

AVATAR COMMAND FILE GUIDE

VERSION 4.5

NOVEMBER 2004

DOCUMENT REVISION 1.3

TERMS OF USE / SOFTWARE LICENSE AGREEMENT

Thank you for using Avatar, a software application provided by Vexus Consulting Group Inc. (the "Company"). This page states the terms and conditions (the "Terms" or the "Agreement") under which you may use the software. Please read this page carefully. By use of the software you accept and agree to be bound, without limitation or qualification, by these Terms. If you do not accept any of the Terms stated here, do not use the software. The Company may, in its sole discretion, modify or revise these Terms at any time. The Company may, in its sole discretion, modify, revise, or rewrite the software at any time, without penalty of prosecution or breach of this agreed contact.

Use of Material

The contents of this document, including but not limited to text, software, photographs, graphics, illustrations, artwork, video, music, sound, names, logos, trademarks, service marks and other material ("Material" or "Materials") are protected by copyright and other laws in both Canada and elsewhere. The Material includes both content owned or controlled by the Company, and content owned or controlled by third parties and licensed to the Company.

You may not sell or modify the Material or reproduce, display, publicly perform, distribute, or otherwise use the Material in any way for any public or commercial purpose without the written permission of the Company. Special rules may apply to the use of certain software and other items provided by The Company. Any such special rules are listed as "Legal Notices" are incorporated into this Agreement by reference.

Warranty of Information

THE SOFTWARE AND MATERIALS ARE PROVIDED ON AN "AS IS" BASIS WITHOUT ANY WARRANTIES OF ANY KIND, WHETHER EXPRESS OR IMPLIED. THE COMPANY AND ITS SUPPLIERS, TO THE FULLEST EXTENT PERMITTED BY LAW, DISCLAIM ALL WARRANTIES, INCLUDING BUT NOT LIMITED TO WARRANTIES OF TITLE, FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY AND NON INFRINGEMENT OF PROPRIETARY OR THIRD PARTY RIGHTS. THE COMPANY AND ITS SUPPLIERS MAKE NO WARRANTIES ABOUT THE ACCURACY, RELIABILITY, COMPLETENESS, OR TIMELINESS OF THE MATERIAL, SERVICES, SOFTWARE, TEXT, DOCUMENTS, GRAPHICS, AND LINKS.

THE COMPANY DOES NOT WARRANT THAT THE SOFTWARE WILL OPERATE ERROR-FREE OR THAT THIS SOFTWARE OR ITS SERVER(S) ARE FREE OF POSSIBLY HARMFUL ITEMS. IF YOUR USE OF THE SOFTWARE OR THE MATERIAL RESULTS IN THE NEED FOR SERVICING OR REPLACING EQUIPMENT OR DATA, THE COMPANY IS NOT RESPONSIBLE FOR THOSE COSTS. IF YOUR USE OF THE SOFTWARE RESULTS IN POTENTIAL AND/OR REALIZED LOSS, THE COMPANY IS NOT RESPONSIBLE FOR THESE LOSSES.

Limitation of Liability / Disclaimer of Damages

Your use of the Software is at your own risk. If you are dissatisfied with any of the Materials or other contents of the Software or with these Terms and Conditions, the Company's Privacy Policy, or other policies, your sole remedy is to discontinue use of the Software.

IN NO EVENT SHALL THE COMPANY OR ITS SUPPLIERS BE LIABLE TO ANY USER OR ANY THIRD PARTY FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, EXEMPLARY OR LOST PROFITS) RESULTING FROM THE USE OR INABILITY TO USE THE SOFTWARE OR THE MATERIAL, WHETHER BASED ON WARRANTY, CONTRACT, TORT, OR ANY OTHER LEGAL THEORY, AND WHETHER OR NOT THE COMPANY IS ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Indemnity

You agree to defend, indemnify, and hold harmless the Company, its officers, directors, employees and agents, from and against any claims, actions or demands, including, without limitation, reasonable legal and accounting fees, alleging or resulting from your use of the Material (including Software) or your breach of the terms of this Agreement. The Company shall provide notice to you promptly of any such claim, suit, or proceeding and shall reasonably assist you, at your expense, in defending any such claim, suit or proceeding.

TABLE OF CONTENTS

1 INTRODUCTION.....1

 1.1 Purpose and Scope of this document.....1

 1.2 Document Conventions.....1

 1.3 Documents Related.....1

 1.4 Contacting Us.....2

2 WHAT IS A COMMAND FILE?.....3

 2.1 Elements of the Command File.....3

 2.1.1 Comment Syntax.....3

 2.1.2 Include File Syntax.....3

 2.1.3 Environment Variable Syntax.....4

 2.1.4 Object Syntax.....5

 2.2 Object Types.....5

 2.2.1 Batch Dependency Object.....5

 2.2.2 Batch Start Object.....5

 2.2.3 Command Object.....5

 2.2.4 File Check Object.....5

 2.2.5 File Match Object.....5

 2.2.6 File Transfer Object.....6

 2.2.7 FTP Transfer Object.....6

 2.2.8 HTTP Request Object.....6

 2.2.9 Syslog Object.....6

 2.2.10 Global Object.....6

 2.2.11 Mail Object.....7

 2.2.12 Third Party Avatar Command Objects.....7

3 GENERAL OBJECT FIELDS.....8

 3.1 Branch On Error (optional).....8

 3.2 Branch On Success (optional).....8

 3.3 Contact (optional).....8

 3.4 Critical (optional).....8

 3.5 Description (optional).....9

 3.6 Die Command (optional).....9

 3.7 Die Command Output File (optional).....9

 3.8 Die Time.....9

 3.9 Exit On Completion (optional).....10

 3.10 Holiday (optional).....10

 3.11 Message (optional).....11

 3.12 Minimum Time (optional).....11

 3.13 Monthly Schedule (optional).....12

 3.14 Post Process (optional).....13

 3.15 Post Process Output File (optional).....13

 3.16 Pre Process (optional).....13

 3.17 Pre Process Output File (optional).....14

 3.18 Resume At (optional - dependent on Suspend At).....14

 3.19 Retry Count (optional).....14

 3.20 Retry Time (optional - dependent on Retry Count).....14

 3.21 Site (optional).....14

 3.22 Start Time (optional).....15

 3.23 Suspend At (optional).....16

 3.24 Warn Command (optional).....16

 3.25 Warn Command Output File (optional).....16

 3.26 Warn Time (optional).....16

 3.27 Weekly Schedule (optional).....17

 3.28 Yearly Schedule (optional).....18

4 BATCH DEPENDENCY OBJECT.....19

4.1	Batch Name (required).....	19
4.2	Error File (required).....	19
4.3	Host (optional).....	19
4.4	Threshold (optional).....	19
4.5	Batch Dependency Example.....	19
5	BATCH START OBJECT.....	20
5.1	Batch Name (required).....	20
5.2	Error File (required).....	20
5.3	Host (optional).....	20
5.4	Batch Start Example.....	20
6	COMMAND OBJECT.....	21
6.1	Directory (optional).....	21
6.2	Error File (required).....	21
6.3	Execute (required).....	21
6.4	Exit Codes (optional).....	21
6.5	Group (optional - dependent on the User field).....	22
6.6	Host (optional).....	22
6.7	Output File (required).....	22
6.8	User (optional).....	22
6.9	Command Example.....	22
7	EMAIL OBJECT.....	23
7.1	BCC (optional).....	23
7.2	Body (required).....	23
7.3	CC (optional).....	23
7.4	Error File (required).....	23
7.5	SMTPServer (optional).....	23
7.6	Port (optional).....	23
7.7	Subject (optional).....	23
7.8	To (required).....	23
7.9	Email Example.....	23
8	FILE CHECK OBJECT.....	24
8.1	Error File (required).....	24
8.2	Filename (required).....	24
8.3	Host (optional).....	24
8.4	Minimum Size (optional).....	24
8.5	Maximum Size (optional).....	24
8.6	File Check Example.....	24
9	FILE MATCH OBJECT.....	25
9.1	Directory (required).....	25
9.2	Error File (required).....	25
9.3	Execute (required).....	25
9.4	Filename (required).....	25
9.5	Host (optional).....	25
9.6	Output File (optional).....	25
9.7	File Match Example.....	25
10	FILE TRANSFER OBJECT.....	26
11	FTP TRANSFER OBJECT.....	27
11.1	Action.....	27
11.2	Destination (required).....	27
11.3	Error File (required).....	27
11.4	Password (required).....	27
11.5	Permissions (optional).....	27
11.6	Port (optional).....	27
11.7	Server (required).....	27
11.8	Source (required).....	27
11.9	Timeout (optional).....	27
11.10	User (required).....	27

