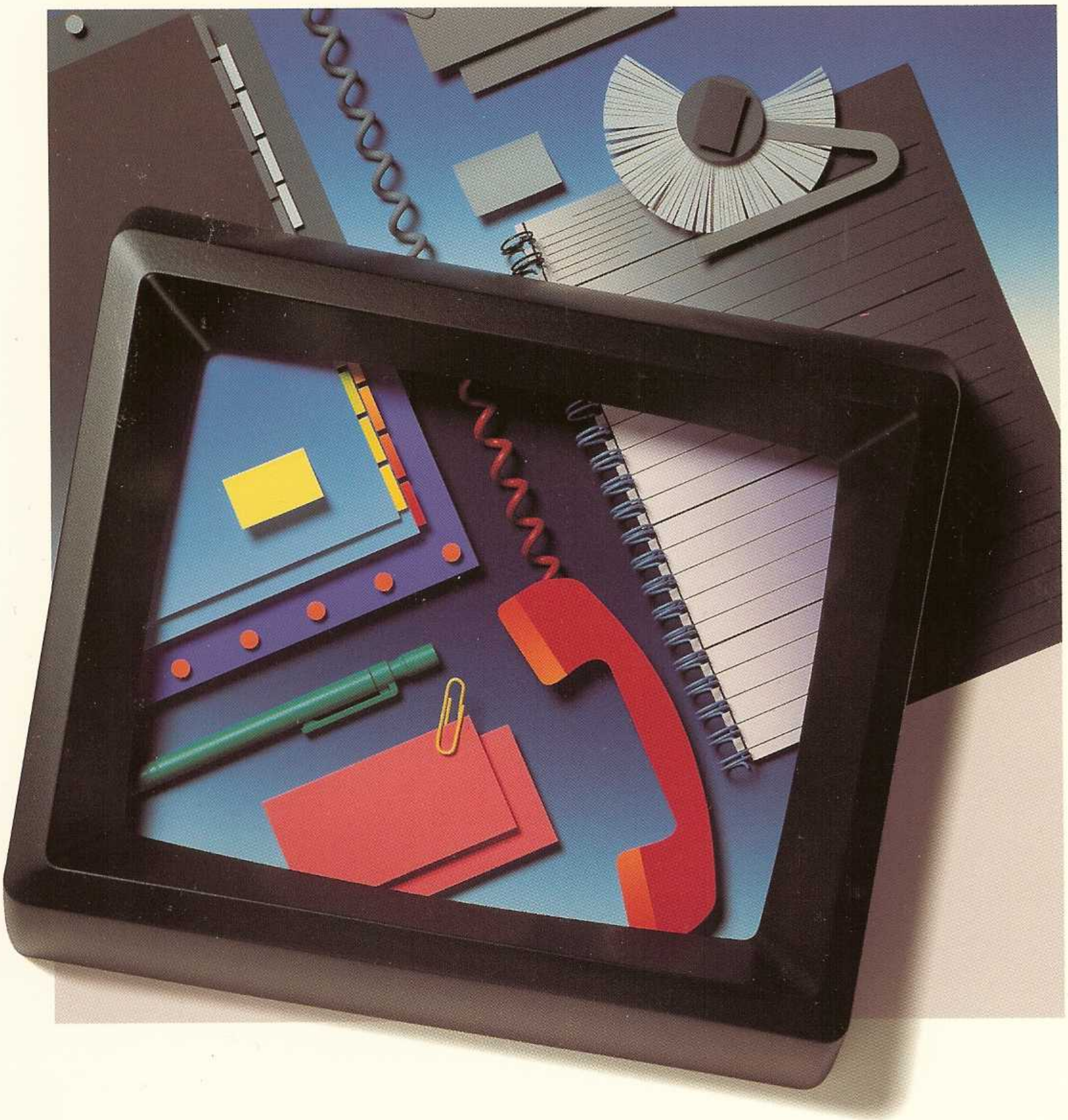
Lotus Agenda
Working with Definition Files



Release 2.0

Lotus Agenda
Working with Definition Files

Release 2.0

Contents

Chapter 3 Using Definition Files 3-1

Chapter 4 Patterns 4-1

Chapter 6 Converting and Importing Text 6-1

Chapter 7 Debugging a Definition File 7-1

Appendix A What's New in TXT2STF for Agenda 2.0 A-1

Appendix B Structured Files B-1

Appendix C Quick Reference C-1

Appendix D Error Messages D-1

Index

Introduction

Working with Definition Files provides information about writing and debugging definition files to use with the Lotus Agenda® TXT2STF utility.

