absolute path to the agent's work area directory. The directory must exist before `tbagent.exe` starts `tbuilder.exe`.

`installation`: The directory where `tbagent.exe` and `tbuilder.exe` are installed.

`servers`: A string specifying the number of abstract servers provided by the agent. The format is expected to be floating point formatted with two integers separated by ".". For example, "1.5" means that the process may service one job with a `minimum` concurrency and one job with a `low` concurrency at the same time. The recommended value is the number of cores on the server as a floating point number.

The agent makes use of the same development environment registry key as the service.

Since the agent runs in the foreground, it may be started and stopped like any Windows application. If an attempt is made to start a second instance of the agent, the second instance will silently exit.

SSH Integration

TinyBuilder has no native authentication and no native encryption. All source code used in a build could be intercepted easily and man in the middle attacks could be successfully launched. If the integrity of the network may be relied upon, all of the preceding are not problems. To secure communication over less safe networks, TinyBuilder is fully integrated with the port forwarding feature of SSH.

Without a machine block, the TinyBuilder client connects to the server over its insecure native protocol. A machine block must be defined to instruct the client to use SSH port forwarding. For example:

```
machine build server
    path
        tbs://build-server
```

The `tbs` scheme in the URL specifies that SSH port forwarding is to be used. Once defined this way, any job using `build server` will use SSH to connect to `build-server`.

If the `machine` block specifies that a server is to be connected using SSH port forwarding, the `ssh` command will be executed in port forwarding mode using the `-L` command line parameter. In Linux and macOS, a name for a UNIX socket is created by the client and passed to `ssh` over the command line. In Windows, an ephemeral TCP port is allocated and used for the communication between `tbuild.exe` and `ssh`.

For each server connection, the client will start an `ssh` process. If the client is not connected to a terminal, then `ssh` will not have a terminal either; so if the client is used without a terminal, such as in an automated build, `ssh` must be able to connect without human interaction. The `ssh-agent` can be useful for this use case. On Linux and macOS, `ssh` shares the terminal with the client and the password may be entered in a pop up window within the terminal. On Windows, the client will open a separate console window for each instance of `ssh`; the password may be entered there if one is needed. The agent requires its own connection; if both the agent and the service are used, two instances of `ssh` will be started.

After starting `ssh`, the client attempts to connect to the socket; `ssh` does not provide one before it has authenticated the connection to the server. If `ssh` requires a password and a terminal is available, the client will wait two minutes for the password to be entered and the connection established. If there is no terminal for `ssh` to use and the client cannot connect within ten seconds, it will give up and the connection will fail. If `ssh` terminates before the client can connect to it, the connection will fail.

SSH Integration

To prevent the main thread from blocking a long time, the client performs DNS lookups and connections on a separate thread; while that thread is busy, no other new connections can be made. If `ssh` requires a password, jobs on connected servers will run, but jobs belonging to servers with no connections will wait until after `ssh` connects; the wait includes the time it takes for the user to enter the password. The `ssh-agent` service can be used to speed up connections.

Specifying a User Name

It is possible that the client may need to use a different user account than the developer normally uses to connect to the server over `ssh`. The way to connect as a different user over `ssh` is to specify `user@server` as the destination on the command line, but specifying the user in the `machine` block will not work.

There are two ways to provide the client with a server/user name mapping, over the command line and through an environment variable. To specify the mapping over the command line, use the option `--server-user-list` followed by a '`:`' or '`;`' separated list of strings in the `user@server` format. When the client is formulating the `ssh` command line for its connection to the server, it will search this list for exact matches to the `server` part of the `user@server` string. If the client finds a match, it will use the `user@server` string as the destination instead of only `server`. To specify a user name with an environment variable, set the `TB_SSH_SERVER_LIST` environment variable to the same '`:`' or '`;`' separated list of strings in the `user@server` format. The list specified on the command line will replace the environment variable setting. The intent of the command line argument is to make it easy to find a user list string that works. The environment variable is intended to be used as part of the developer's setup.

Troubleshooting

Since the TinyBuilder client uses the `ssh` command line, any problems encountered while running `ssh` will also happen to TinyBuilder. If the command line `ssh build-server` works, the TinyBuilder client should be able to use `ssh` without any additional setup within the same shell.

If the command line `ssh user@build-server` is required, then try the command line:

```
tbuild --server-user-list user@build-server main.tb
```

SSH Integration

If that works, then use the platform specific method to add the `TB_SSH_SERVER_LIST` environment variable to the environment and set it to `user@build-server`. If another server is already identified in the environment, use '`:`' or '`;`' to separate the values:

```
user@build-server:user2@build-server2.
```

If `ssh` states the connection is `administratively prohibited`, edit the `sshd_config` file on the server and ensure `AllowTcpForwarding` is set to `yes`. TinyBuilder's SSH integration relies on TCP port forwarding; it will not work without it.