- 11.11FTP Transfer Example.....28
- 12HTTP REQUEST OBJECT.....29
 - 12.1Error File (required).....29
 - 12.2Output File (required).....29
 - 12.3URL (required).....29
 - 12.4HTTP Request Example.....29
- 13SYSLOG OBJECT.....30
 - 13.1Ident (optional).....30
 - 13.2Message (optional).....30
 - 13.3Priority (optional).....30
 - 13.4Syslog Example.....30
- 14SPECIAL COMMAND FILE SYNTAX.....31
 - 14.1Creating a New File.....31
 - 14.2Appending To The End of An Existing File.....31
 - 14.3Auto-Incrementing a File.....32
 - 14.4Special Command File Variables.....32
 - 14.4.1Pre-Defined Environment Variables.....32
 - 14.4.2Pre-Defined Variables.....33
- 15COMMAND FILE EXAMPLES.....35
 - 15.1Database Backup Example.....39

INDEX OF TABLES

Table 1.1 Document Conventions.....	1
Table 1.2 Related Documents.....	1
Table-1.3 Contact Information.....	2
Table 3.1: Die Time Syntax.....	9
Table 3.2 Holiday Expression Syntax.....	10
Table 3.3 Monthly Range Expressions.....	12
Table 3.4 Resume At Time Formats.....	14
Table 3.5 Start Time Formats.....	15
Table 3.6 Suspend At Formats.....	16
Table 3.7 Weekly Range Expressions.....	17
Table 3.8 Yearly Range Expressions.....	18
Table 6.1 Exit Code Range Expression.....	21
Table 14.1 Pre-Defined Environment Variables.....	32
Table 14.2 Pre-Defined Variables.....	33

INDEX OF EXAMPLES

Example 1: Command File Comment.....	3
Example 2: Include File Syntax.....	3
Example 3: Include File Syntax (2).....	3
Example 4: Environment Variables.....	4
Example 5: Environment Variables Via Shelling Out.....	4
Example 6: Generic Object Syntax.....	5
Example 7: Global Object Syntax.....	6
Example 8: Branch On Error Field.....	8
Example 9: Die Time.....	9
Example 10: Exit On Completion.....	10
Example 11: Global Holiday Definition.....	11
Example 12: Multiple Holiday Definition.....	11
Example 13: Minimum Time.....	11
Example 14: Monthly Command Schedule.....	12
Example 15: Last Business Day of the Month.....	13
Example 16: Post Process	13
Example 17: Pre Process	13
Example 18: Retry Count.....	14
Example 19: Start Time	15
Example 20: Suspend At.....	16
Example 21: Warn Time	16
Example 22: Weekly Command Schedule.....	17
Example 23: Yearly Command Schedule.....	18
Example 24: Batch Dependency	19
Example 25: Batch Start.....	20
Example 26: Exit Code Definition.....	21
Example 27: Simple Command Execution.....	22
Example 28: Remote Command Execution.....	22
Example 29: Load Balanced Command Execution.....	22
Example 30: Sending An Email.....	23
Example 31: File Check	24
Example 32: File Match	25

Example 33: FTP Transfer.....28
Example 34: HTTP Request.....29
Example 35: Syslog Object.....30
Example 36: Clobbering an Exiting Output File.....31
Example 37: Appending to the End of an Existing File.....31
Example 38: Auto-Incrementing Files.....32
Example 39: Pre-Defined Environment Variables.....33
Example 40: Pre-Defined Variables.....33
Example 41: Signaling an Error.....35
Example 42: Load Balancing a Resource Intensive Command.....35
Example 43: Setting a Warn Time (relative).....35
Example 44: Setting a Warn Time (absolute).....36
Example 45: Setting a Die Time.....36
Example 46: Setting a Command Start Time.....36
Example 47: Setting a Minimum Run Time.....37
Example 48: Setting a Retry Count.....37
Example 49: Setting Multiple Holidays.....37
Example 50: Creating a Complex Range.....38
Example 51: Automatically Pausing the Batch.....38
Example 52: Branching Commands.....38
Example 53: System and Database Backup Example.....39

1 Introduction

What is Avatar? Avatar is a sophisticated, yet simple to use, batch processing client/server application. Avatar is designed to bridge the gap between what standard Unix based systems provide for batch scheduling (*cron*) and the requirements of most (if not all) Unix based information technology companies. Avatar provides features for reporting on success or error, creating windows of time for commands, system load balancing, dynamic module loading and a highly designed extensible framework.

1.1 Purpose and Scope of this document

The purpose of this document is to provide a comprehensive reference to the syntax and semantics of the Avatar command file. The command file provides a majority of the functionality, capability and personality of Avatar. This document is a reference for anyone wanting to create a batch through Avatar.

1.2 Document Conventions

In order to keep the documentation as consistent as possible The following table outlines the conventions that are used in this document.

TABLE 1.1 DOCUMENT CONVENTIONS

TYPE	PURPOSE
Bold Courier Font	A command to be typed in.
Courier Font	Output of a command.
Bold dollar sign. (\$)	Bourne shell prompt. (Not to be typed in).
Bold percent sign (%)	C shell prompt (Not to be typed in).
Bold pound sign. (#)	Root shell prompt. (Not to be typed in).
<i>Italic Courier Font</i>	A filename or directory name.
CAPITALIZED BOLD	Environment variable names.

1.3 Documents Related

TABLE 1.2 RELATED DOCUMENTS

DOCUMENT	DESCRIPTION
Avatar Installation Guide	This document provides the information to install the license and install both the client and the server.
Avatar Command File Guide	This document outlines the syntax of the Avatar Command File.
Avatar Administration Interface	This document provides all the information required to use administration interface.
Avatar Client Interface	This document provides all the information required to use client interface.

1.4 Contacting Us

To ask any pre-sales or post-sales questions, please use one of the following methods.

TABLE-1.3 CONTACT INFORMATION

METHOD	ADDRESS
World Wide Web	http://www.vexus.ca
E-mail	sales@vexus.ca

2 What is a Command File?

The command file is the file which contains all of the commands required for an Avatar batch. A command file contains the commands, environment variables, loop definitions and command error checking and response. A command file is akin to a shell script; except command files are easier to define, and can be more powerful than a script. The power of the command file comes from the ease at which a batch designer can construct a highly complex batch with little effort. In order to design Avatar batch jobs, the individual elements of the command file must be understood.

2.1 Elements of the Command File.

There are four distinctive elements in a command file:

- Comments
- Include Files
- Environment Variables
- Command Objects

2.1.1 Comment Syntax

A command file may contain as many comments as needed to describe the details of the command file. The syntax of a comment in a command file was borrowed from shell programming comment syntax. A leading hash sign (#) on a line designates a comment.

EXAMPLE 1: COMMAND FILE COMMENT

```
#  
# This is a comment in a command file.  
#
```

Comments may be put anywhere in a command file.

2.1.2 Include File Syntax

A command file may import or include another command file. This allows a batch designer to put common commands in a generic command file. This allows the designer to build robust batches with minimal amount of redundancy. To include a file use the **include** keyword followed by a filename.

EXAMPLE 2: INCLUDE FILE SYNTAX

```
#  
# Include two files; one named /tmp/generic.commands  
# the other /home/avatar/generic/environment.include  
#  
setenv INCLUDE_DIRECTORY /home/avatar/include  
include $INCLUDE_DIRECTORY/generic.commands  
include /home/avatar/generic/environment.include
```

Note that the filenames are absolute and not relative. Although both are allowed, using a relative path name requires knowing which directory the process will be in at the time the file is included. This can lead to confusion and incorrect results. Please note, that include path names can be specified with an environment variable.

Unlike C or C++, files can be included in any part of a command file so the following syntax is valid.

EXAMPLE 3: INCLUDE FILE SYNTAX (2)

```
#  
# Example showing a file can be included any where in a
```

```
# command file.
#
Command = Test Command 1
    Execute      = find / -name core -exec rm -rf {} \;
    Output File  = /dev/null
    Error File   = /dev/null

include /home/avatar/commandset2

Command = Test Command 2
    Execute      = ls
    Output File  = /dev/null
    Error File   = /dev/null
```

The contents of an include file is the same as a regular command file, including environment variables, and commands. Nested include files are also allowed, so included files can include other files themselves. Also, unlike C or C++, a file included in a command file will only be included once. So if there is a cyclic loop of include files, the command file parser will avoid it. There is no need for equivalent **#define** statements at the top of include files.

Keep in mind, if a command file defines a global command, and an include file defines a global command, the last definition will be used.

2.1.3 Environment Variable Syntax

Another feature of a command file are environment variables. Much like shell programs, command files can define and use environment variables. This allows the batch designer to set environment variables within the batch and not the environment of the user running the batch. This allows for greater control over the batch environment. The syntax for defining environment variables is borrowed from C shell variants.

EXAMPLE 4: ENVIRONMENT VARIABLES

```
#
# Set An environment variable named MYSQL_DB, and set
# the value to GTA
#
setenv MYSQL_DB    GTA

#
# Use the environment variable MYSQL_DB
#
Command = Dump Database
    Execute      = mysqldump -u root -p -A $MYSQL_DB
    Output File  = mysql.dump
    Error File   = mysql.error
```

You can also set environment variables by shelling out¹, as demonstrated by the following example. (note the back-ticks around the command)

EXAMPLE 5: ENVIRONMENT VARIABLES VIA SHELLING OUT

```
#
# Shell an environment variable.
#
setenv DATE `date +%d-%b-%Y %M`
```

¹Unix/Linux systems only.