TXT2STF is a versatile utility that prepares text to be imported into Agenda. To guide TXT2STF, you can create definition files that provide custom instructions for converting the contents of text files into Agenda items, categories, and notes.

How this Book Is Organized

Working with Definition Files is organized into seven chapters and four appendixes that provide the following information:

- **Chapter 1, "Getting Started with Definition Files,"** describes what a definition file is, when you need to write definition files, and provides an example of how you might write a definition file to convert a text file.

  Read this chapter to get an overview of what definition files are and when to use them.

- **Chapter 2, "Creating a Definition File,"** describes how to create a definition file and what a definition file contains. This chapter also describes how TXT2STF uses the definition file to process a text file based on the definition file you write.

  Read this chapter for a basic understanding of how to write a definition file.

- **Chapter 3, "Using Definition Files,"** provides examples of what you can do with a definition file. This chapter presents examples of text files along with definition files that convert the text files, and shows how the converted information looks after you import it into Agenda.

Read this chapter to see some examples of how actual definition files convert text for importing into Agenda.

- **Chapter 4, "Patterns,"** provides specific information about writing patterns in definition files, including a list of match-control characters you can use in patterns.

  Read this chapter to learn about patterns in definition files, or as a reference when creating definition files.

- **Chapter 5, "Definition File Commands,"** provides specific information about using commands in definition files, including a list of all the commands with examples.

  Read this chapter to learn about definition file commands, or as a reference when creating definition files.

- **Chapter 6, "Converting and Importing Text,"** describes how to run TXT2STF to convert your text file into a structured file and how to import the structured file into Agenda.

  Read this chapter when you're ready to run TXT2STF and use your definition file to convert text for importing into Agenda.

- **Chapter 7, "Debugging a Definition File,"** explains how to debug your definition file and gives tips on how to write better definition files.

  Read this chapter when you've created a definition file and you want to test how accurately it converts your text.

- **Appendix A, "What's New in TXT2STF for Agenda 2.0"** lists new and changed TXT2STF features that support Agenda Release 2.0 features, enhance TXT2STF performance, and give you greater control over how TXT2STF converts text files into structured files.

  Read this chapter if you've used a previous release of TXT2STF, and you want a quick overview of what's changed.

- **Appendix B, "Structured Files,"** provides details about structured files, which are the intermediate ASCII files created by TXT2STF to hold the converted text. This chapter describes the special tags that TXT2STF embeds in the structured file that provide Agenda with details about how to import the converted text.

  Read this chapter when you are debugging a definition file, or when you are creating your own structured files using a tool other than TXT2STF.

- **Appendix C, " Quick Reference,"** is a quick reference to definition file patterns, match-control characters, commands, and predefined variables for commands.

  Read this appendix when you need a quick review of definition file pattern and command syntax.

- **Appendix D, "Error Messages,"** lists the error messages you may see when running TXT2STF with definition files and describes possible reasons for them and possible solutions.

  Consult this chapter if you receive error messages when you run TXT2STF with a definition file.

Before You Begin

You should understand how to use Agenda before you begin to write definition files. In particular, you should be familiar with the way Agenda imports information. For more information, see Chapters 23 and 24 in the *User's Guide*.

Typographic Conventions

Working with Definition Files uses the following typographical conventions:

Information that you type is in a different typeface.

Example

```
Call Meg tomorrow
```

Agenda commands, settings, and choices are in boldface type.

Example

To import the contents of a text file, use the **File Transfer Import** command.

Chapter 1
Getting Started with Definition Files

Agenda lets you import information from other sources into your Agenda file. For example, you can import text from a memo created by using a word processor. In some instances, such as when you import the contents of a text file into an item note, this involves a simple combination of two or three keystrokes.

In other cases, you might want to import information that is formatted in a more complicated manner. For example, you might want Agenda to create new items, modify the existing category hierarchy, or even add to the category hierarchy when it imports text. So that Agenda can import information in the way that you want, you must first structure the text into a format that Agenda understands. Agenda provides two utilities for this purpose:

- TXT2STF (Text to Structured File) formats generic text files and is discussed in this book.
- LM2STF (List Manager to Structured File) formats files created by the Lotus Metro List Manager. For information on LM2STF, see Chapter 23 of the *User's Guide*.

This book focuses on the use of definition files with the TXT2STF utility. Definition files guide the activities of the TXT2STF utility and tell TXT2STF how to process text files to be imported into Agenda.

Note The information TXT2STF converts must be in an ASCII text file. Most software products, such as word processors, add hidden formatting codes to files. However, most products also let you create ASCII text files (which do not include hidden formatting codes). For example, a Lotus 1-2-3® PRN file, created by printing to a file, is such a text file.