Index

A
- agent-port file 48
- agent-tid file 48

B
- base name expansion option 7, 8
- build attribute 38
- BuildHost attribute 38
- BuildLog element 37
- build_log.xml 37

C
- ChildCommError element 44, 45
- ChildRequestError element 44, 45
- ClientInputError element 41
- ClientInputErrorMessage element 41
- ClientOutputError element 46
- ClockError element 44, 45
- code attribute (error) 39
- CodePoint element 43
- command_argument parser 7, 8, 29
- command element 37, 41, 42, 44
- comments 4
- concurrency attribute 40

D
- data block 2, 4, 6, 9, 10, 11, 12, 19, 23
- data_include_data parser 9, 10, 19
- data_path parser 9, 11, 19
- data value 10, 12, 18, 23
- data_value parser 9, 12, 19
- DelayTime attribute 40
- DevelopmentEnvironment attribute 40
- development environment block 13, 52
- development_environment parser 13, 19
- directory attribute 41
- directory name expansion option 7, 8

E
- elapsed attribute 44
- elapsed element 43, 44
- elapsed tag 43, 44
- enumerate along expansion option 7, 8
- enumerate expansion option 7, 8
- enumerate within expansion option 8
- environment attribute 42
- environment expansion option 7, 8, 42
- EOF attribute 44
- EOL attribute 42, 43
- err element 42, 43
- error attribute 46
- ErrorCode attribute 40, 41, 42, 45, 46
- error element 38, 39
- ErrorMessage attribute 42
- ErrorPath attribute 40, 41, 46
- ErrorReason attribute 40, 41, 46
- executable attribute 42
- ExecutableFromEnvironment attribute 42

F
- file_file_list parser 14, 15, 19
- file list block 2, 4, 14, 15, 17, 23
- file_list parser 6, 15
- file name expansion option 7, 8

H
- hop element 38

I
- import block 2, 6, 16
- import database 4, 16
- import_parse parser 6, 16
- include_file_list parser 15, 17, 19
- InvalidByte element 43

J
- job block 2, 4, 6, 7, 13, 18, 19, 20, 21, 26, 28, 29
- job directory 47, 48
- job element 37, 38, 39, 40, 41, 46
- job_parser_concurrency parser 18, 19, 20
- job_parser_machine parser 19, 21
- job_parser parser 6, 18

M
- machine attribute 40
- machine block 2, 4, 6, 18, 19, 21, 22, 23, 34, 35, 36, 38, 40, 57, 58
- machine_block parser 6, 21, 22
- machine database 4, 21, 22
- machine element 38
- MajorVersion attribute 38
- message attribute 45
- MinorVersion attribute 38

N
- name attribute (job) 40, 41
- name attribute (machine) 38
- named_data_include_data parser 9, 19, 23
- new line 2, 3

O
- object database 4, 9, 10, 15, 17, 18, 23, 26, 27, 28, 30, 32
- offset attribute 43, 44
- out element 42, 43
- OutputDownloadError element 46
- output element 45
- OutputError element 46

OutputResetError element 46
P
panic.txt file 50, 53
parameter element 42
path attribute 46
path block 34, 35, 36, 38
PathID attribute (job) 40
PathID attribute (machine) 38
path_list parser 22, 25
path parser 22, 24
path value 10, 11, 18, 23, 42
PrefixEnvironment element 41
project block 2, 4, 6, 26, 27, 28
project_build_parser parser 26, 27
project_parser parser 6, 27
project_test_parser parser 27, 28
R
ReplaceEnvironment element 41
required expansion option 7, 8
return element 44
root block 2, 6
RunningTime attribute 38, 40
S
seconds attribute 44
signal element 44
SshCommandLine attribute 38
ssh port forwarding 57
StartTime attribute 38
StartupFailed element 44, 45
status attribute 40
StatusCheckFailure element 44, 45
step block 2, 4, 6, 7, 18, 19, 29, 30, 31, 32, 33
step_command parser 18, 29, 32
step_include_parser parser 19, 30, 33
step_parameter parser 31, 32
step_parser parser 6, 32
SuffixEnvironment element 41
T
tbi_url parser 24, 25, 35
tbs_url parser 24, 25, 36
tb_url parser 24, 25, 34
TerminateFailure element 44, 45
throttle element 44
ThrottleOnElapsed attribute 44
time attribute (error) 39
to attribute 38
type attribute 39
U
url attribute 38
V
value attribute 41, 42, 43, 44, 45
version attribute 37
W
work area 7, 47, 48, 53, 54, 55, 56