2.1.4 Object Syntax

There are currently several types of objects available in a command object: *Global Object*, *Command Object*, *File Transfer Object*, *File Check Object*, *FTP Transfer Object*, *HTTP Request Object*, *Mail Object*, *Batch Dependency Object*, and *Batch Start Object*. Each object type performs a specific task, which will be discussed later on. The general syntax for all object types in the command file is as follows:

EXAMPLE 6: GENERIC OBJECT SYNTAX

```
Object Type = Name
  Field      = Value
  Field      = Value
```

When defining an object; the object type needs to be at the far left side of the file. There can not be any leading white space at the definition of an object type. The **Name** of the object is an arbitrary name which is used to identify the command. It is usually a good idea to create unique names for each command in a command file², but it is not necessary. Each **Field** value in an object has to start with a white space character. (either a TAB or a space) Between the field name and the value is an equal sign which also has to be surrounded by white space. There is no limit on how many white space characters which can be used, so it is possible to line up equal signs with TAB characters for aesthetic purposes. The specific fields of each object type are discussed in greater detail later in the document. The order in which the fields are defined in a command object makes no difference to how the object will behave.

2.2 Object Types

This section defines the role of each object type.

2.2.1 Batch Dependency Object

The **Batch Dependency** object allows the batch designer to create dependencies on batches from the local Avatar server or remote. This allows the ability to create cross machine dependencies. The specifics of this object are discussed in greater detail, further on in the document.

2.2.2 Batch Start Object

The **Batch Start** object allows the batch designer to start batch jobs on remote machines running the Avatar server. This allows the ability to create cross machine batch chains. The specifics of this object are discussed in greater detail, further on in the document.

2.2.3 Command Object

The **Command** object is the object which executes commands. The commands the command object executes are in the same format as if you typed the command a shell prompt.

2.2.4 File Check Object

The **File Check** object determines if a file exists on a given server. Since many batch jobs use files for state management; Avatar has the ability to detect if a file exists on a local or remote Avatar server. The specifics of the file check object are discussed in greater detail, further on in the document.

2.2.5 File Match Object

The **File Match** object determines if a files exist on a given server which match the regular expression provided. The object allows for commands to be run on the files found.

² Especially if you are to use the **Branch On Success** or **Branch On Error** directives.

2.2.6 File Transfer Object

The **File Transfer** object is the object which allows for seamless file transfer between Avatar servers. This object allows the batch designer to send files between Avatar servers without having to specify the user's password. The specifics of this object are discussed in greater detail, further on in the document.

2.2.7 FTP Transfer Object

The **FTP Transfer** object is the object which sends and receive files via the FTP protocol. This object allows the batch designer to send files to remote machines, which may not be running Avatar. The specifics of this object are discussed in greater detail, further on in the document.

2.2.8 HTTP Request Object

The **HTTP Request** object will connect to a web page and download the page. (images excluded). This allows cold fusion developers to start cold fusion batch jobs from the Avatar server. The specifics of the HTTP Request object are discussed in greater detail, further on in the document.

2.2.9 Syslog Object

The **Syslog** object allows the batch designer to send message to syslog. Typically batches will send a message to syslog at the start and end of the batch run – this object simplifies this requirement.

2.2.10 Global Object

A **Global** object is a special object which defines values for all of the objects in the command file. The role of the global object is to reduce redundancy between objects within the command file. To define a global object, just name the command *global*³. If a global object is defined, all objects in the command file will inherit the field values defined in the global object. If a global object defines a value and a command object defines the same value, the command definition will override the global definition. The global object does not have to be defined; when it is the global object does not get executed.

EXAMPLE 7: GLOBAL OBJECT SYNTAX

```
#
# Define the environment variables.
#
setenv ROOT_DIR /home/glover/Avatar/examples
setenv LOG_DIR $ROOT_DIR/logs

#
# Define the global command object.
#
Command = global
    Error File = /dev/null

#
# Create a command object.
#
Command = Example Command Object
    Execute      = find / -ls
    Output File  = $ROOT_DIR/ls.out

#
# Create a file transfer object.
#
File Transfer = Example File Transfer Object
    Source      = /tmp/source
    Destination = /tmp/destination
    Output File = /tmp/tx.out
```

³ The **global** command name is not case sensitive.

```
Host          = vcg02
User          = bubba
Permissions   = Preserve
Direction     = Push
```

Please note that both the command object and the file transfer object, did not define the required **Error File** field. This is because the global object defined the value. The command objects do not need to define the field because they inherited the value from the global object.

2.2.11 Mail Object

The **Mail** object is the object which sends e-mail. This object allows the batch designer to send an e-mail via SMTP through the Avatar Server. The specifics of this object are discussed in greater detail, further on in the document.

2.2.12 Third Party Avatar Command Objects

Avatar 4.0 provides functionality that wasn't previously available. Avatar 4.0 allows third party vendors to create loadable modules. Those modules will be documented independently of this document. Any module not included in this document, is to be assumed a third party module and will have it's own documentation.

3 General Object Fields

This section of the document defines all the fields that each object type can use. More specific information about each object type will follow this section. All examples in this section use a command object, but all the fields listed are relevant to all object types.

Each field listed will have either *optional* or *required* beside the name. This signifies whether the field is required or not.

3.1 Branch On Error (optional)

The **Branch On Error** field is used to jump to another command in the command file if the command exited with an error. The following example demonstrates the use of the **Branch on Error** field.

EXAMPLE 8: BRANCH ON ERROR FIELD

```
#
# This example demonstrates the Branch On Error field.
#
Command = Bad Command
  Execute      = XXX
  Error File   = /dev/null
  Output File  = /dev/null
  Branch On Error = Error Command

#
# This command will never get run if the previous command
# exits with an error. It will be 'jumped'
#
#
Command = Find
  Execute      = find /
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error

#
# This command will be called by 'Bad Command'
#
Command = Error Command
  Execute      = mail -s 'Batch Died' < errorMessage
  Output File  = /dev/null
  Error File   = /dev/null
```

3.2 Branch On Success (optional)

The **Branch On Success** field is used to jump to another command in the command file if the command exited successfully. This is akin to the **Branch On Error** field.

3.3 Contact (optional)

The **Contact** field is used in the error file. When a command dies, an error file is created. The **Contact** field is one of many fields which appears in the error file. The **Contact** field is purely an informational field used to enhance the details of the error file.

3.4 Critical (optional)

This field specifies if the command is a critical process or not. If the value of this field is set to *No*, and the command exits with an error, the batch will not halt. If the value is set to *Yes*, then the batch will halt and create an error file. By default all commands are considered critical in nature, so setting this field explicitly to *Yes* does not change the criticality of the command.

3.5 Description (optional)

The **Description** field is a free form description of the command.

3.6 Die Command (optional)

This is the command which is run if the command exits with an error. If the command does not exit with an error, then the command defined in the **Die Command** field will not be executed.

3.7 Die Command Output File (optional)

If this field is defined, and the command in the **Die Command** field is executed, then the output of the **Die Command** will be sent to this file. If this field is not defined, and the command exits with an error, then the output of the **Die Command** will be redirected to `/dev/null`.

3.8 Die Time

The **Die Time** field allows the designer of the batch to set windows around execution times. If a command should only run for a certain length of time, then the **Die Time** can be set. If a command has the **Die Time** field set and the command runs for too long, the Avatar server will kill the command and exit the batch with an error. If the **Die Command** field has been set, then the command defined in the **Die Command** field will be executed. The following table outlines the syntax of the **Die Time** field.

TABLE 3.1: DIE TIME SYNTAX

FORMAT	DESCRIPTION	EXAMPLE VALUE	RESULT
HH:MM:SS	Specifies a time value relative to the current day. This value is in a 24 hour format.	12:00:02	If the command is still running at 12:00:02 then the process will be killed.
+HH:MM:SS	Specifies a time value as a relative offset from the current time.	+00:10:15	If the command is still running after running for 15 seconds, the process will be killed.
HH:MM:SSn	Specifies a time value for the next day. This gives Avatar the ability to span the 12 midnight time line.	09:37:00n	If the command is still running at 09:37:00 the next day, then the process will be killed.

Where **HH** represents the hour, **MM** represents the minutes and **SS** represents the seconds.

EXAMPLE 9: DIE TIME

```
#
# This example demonstrates the Die Time field. This example
# will kill the command if it's been running for more than
# 5 minutes.
#
Command = Command
Execute = longRunningCommand
Error File = /dev/null
Output File = /dev/null
Die Time = +00:05:00
Die Time Output File = /tmp/output
Die Time Command = sendErrorAlert
```

3.9 Exit On Completion (optional)

This field is used to exit the batch. It is only needed if a branch is done via the **Branch On Error** or **Branch On Success** fields were defined.

EXAMPLE 10: EXIT ON COMPLETION

```
#
# This example demonstrates the Exit On Completion field.
#
Command = Bad Command
  Execute           = XXX
  Error File        = /dev/null
  Output File       = /dev/null
  Branch On Error   = Error Command

#
# This command will be called by 'Bad Command'. Note the
# field Exit On Completion states the batch will exit if
# this command is run. This avoids the server from running
# any subsequent commands. In this case the Find command
# defined below.
#
Command = Error Command
  Execute           = mail -s 'Batch Died' < errorMessage
  Output File       = /dev/null
  Error File        = /dev/null
  Exit On Completion = Yes

#
# This command will never get run if the previous command
# exits with an error. It will be 'jumped'
#
Command = Find
  Execute           = find /
  Output File       = /tmp/find.out
  Error File        = /tmp/find.error
```

3.10 Holiday (optional)

The **Holiday** field defines a holiday the command is scheduled to take. This allows some commands in a batch not to run on specified holidays. The format of the **Holiday** field is:

Holiday = HOLIDAY_NAME (expression)

where expression can be one of the following

TABLE 3.2 HOLIDAY EXPRESSION SYNTAX

EXPRESSION	EXAMPLE VALUE	MEANING
YYYY/MM/DD	1999/06/01	Do not run on June 1 st 1999
YYYY/MM/*	1999/06/*	Do not run on any day in June of 1999
YYYY/*/DD	1999/*/01	Do not run on the 1 st of any month during 1999.
MM/DD	06/01	Do not run on June 1 st .
MM/*	06/*	Do not run during the month of June.
DD	01	Do not run on the first of any month.

Where **YYYY** represents the year, **MM** represents the month, and **DD** represents the day.

If you wish to schedule holidays for all of the commands in the batch, define the holiday(s) in the global command object.

EXAMPLE 11: GLOBAL HOLIDAY DEFINITION

```
#
# Example command object which takes Christmas off.
#
Command          = Example Command Object
  Execute        = find / -ls
  Output File    = /tmp/out
  Error File     = /dev/null
  Holiday        = Christmas (12/25)
```

If you wish to have multiple holidays for a command, just add another holiday definition.

EXAMPLE 12: MULTIPLE HOLIDAY DEFINITION

```
#
# Example command object with a holiday of Christmas and
# New Years
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/out
  Error File   = /dev/null
  Holiday      = Christmas (12/25)
  Holiday      = NewYears (01/01)
```

3.11 Message (optional)

This field allows the user to add an extra message in the **Message** field of the running log file. This is a useful field for sending messages to the operators which might assist them in fixing the problem. This message will only be printed out when the command exits with an error.

3.12 Minimum Time (optional)

The **Minimum Time** field specifies the minimum length of time the command is expected run for. This is a useful field for detecting if something went wrong because it did not run as long as expected. The format of this field is HH:MM:SS where HH is the length in hours, MM is the length in minutes, and SS is the length in seconds. Since this field value is absolute, a value of 02:33:12 means the command is expected to run a minimum of 2 hours, 33 minutes and 12 seconds. If it doesn't then it is considered an error.

EXAMPLE 13: MINIMUM TIME

```
#
# This demonstrates the Minimum time field. The command called
# 'longRunningCommand' is called and expected to run at least
# 30 seconds. If it does not run 30 seconds it is considered an
# error.
#
Command = Example Command Object
  Execute      = longRunningCommand
  Output File  = /tmp/out
  Error File   = /dev/null
  Minimum Time = 00:00:30
```

3.13 Monthly Schedule (optional)

This field allows the designer to schedule commands on a monthly basis. This means commands can be scheduled to run on various days within the month. This field accepts a complex expression which defines which days the command is to be executed. The following table outlines the syntax of the **Monthly Schedule** field.

TABLE 3.3 MONTHLY RANGE EXPRESSIONS

EXPRESSION	DESCRIPTION
A	Run on day A of the month.
A,B	Run on days A and B.
A-B	Run on days A to B inclusive.
!A	Do not run on day A.
!A,!B	Do not run on day A or day B.
!A-!B	Do not run on days A to B inclusive.
A/B	Run if today is day A of the week and B is day B of the month.
!A/B	Run if today is not day A of the week and B is day B of the month.
!A!/B	Run if today is not day A of the week and B is not day B of the month.
end	Run on the end of the month.
A/end-X	Run if today is day A of the week and X days from the end of the month.
!A/end-X	Run if today is not day A of the week and X days from the end of the month.
!A!/end-X	Run if today is not day A of the week and not X days from the end of the month.

The syntax A/B in the monthly expression means run the command if today is the day of the week specified by A, and B is the day of the month specified by B. The syntax end-1, evaluates to the second to last day of the month, while end-2 evaluates to the third to last day. The following example demonstrates a simple monthly schedule definition.

EXAMPLE 14: MONTHLY COMMAND SCHEDULE

```
#
# This creates a schedule for the command so it runs on
# the first, fifth and the end of the month.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /dev/null
  Error File   = /dev/null
  Monthly Schedule = 1,5,end
```

The **end** keyword is used to state the end of the month. This is necessary since months have different lengths. The monthly command range has the most complex expression syntax because commands can be told to run on a specific day, if the day occurs on a specific day of the week. The following demonstrates how to run a on the last business day of the month, if the end of the month lands on a weekend.

EXAMPLE 15: LAST BUSINESS DAY OF THE MONTH

```
#
# This schedules the command to run on the last
# Friday of the month if the end of the month
# happens to land on the weekend.
#
Command = Example Command Object
  Execute           = find / -ls
  Output File       = /dev/null
  Error File        = /dev/null
  Monthly Schedule  = 5/end-1,5/end-2
```

This field can be used in conjunction with the **Weekly Schedule** and **Yearly Schedule** fields to create a highly complex schedule.

3.14 Post Process (optional)

The **Post Process** field allows the designer to run a command after the actual object has run successfully. This allows the batch designer to do some cleaning up after the command has run successfully.

EXAMPLE 16: POST PROCESS

```
#
# This dumps the database to a file. Since the dbdump command
# creates a lot of temporary files in /tmp we want to clean
# them up after the command exits successfully.
#
Command = Database dump.
  Execute           = dbdump -d Fuzzy -o fuzzy.dump
  Output File       = fuzzy.out
  Error File        = fuzzy.error
  Post Process      = rm -f /tmp/dbdump.tmp*
```

If the command exited with an error, then the **Post Process** command will not get called. The **Post Process** command will only be called if the command completes successfully.

3.15 Post Process Output File (optional)

If this field is defined, and the command in the **Post Process** field is executed; the output of the **Post Process** command will be sent to this file. If this field is not defined then the output will be redirected to `/dev/null`.

3.16 Pre Process (optional)

The **Pre Process** fields allow the designer to perform a command before the actual command has to be run. This allows the person who is assembling the batch, to do some setting up before a command gets run. The following example demonstrates the use of the **Pre Process** command.

EXAMPLE 17: PRE PROCESS

```
#
# The pre process command just removes the database output
# file before we dump the database. This assures that the
# file created is new.
#
Command = Database dump.
  Execute           = dbdump -d Fuzzy -o fuzzy.dump
  Output File       = fuzzy.out
  Error File        = fuzzy.error
  Pre Process       = rm -f fuzzy.dump
```

3.17 Pre Process Output File (optional)

If this field is defined, and the command in the **Pre Process** field is executed, then the output of the **Pre Process** command will be sent to this file. If this field is not defined then the output will be redirected to `/dev/null`.

3.18 Resume At (optional - dependent on Suspend At)

The **Resume At** field works in conjunction with the **Suspend At** field. It's main responsibility is to restart the command once it has been automatically suspended via the **Suspend At** field definition. The format of the **Resume At** field is defined in the following table.

TABLE 3.4 RESUME AT TIME FORMATS

FORMAT	DESCRIPTION
HH:MM:SS	Tells the command to resume at the given time.
+HH:MM:SS	Resumes the command HH hours, MM minutes, and SS seconds after suspension.
HH:MM:SSn	This resumes the command at the given time the next day. This allows the commands in the command file to span the 12 midnight time line.

3.19 Retry Count (optional)

This field specifies how many times to try a given command before giving up. This allows the command to error and run again and still be considered to have completed successfully. This is useful when time dependent elements of a overnight batch may fail. If the field **Retry Time** has not been set, then the command will be restarted immediately. If the command fails and is restarted more than the **Retry Count** value, the command will have failed and the appropriate action will have been taken.

EXAMPLE 18: RETRY COUNT

```
#
# This example demonstrates the use of the Retry Count
# field. If the command fails the server will retry the
# command the given # of times until it the count is
# exhausted. It will wait 1 minute between tries.
#
Command = Failing Command
Execute   = dbdump -d Fuzzy -o fuzzy.dump
Output File = fuzzy.out
Error File  = fuzzy.error
Retry Count = 10
Retry Time  = 00:01:00
```

3.20 Retry Time (optional - dependent on Retry Count)

This is the time between retries. If the **Retry Count** field has been set, and the command exited with an error, then the server will wait the number of seconds before retrying the command. Since this field value is absolute, a value of 00:33:12 means the server will wait 33 minutes and 12 seconds before rerunning the command. This field is only meaningful if the **Retry Count** field has been defined.

3.21 Site (optional)

The **Site** field is used in the error file. When a command dies, an error file is created. The **Site** field is one of many fields which appears in the error file. The **Site** field is purely an informational field used to enhance the details of the error file.

3.22 Start Time (optional)

The **Start Time** field states which time the command is to run. If the time specified has not passed yet, the Avatar server will sit and for the time to arrive. If the time has already passed; then the command will be run immediately. The **Start Time** field can be an absolute time or a relative time.

TABLE 3.5 START TIME FORMATS

FORMAT	DESCRIPTION
HH:MM:SS	Starts the command at the given time. This time is relative to the current day.
+HH:MM:SS	Starts the command HH hours, MM minutes, and SS seconds from the current time.
HH:MM:SSn	This starts the command at the given time the next day. This allows the commands in the command file to span the 12 midnight time line.

The following example demonstrates how to use the different **Start Time** formats.

EXAMPLE 19: START TIME

```
#
# This command will start at 7pm.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /dev/null
  Error File   = /dev/null
  Start Time   = 19:00:00

#
# This command will start 2 hours after the
# Avatar server reaches this point in the
# batch
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /dev/null
  Error File   = /dev/null
  Start Time   = +02:00:00

#
# This command will start at 2am the next day.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /dev/null
  Error File   = /dev/null
  Start Time   = 02:00:00n
```

3.23 Suspend At (optional)

The **Suspend At** field sets a time in which the command will automatically suspend itself. This allows the designer to suspend long running commands during periods where system resources are limited.

TABLE 3.6 SUSPEND AT FORMATS

FORMAT	DESCRIPTION
HH:MM:SS	Suspend the command at the time given.
+HH:MM:SS	Suspend the command HH hours, MM minutes, and SS seconds from the current time.
HH:MM:SSn	This suspends the command at the given time the next day.

The **Suspend At** field must be used in conjunction with the **Resume At** field.

EXAMPLE 20: SUSPEND AT

```
#
# This suspends the command at 9am until 5pm.
#
Command = Example Command Object
Execute           = longRunningCommand
Output File       = /dev/null
Error File        = /dev/null
Suspend At        = 09:00:00
Resume At         = 17:00:00
```

3.24 Warn Command (optional)

The **Warn Command** field defines what command to run when an object lapses into the defined warning period. Avatar has the ability to set warning signals off if a command runs for too long, or runs past a specified time. This command will only be executed if the **Warn Time** field has been set.

3.25 Warn Command Output File (optional)

If this field is defined, and the command in the **Warn Command** field is executed, then the output of the **Warn Command** will be sent to this file. If this field is not defined then the output will be redirected to `/dev/null`.

3.26 Warn Time (optional)

The **Warn Time** is a time specified in the same format as the **Start Time**. The **Warn Time** field specifies when a warning should be sent off for this command. This is useful if a command seems to be taking too long to run. The warning will only be issued if the command is still running when the warn time has lapsed.

EXAMPLE 21: WARN TIME

```
#
# A warn time of 7pm has been set for this command. If the
# command is still running at 7pm a warning will be sent out.
#
Command = Example Command Object
Execute           = find / -ls
Output File       = /dev/null
Error File        = /dev/null
Warn Time         = 19:00:00
Warn Command      = sendWarningAlert
Warn Command Output File = /tmp/warn
```

3.27 Weekly Schedule (optional)

This field allows the designer to schedule commands on a weekly basis. This means commands can be run Monday to Friday, Monday to Wednesday, Wednesday to Sunday, or any other combination. This field accepts a complex expression which defines which days the command is to be executed. The following table outlines the syntax of the **Weekly Schedule** field.

TABLE 3.7 WEEKLY RANGE EXPRESSIONS

EXPRESSION	DESCRIPTION
A	Run on day A of the week.
A,B	Run on days A and B .
A-B	Run on days A to B inclusive.
!A	Do not run on day A.
!A,!B	Do not run on day A or day B.
!A-!B	Do not run on days A to B inclusive.

EXAMPLE 22: WEEKLY COMMAND SCHEDULE

```
#
# This schedules the command to run Monday to Thursday,
# and Saturday
#
Command = Example Command Object
Execute      = find / -ls
Output File  = /dev/null
Error        = /dev/null
Weekly Schedule = 1-4,6
```

Take note that Monday is 1 and Friday is 5. The days of the week start with Sunday, which is zero, and ends with Saturday, which is 6.

This field can be used in conjunction with the **Monthly Schedule** and **Yearly Schedule** fields to create a highly complex schedule.

3.28 Yearly Schedule (optional)

This field allows the designer to schedule commands on a yearly basis. This field accepts a complex expression which defines which days the command is to be executed. The following table outlines the syntax of the **Yearly Schedule** field.

TABLE 3.8 YEARLY RANGE EXPRESSIONS

EXPRESSION	DESCRIPTION
A	Run on day A of the year.
A,B	Run on days A and B.
A-B	Run on days A to B inclusive.
!A	Do not run on day A.
!A,!B	Do not run on day A or day B.
!A-!B	Do not run on days A to B inclusive.
A/B	Run if today is day A of the month and B is month B of the year.
!A/B	Run if today is not day A of the month and B is month B of the year.
!A!/B	Run if today is not day A of the month and B is not month B of the year.

EXAMPLE 23: YEARLY COMMAND SCHEDULE

```
#
# This schedules the command to run on the 1st day of the
# year, the fifth day of the year and the first of June.
#
Command = Example Command Object
Execute      = find / -ls
Output File  = /dev/null
Error File   = /dev/null
Yearly Schedule = 1,5,1/6
```

This field can be used in conjunction with the **Weekly Schedule** and **Monthly Schedule** fields to create a highly complex schedule.

4 Batch Dependency Object

The **Batch Dependency** object allows the batch designer to create batch level dependencies; so if one batch fails and another is dependent on it's success, the dependent batch will fail.

4.1 Batch Name (required)

The **Batch Name** field is the name of the batch to check the status of.

4.2 Error File (required)

This field defines the name of the error file for the batch dependency. This file is created if the batch start does not succeed.

4.3 Host (optional)

The **Host** field tells the server which server to check for the given batch name. If this is not provided, then the local server is used.

4.4 Threshold (optional)

The **Threshold** field specifies a time threshold in which the status of the dependent batch will be considered stale.

4.5 Batch Dependency Example

The following example demonstrates the complete definition of a **Batch Dependency** object.

EXAMPLE 24: BATCH DEPENDENCY

```
#
# This will check the status of a remote batch named
# Oracle Backup. The threshold field is set to 1 hour, so
# if the status of the batch is older than 1 hour, then
# the status is considered to be invalid and the command
# will exit with an error.
#
Batch Dependency = Check the batch named 'Oracle Backup'
  Host           = vcg01
  Error File     = /dev/null
  Batch Name     = Oracle Backup
  Threshold      = 01:00:00
```

5 Batch Start Object

The **Batch Start** object allows the batch designer to start batch jobs on remote servers from within a command file. Thus allowing the batch designer to create cross machine dependencies. This section of the document describes all the **Batch Start** fields available.

5.1 Batch Name (required)

The **Batch Name** field specifies the name of the batch on the remote server to start. If the remote server has more than one batch job defined with the given name, then the first batch found in the registration list will be started.

5.2 Error File (required)

This field defines the name of the error file for the batch start. This file is created if the batch start does not succeed.

5.3 Host (optional)

The **Host** field tells the local Avatar server which host to contact when starting a remote batch job. If not provided it will use the local server.

5.4 Batch Start Example

The following example demonstrates the complete definition of a **Batch Start** object.

EXAMPLE 25: BATCH START

```
#
# This will start the batch named Sybase Dump on the
# server named vcg02
#
Batch Start = Remote batch start.
  Batch Name      = Sybase Dump
  Error File     = /tmp/error
  Host           = vcg02
```

6 Command Object

This section of the document lists all the fields specific to the command object.

6.1 Directory (optional)

The **Directory** field specifies the current working directory of the command. This field eliminates the need to specify a command with a built in change directory.

6.2 Error File (required)

This field defines the name of the error file for the command. This file is created if the command specified in the **Execute** field exits with an error. The following example demonstrates the use of the **Error File** field.

6.3 Execute (required)

The **Execute** field contains the actual command which needs to be run. The following example runs the *find* command on the */export* filesystem.

6.4 Exit Codes (optional)

The **Exit Codes** values specifies the acceptable non-zero values the command may exit with. Since not every command exits with a non-zero value, this field provides the flexibility to allow any command to exit with any of the values specified in this field. If the command exits with a value which is not specified in this field, then it is considered an error. If this field is not defined, then the only acceptable exit value is zero. Anything else will be taken as an error. The following table outlines the syntax of this field.

TABLE 6.1 EXIT CODE RANGE EXPRESSION

EXPRESSION	DESCRIPTION
A	The exit code represented by A is allowed and nothing else.
A,B	The exit codes A and B are allowed and nothing else.
A-B	If the command exited with anything but A it is considered a successful exit.
!A,!B	If the command exited with anything but A and B it is considered a successful exit.
!A-!B	If the command exited with anything outside the range of A to B inclusively then it is considered a successful exit.

Any of these expressions can be strung together using the comma (,) as a joiner. The comma is considered an AND operator in this case. The following example outlines how to set up a command with allowable exit codes of 1, 2-9, 12 and 21.

EXAMPLE 26: EXIT CODE DEFINITION

```
#
# Create allowable exit codes of 1, 2-9, 12, and 21
#
Command = Example Command Object
Execute      = find / -ls
Output File  = /dev/null
Exit Codes   = 1,2-9,12,21
```

Notice that 0 (zero) is not listed, so if this command exits with zero, it will be considered an error.

6.5 Group (optional - dependent on the User field)

When the **Group** field is specified in conjunction with the **User** field; the command will be run as the given user, using the given group as the primary group.

6.6 Host (optional)

This specifies which host to run the command on. If the **User** field is defined in conjunction with this field then the command will be run on the remote machine as that user. Otherwise it will be run as the user who the batch will be run as. If more than one hostname is provided in this field⁴, then the server will poll each remote server and pick the server with the lowest system load. Notice that in order to load balance, the same user account must exist on all the remote hosts; otherwise the remote command execution will not work.

6.7 Output File (required)

This field of the command object specifies where the output of the command is to go.

6.8 User (optional)

This field allows a batch designer to switch users while a command file is running. This applies to both local and remote calls. If the user running the batch, does not have sufficient privileges to switch to the defined user, the command will fail.

6.9 Command Example

These examples demonstrate several command objects.

EXAMPLE 27: SIMPLE COMMAND EXECUTION

```
#
# This example runs a simple command.
#
Command = Dump the database to disk.
Execute      = dumpdb -d Fuzzy -o /tmp/fuzzy.out
Output File  = /tmp/dump.out
Error File   = /dev/null
```

EXAMPLE 28: REMOTE COMMAND EXECUTION

```
#
# This example runs a remote command.
#
Command = Dump the database to disk.
Execute      = dumpdb -d Fuzzy -o /tmp/fuzzy.out
Output File  = /tmp/dump.out
Error File   = /dev/null
Host         = vcg02
```

EXAMPLE 29: LOAD BALANCED COMMAND EXECUTION

```
#
# This example runs a load balanced command.
#
Command = Dump the database to disk.
Execute      = dumpdb -d Fuzzy -o /tmp/fuzzy.out
Output File  = /tmp/dump.out
Error File   = /dev/null
Host         = vcg02, vcg01, vcg03
```

⁴The list of hosts is a comma separated list of host names. Spaces are allowed.

7Email Object

The **Email** object allows the batch designer to send an e-mail during the course of a batch. This eliminates the need to write e-mail scripts. Since it is an SMTP client, the SMTP daemon does not have to be local; which is an added advantage to running a local e-mail script.

7.1BCC (optional)

The BCC field is a blind carbon copy list. It is a space separated list of e-mail addresses the message is to be sent too. In a blind carbon copy, the recipient does not see the other people on the carbon copy.

7.2 Body(required)

The body specifies the actual text of the e-mail message.

7.3CC (optional)

The CC field is a carbon copy list. It is a space separated list of e-mail addresses the message is to be sent too.

7.4Error File (required)

This field defines the name of the error file for the command. This file is created if the command does not succeed.

7.5SMTPServer (optional)

The host field specifies the hostname of the SMTP server to use to relay the message out. If it is not specified then the local host SMTP server will be used.

7.6Port (optional)

This field specifies the port to use when connecting to the SMTP server. If non is specified then the default port (25) is used.

7.7Subject (optional)

This field specified the subject of the mail message. If it is not defined, then a default value of (*no subject*) will be used.

7.8To (required)

This field specifies the address of the recipient of the e-mail.

7.9Email Example

The following example demonstrates the complete definition of a **Mail** object.

EXAMPLE 30: SENDING AN EMAIL

```
#
# This sends an e-mail to sales@vexus.ca asking for more copies
# of Avatar.
#
Mail = Request more copies.
  To           = sales@vexus.ca
  BCC          = sales@mycompany.com
  Subject      = Avatar Is Great
  Body        = Send 100 more copies of Avatar.
  Error File   = /tmp/error
```

8 File Check Object

The **File Check** object allows the batch designer to check for the existence of files on the local or remote machine. This section of the document describes all the **File Check** fields available.

8.1 Error File (required)

This field defines the name of the error file for the file check. This file is created if the file check does not succeed.

8.2 Filename (required)

The **Filename** field specifies the full path name of the file to look for.

8.3 Host (optional)

The **Host** field allows the batch designer to check for a file on a remote Avatar server. If the **Host** field is not specified, then the local host is assumed.

8.4 Minimum Size (optional)

The **Minimum Size** field allows the batch designer to set a minimum acceptable size for the given filename. The size given is in bytes so a value of 1000 would be 1000 bytes.

8.5 Maximum Size (optional)

The **Maximum Size** field allows the batch designer to set a maximum acceptable size for the given filename. The size given is in bytes so a value of 1000 would be 1000 bytes.

8.6 File Check Example

The following example demonstrates the complete definition of a **File Check** object.

EXAMPLE 31: FILE CHECK

```
#
# This checks for the file named /tmp/XXX on host vcg02. The file
# needs to be between 1000 bytes and 3000 bytes in size for this
# command to exit successfully.
#
File Check = Look for the file /tmp/XXX
  Filename      = /tmp/XXX
  Host          = vcg02
  Error File    = /dev/null
  Minimum Size  = 1000
  Maximum Size  = 3000
```

9 File Match Object

The **File Match** object allows the batch designer to check for the existence of a set of files based on a regular expression, on the local or remote machine. This section of the document describes all the **File Match** fields available.

9.1 Directory (required)

This field specifies the directory to use when looking for the files.

9.2 Error File (required)

This field defines the name of the error file for the file check. This file is created if the file match does not succeed.

9.3 Execute (required)

This field specifies the command to run when a match is found. In order to include the matched filename in the command the special variable *%filename* is used.

9.4 Filename (required)

The **Filename** field specifies the name of the file or regular expression when matching multiple files.

9.5 Host (optional)

The **Host** field allows the batch designer to check for a file on a remote Avatar server. If the **Host** field is not specified, then the local host is assumed.

9.6 Output File (optional)

This field of the command object specifies where the output of the command is to go.

9.7 File Match Example

The following example demonstrates the complete definition of a **File Match** object.

EXAMPLE 32: FILE MATCH

```
#
# This checks for all files in the directory /home/ftp/inbound
# which end in .gz and moves them into an archive directory.
# Please note: The filename expressions are standard Unix based
# regular expressions. No information on regular expressions is
# described in this manual- but there are plenty of examples and
# information on the web.
#
File Match = Look for all .gz files in /home/ftp/inbound
  Directory      = /home/ft/inbound
  Filename       = .gz$
  Error File     = /dev/null
  Execute        = mv %filename /home/ftp/archive
  Output File    = /home/ftp/archive.log
```

10File Transfer Object

The **File Transfer** object has been devalued in version 4. Use **FTP Transfer Object Instead**.

11 FTP Transfer Object

The **FTP Transfer** object allows the batch designer to transfer files between Avatar servers without compromising security. The **FTP Transfer** object differs from the **File Transfer** object, in that it requires a password, but does not have a file size limit. This object follows the RFC 959 standard protocol, so it replaces all insecure FTP scripts.

11.1 Action

The **Action** field the direction of the file transfer. There are 2 valid values: push and pull.

11.2 Destination (required)

The **Destination** field specifies the name of the file once it has been transferred. It is usually a good idea to specify an absolute path name for the file; otherwise it makes it difficult to determine where the file is being written.

11.3 Error File (required)

This field defines the name of the error file for the transfer. This file is created if the transfer does not succeed

11.4 Password (required)

The **Password** field is the password of the account defined in the **User** field.

11.5 Permissions (optional)

The **Permissions** field sets the final permissions of the file once transferred. The value provided are in the standard octal format (i.e. 777, 644) Unix permissions.

11.6 Port (optional)

The **Port** field defines which port to connect to. When not defined, the Avatar server will use the default FTP port (21) is used.

11.7 Server (required)

The **Server** field specifies the name of the remote server to transfer the file from. If more than one host is specified, then the FTP will be performed to each one of the servers specified. The format is a space separated list of hosts. This will only work if the user/password/directory combinations on the different machines are exact.

11.8 Source (required)

The **Source** field specifies the name of the file before it has been transferred. It is usually a good idea to specify an absolute path name for the file; otherwise it makes it difficult to determine which file is being read.

11.9 Timeout (optional)

The **Timeout** field sets the socket timeout on the connection. The value provided is in seconds.

11.10 User (required)

The **User** field specifies the user to transfer the file as.

11.11 FTP Transfer Example

The following example demonstrates the complete definition of a FTP transfer object.

EXAMPLE 33: FTP TRANSFER

```
#
# This will transfer the file /tmp/vcg02 from the
# host vcg02 to the local machine and save it as
# /tmp/vcg01. A timeout of 60 seconds has been placed
# on the socket connection.
#
FTP Transfer = FTP a file from vcg02
  Source      = /tmp/vcg02
  Destination = /tmp/vcg01
  User        = sysops
  Password    = password
  Server      = vcg02
  Direction   = Pull
  Error File  = /tmp/error
  Timeout     = 60
```

12 HTTP Request Object

The HTTP Request Object allows a batch designer to act as a web client and request a web page. This object is very useful for monitoring a web page or for starting cold fusion batch jobs. The HTTP Request object just pulls the text of the page and not the images; so timing is based more on the ability to connect, not the weight of the page itself.

12.1 Error File (required)

This field defines the name of the error file for the command. This file is created if the command does not succeed.

12.2 Output File (required)

This is the filename where the text of the web page will be sent.

12.3 URL (required)

This specifies the URL in which to download. Valid URL protocols are http, https and ftp.

12.4 HTTP Request Example

The following example demonstrates the complete definition of a **HTTP Request** object.

EXAMPLE 34: HTTP REQUEST

```
#
# This will go to the web site of www.vexus.ca and get the
# page. Use the 'backdoor' port to get the new page.
#
HTTP Request = Get www.vexus.ca's web site.
    URL          = http://www.vexus.ca:12345/avatar.html
    Output File   = /tmp/web.output
    Error File    = /tmp/web.error
```

13 Syslog Object

The **Syslog** object allows the batch designer to send message to syslog. Typically batches will send a message to syslog at the start and end of the batch run – this object simplifies this requirement.

13.1 Ident (optional)

This identifies the syslog message string. The default will be the batch name.

13.2 Message (optional)

This is the message to be written to syslog.

13.3 Priority (optional)

The priority sets the syslog priority. The valid priorities are one of:

- Info (default value if not supplied)
- Notice
- Warning
- Error
- Critical

13.4 Syslog Example

The following example demonstrates the complete definition of a **Syslog** object.

EXAMPLE 35: SYSLOG OBJECT

```
#
# This will write a critical message to syslog.
#
Syslog = Send an critical message to Syslog.
  Message = Batch exited with an error.
  Priority = Critical
  Ident = Example Batch
```

14 Special Command File Syntax

The syntax for the command file has a few exceptions which have not been noted yet. The exceptions being for output files. All files created during a batch run can be created new, auto-incremented, or append to an existing file. The special file syntax applies to the following fields:

- Die Command Output File
- Error File
- Output File
- Post Process Output File
- Pre Process Output File
- Warn Command Output File

This section of the document outlines each type and how to use them.

14.1 Creating a New File

So far the document has only new file creation. This is usually called clobbering; where the old file is clobbered with the contents of the new file. The following example demonstrates the creation of an output file which will get clobbered each run.

EXAMPLE 36: CLOBBERING AN EXISTING OUTPUT FILE

```
#
# This command will run the find command and send the
# output to the file /tmp/find.out. Each time the command
# runs, the file /tmp/find.out will be clobbered.
#
Command= Perform a simple find on the system.
Execute      = find / -print
Error File   = /dev/null
Output File  = /tmp/find.out
```

14.2 Appending To The End of An Existing File

If the contents of an output file needs to be kept for a reason, there is a syntax in the command file which allows the designer of the batch job append the contents of the file to the end of an existing file. The following example demonstrates how to append to a file.

EXAMPLE 37: APPENDING TO THE END OF AN EXISTING FILE

```
#
# This command will run the find command and send the
# output to the file /tmp/find.out. The output of the find
# command will be appended to the file /tmp/find.out. If the
# file does not exist, it will be created.
#
Command = Perform a simple find on the system.
Execute      = find / -print
Error File   = /dev/null
Output File  >> /tmp/find.out
```

Notice the syntax of the **Output File** field. Instead of using a single equals sign =, a double greater than sign is used; >>. This syntax is borrowed from Unix shells which use the >> to append to a file.

14.3 Auto-Incrementing a File

If the contents of an output file needs to be kept for a reason, but each run needs to create a new file, then the batch designer may tell the server to auto-increment the file. The following example demonstrates how to specify an auto-incrementing filename.

EXAMPLE 38: AUTO-INCREMMENTING FILES

```
#
# This command will run the find command and send the
# output to the file /tmp/find.out. If the file /tmp/find.out
# already exists, then it will be moved to /tmp/find.out.000
# and the contents of the new run will be put into
# /tmp/find.out
#
Command = Perform a simple find on the system.
Execute   = find / -print
Error File = /dev/null
Output File += /tmp/find.out
```

Notice the syntax of the **Output File** field. Instead of using a single equals sign =, a plus equals sign is used; +=. If the command is run successively, then the file */tmp/find.out.000* will be moved to */tmp/find.out.001*, the current */tmp/find.out* will be moved to */tmp/find.out.000*, and the new file will be */tmp/find.out*.

14.4 Special Command File Variables

Special variables and environment variables have been created to aid the person in assembling batch jobs.

14.4.1 Pre-Defined Environment Variables

The following table outlines the environment variables defined and available for use within the command file.

TABLE 14.1 PRE-DEFINED ENVIRONMENT VARIABLES

VARIABLE	DESCRIPTION
RUNDATE	This is the run date the server currently thinks it is. If the batch is being run on the 1st of June 1990, then the run date will be set to 19900601.
YEAR	This is the year of the run date. In the above example this would be set to a value of 1990.
MONTH	This is the month of the run date. In the above example this would be set to a value of 06.
DAY	This is the day of the run date. In the above example this would be set to a value of 01.
PATH	This is a basic PATH environment variable. Set to: /bin:/sbin:/usr/bin:/usr/sbin
HOME	Set to the home directory of the user the batch runs as.
SHELL	Set to the shell of the account the batch runs as.
LOGIN	Set to the name of the account the batch runs as.
LOGNAME	Set to the name of the account the batch runs as.
ABSTIME	This variable will be swapped with the systems absolute time in seconds. ⁵

⁵ UNIX systems keep time in the number of seconds elapsed since January 1st 1970.

VARIABLE	DESCRIPTION
START_HOUR	This is the hour the batch started.
START_MINUTE	This is the minute the batch started.
START_SECOND	This is the second the batch started.
START_DAY	This is the day the batch started. This has nothing to do with the run date.
START_MONTH	This is the month the batch started. This has nothing to do with the run date.
START_YEAR	This is the year the batch started. This has nothing to do with the run date.

The pre-defined environment variables can be used just like any other environment variable. The following example demonstrates the use of some of the pre-defined environment variables.

EXAMPLE 39: PRE-DEFINED ENVIRONMENT VARIABLES

```
#
# This demonstrates the user of some of the pre-defined
# environment variables.
#
Command = Example Command Object
Execute   = find / -ls
Output File = /tmp/find.$RUNDATE
Error File = /tmp/find_error.$RUNDATE
```

14.4.2 Pre-Defined Variables

Along with predefined environment variables, there are some pre-defined variables. The difference being is that the pre-defined variables are defined for each object; while environment variables hold for all objects within a command file. The following table lists all of the pre-defined variables for use.

TABLE 14.2 PRE-DEFINED VARIABLES

VARIABLE	DESCRIPTION
%BATCH_NAME	This will be expanded to the batch name.
%COMMAND_NAME	This will be expanded to the name of the current command.
%ERROR_FILE	This will be expanded to the value in the Error File field.
%OUTPUT_FILE	This will be expanded to the value in the Output File field.
%RUNNING_LOG_FILE	This will be expanded to the value of the batches running log file.
%CONTACT	This will be expanded to the value of the Contact field.
%USER	This will be expanded to the value of the User field.
%SITE	This will be expanded to the value of the Site field.
%HOST	This will be expanded to the value of the Host field.
%HOSTNAME	This will be expanded to the value of the hostname of the machine.

The pre-defined variables are useful for creating a very robust batch process. The following example demonstrates the use of the pre-defined variables.

EXAMPLE 40: PRE-DEFINED VARIABLES

```
#
# This demonstrates the use of some of the pre-defined
# variables. In this case if the command exits with an
```

```
# error, the Avatar server will mail the error file to
# the operator account.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = find.out
  Error File   = find.error
  Die Command  = mail -s 'Avatar Error Report' < %ERROR_FILE
```

15 Command File Examples

This section of the document provides examples to the many features of Avatar. In all of the examples the **Execute** field will be set to perform a *find* on the complete file system.

EXAMPLE 41: SIGNALING AN ERROR

```
#
# This example will signal an error by mailing
# the user named operator if an error happens.
# It will mail the Error File file. We will change
# the acceptable error code to 1 so an exit of 0
# will force an error.
#
Command = Example Command Object
  Execute      = find / -ls
  Exit Codes   = 1
  Die Command  = mail -s 'Avatar Batch Error' operator <
%ERROR_FILE
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
```

EXAMPLE 42: LOAD BALANCING A RESOURCE INTENSIVE COMMAND

```
#
# This example will poll three machines: vcg01,
# vcg02, and vcg03. It will pick the machine with
# the lowest machine load usage and request a remote
# command execution on the server chosen.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Hosts        = vcg01, vcg02, vcg03
  User         = operator
```

EXAMPLE 43: SETTING A WARN TIME (RELATIVE)

```
#
# Setting a warn time means that if the command runs too
# long or runs past a given time the server will send an
# alarm. In this case we will set the warn time for 2 minutes
# and 25 seconds. If the command runs too long we will mail
# the operator and tell them the command is running too
# long. The Warn Time will NOT kill the process, it just sends
# out an error.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Warn Time    = +00:02:25
  Warn Command = mail -s "Avatar Warning: Command running too
long." operator
```

EXAMPLE 44: SETTING A WARN TIME (ABSOLUTE)

```
#
# In this example, the warn time will be set to a given time.
# This means if the command runs past the given time, the warn
# command will be sent. We will set the warning time to
# 7:29:33pm
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Warn Time    = 19:29:33
  Warn Command = mail -s "Avatar Warning: Command running too
long." operator
```

EXAMPLE 45: SETTING A DIE TIME

```
#
# The Die Time is the same as the Warn Time except
# that it kills the process. Normally you can use the
# Warn Time and Die Time in conjunction with one
# another. In this example, we will perform the
# find command with a Die Time of 2 minutes 12 seconds
# without setting a warn time.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Die Time     = +00:02:12
  Die Command  = mail -s "Avatar Error: Avatar process killed
command. Ran too long." Operator < %ERROR_FILE
```

EXAMPLE 46: SETTING A COMMAND START TIME

```
#
# The Start Time of a command has nothing to do with the
# start time of the batch. The command Start Time specifies
# when the command should start. If the Avatar process
# is ready to launch a command, and the start time is set
# the Server will wait until the start time has elapsed before
# launching the command. This example will tell the server
# to wait for 7 minutes and 22 seconds before starting the
# command.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Start Time   = +07:22:00
```

EXAMPLE 47: SETTING A MINIMUM RUN TIME

```
#
# When a minimum run time is set the person who
# assembled the batch is expecting the command to
# run for as least as long as the time stated. If
# it does not, then this is considered an error.
# This example sets the minimum run time to 1 minute.
# If the command does not run for at least 1 minute
# then the server will mail the operator the error
# log.
#
Command = Example Command Object
Execute      = find / -ls
Output File  = /tmp/find.out
Error File   = /tmp/find.error
Minimum Time = 00:01:00
Die Command  = mail -s "Avatar Error" operator < %ERROR_FILE
```

EXAMPLE 48: SETTING A RETRY COUNT

```
#
# When a retry count is set it means the command itself
# might be prone to error. (for whatever reason) Since
# the operator wants to minimize the amount of human
# interference with the batch process, setting a
# retry count tells the server to try the command
# a given amount of tries before giving up. This example
# sets a retry count of 5 times with a 5 minute gap
# between tries. If for some reason after 5 tries the
# command still fails then the server will mail the
# operator with the Error File.
#
Command = Example Command Object
Execute      = find / -ls
Output File  = /tmp/find.out
Error File   = /tmp/find.error
Retry Count   = 5
Retry Time   = 00:05:00
Die Command  = mail -s "Avatar Error" operator < %ERROR_FILE
```

EXAMPLE 49: SETTING MULTIPLE HOLIDAYS

```
#
# Some commands require that they not run on several
# days. Setting multiple holidays for commands is just
# a matter of defining them. This example sets a holiday
# for both Christmas and New Years Day for the command.
#
Command = Example Command Object
Execute      = find / -ls
Output File  = /tmp/find.out
Error File   = /tmp/find.error
Holiday      = Christmas(25/12)
Holiday      = NewYears(01/01)
```

EXAMPLE 50: CREATING A COMPLEX RANGE

```
#
# The ability to create a non-regular schedule
# for a command is one of Avatar's strongest
# features. This example will schedule the command
# to run at the end of the month. If the end of the
# month happens to land on a weekend, the command
# will be run on the Friday.
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Range        = M:(end,5/end-1,5/end-2)
```

EXAMPLE 51: AUTOMATICALLY PAUSING THE BATCH

```
#
# In some cases a batch job may interfere with
# normal day to day operations. So the fields
# Suspend At/Resume At allow the designer to
# suspend the batch during noted critical time
# periods. This example paused between 9am and
# 5 pm6
#
Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Suspend At   = 09:00:00
  Resume At    = 17:00:00
```

EXAMPLE 52: BRANCHING COMMANDS

```
#
# The ability to jump (or branch) to commands
# is very useful Avatar provides this facility with
# the Branch On Error/Branch On Success fields.
# This example branches to one command on success
# an another on failure. Note how the control flow is
# stopped by issuing a command which returns 0;
# but does nothing.
#
Command = global
  Output File  = /dev/null
  Error File   = /dev/null

Command = Example Command Object
  Execute      = find / -ls
  Output File  = /tmp/find.out
  Error File   = /tmp/find.error
  Branch On Error = Error Command
  Branch On Success = Success Command

Command = Error Command
  Execute = mail -s "Avatar Error" operator < %RUNNING_LOG_FILE

Command = Error Stop Command
  Execute = /bin/true

Command = Success Command
  Execute = ls /tmp
```

⁶ Keep in mind; if a batch is suspended during a time which the batch were scheduled to start, then the batch will not start again.

15.1 Database Backup Example

So far all of the examples have been simple one command batches. The next example provides a more realistic and comprehensive example of what a real command file looks like.

EXAMPLE 53: SYSTEM AND DATABASE BACKUP EXAMPLE

```
#
# One of the most common tasks for a nightly procedure
# is to back up a system. In this case both the system
# and the database are to be backed up.
#

#
# Set required environment variables.
#
setenv ROOT_DIR      $HOME/Avatar
setenv LOG_DIR       $ROOT_DIR/logs
setenv DATABASE      /export/local/sybase/Avatar
setenv DATA_FILE    /tmp/dbdump.out
setenv TAPE_DEV      /dev/rmt/0

#
# Create the global command. The global command
# sets various attributes. One notable attribute is
# the Range. It says that all the commands will run from
# Monday to Friday. (just in case someone launches the
# batch on the weekend, nothing will happen)
#
Command = global
    Site                = Toronto Office
    Contact              = Mike Glover x 7811
    Output File          = /dev/null
    Error File           = $LOG_DIR/backup_error.$RUNDATE
    Weekly Schedule      = 1-5
    Die Command          = mail -s "Avatar Backup Died" operator <
%ERROR_FILE

#
# Mail the operator telling the operator we have started.
#
Command = Start of Batch E-Mail
    Execute              = mail -s "Avatar Backup Started" operator
    Error File           = /dev/null
    Critical              = No

#
# This makes sure there is a tape in the drive by
# performing a retention on the tape. If there is
# no tape, an error will occur.
#
Command = Tape Retention
    Execute              = mt -f $TAPE_DEV ret

#
# Dump the database to the output file. The
# script dumpdb is a home grown script which
# dumps a database to a specified file.
#
Command = Dump the database.
    Execute              = dumpdb -o $DATA_DILE -d $DATABASE
    Output File          = $LOG_DIR/dump.out
    Error File           = $LOG_DIR/dump.log
    Pre Process          = rm -f $LOG_DIR/dump.out
```

```
#
# We always want to start the system backup at 11:30
# at night. If someone changes the schedule it could
# be a problem. So for this reason we will set a
# Start Time for this command.
# Note we will pause this job between 2am and 3am
# if it runs that long.
#
Command = System Backup
  Execute      = fsdump -f $TAPE_DEV / /export /usr /home
  Start Time   = 23:30:00
  Output File  = $LOG_DIR/fsdump.out
  Pre Process  = rm -f $LOG_DIR/fsdump.out
  Suspend At   = 02:00:00n
  Resume At    = 03:00:00n

#
# If the batch process makes it to this point everything
# has gone well, so we can clean up the file systems.
# We will set this to a non-critical command. No point
# in tempting fate.
#
Command = Clean Up
  Execute      = rm -f $DATA_FILE
  Error File   = /dev/null
  Critical     = No

#
# Take the tape drive offline. We leave this as
# a critical process because if we can't eject the
# tape then we have a problem and the operator
# should know about it.
#
Command = Eject the tape.
  Execute      = mt -f $TAPE_DEV offline
  Error File   = /dev/null

#
# Mail the operator the running log file when we are done.
#
Command = End of Batch E-Mail
  Execute      = mail -s "Avatar Backup Finished" operator <
%RUNNING_LOG_FILE
  Error File   = /dev/null
  Critical     = No
```