In this Chapter

This chapter describes how to use definition files and provides a simple example that illustrates basic concepts about definition files and how they work.

This chapter, along with Chapter 2, provide background information that describe why and how you create definition files. Even if you are familiar with programming concepts and/or with Agenda, it is still important to read through both of these chapters before trying to create your own definition files. The concepts and examples in these two chapters will help you understand the unique features of definition files and the way they interact with TXT2STF.

For more information on what to put in a definition file, see Chapter 2. For more information about importing text files, see Chapter 23 in the *User's Guide.*

About Definition Files

TXT2STF is a versatile utility, and can process many different types of text. For example, TXT2STF can process text files containing information as varied as electronic mail, online news stories, magazine article abstracts, tabular output from worksheets, and records from a traditional database.

TXT2STF processes the contents of text files, structuring the text so that Agenda can import specific portions of text as items, categories, and notes. By default, when TXT2STF formats a text file, it makes the first 350 characters in each new paragraph into an item, and makes the remaining paragraph text into a note for the item.

To have more control over how Agenda formats imported information, you can create a definition file. A definition file describes patterns of text in the text file, and contains commands that tell TXT2STF how to convert the text identified by these patterns into items, categories, and notes. The definition file is tailored to follow the layout and content of the text file.

The definition file is created before you run TXT2STF, and is used when you request the definition file in the TXT2STF command.

TXT2STF converts the text file according to the instructions in the definition file, and puts the results in a structured file (text in the file is structured in a way that Agenda can interpret).

Figure 1-1 shows the flow of information from text file to structured file when you run TXT2STF with a definition file.

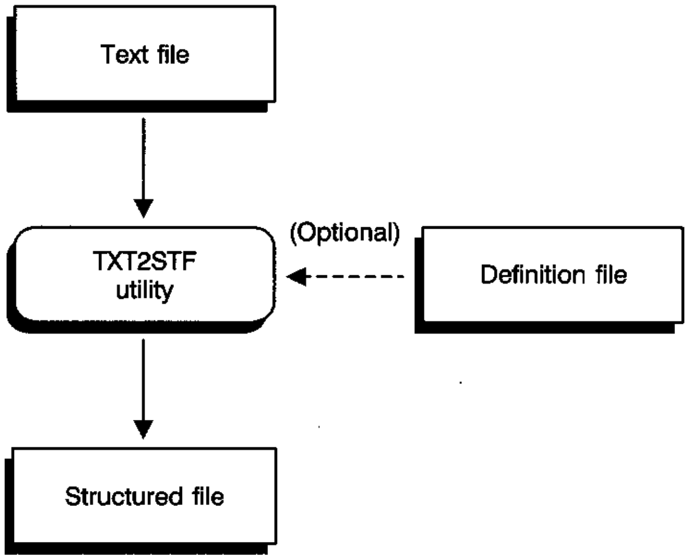


Figure 1-1 ***Converting a text file to a structured file with an optional definition file***

You import the structured file into Agenda by using the Agenda **File Transfer Import** command. (See Chapter 23 in the *User's Guide.*) This command tells Agenda how to incorporate information from the structured file into your Agenda file.

When to Use a Definition File

You can use a definition file any time you run TXT2STF to convert a text file for importing. Using a definition file is particularly helpful if

- You have many text files with the same format that you want to import into Agenda, such as a series of files that contain legal information, all of which are formatted in a similar way.
- You frequently import specific types of files, such as electronic mail files.

In these cases, you can use the definition file to automate the conversion of specific types of text into items, categories, and notes. For example, the definition file can tell TXT2STF to start a new item when it finds a subject line in an electronic mail file, to make the sender and recipient names into categories, and to put the body of the electronic mail into a note for the item. If you use TXT2STF without a definition file and then import the text, you would need to do much of this formatting yourself in Agenda after you import the text.

Benefits of Using Definition Files

There are various benefits to using a definition file to format text before you import it into Agenda. When you use a definition file

- You control exactly what text Agenda will interpret as items, categories, and notes.
- You can set dates and make other category assignments for items.
- You can predefine the category hierarchy.

When Other Approaches Are Appropriate

You don't *have* to use a definition file to import a text file. If you only need to import one or two files of a given format, you might not want to take the time to write a definition file. In that case, you can

- Run TXT2STF without a definition file and import the structured file. (See Chapter 23 of the *User's Guide.*)
- Import the text file into an Agenda note and then, within the note, make selected note text into items and categories. (See Chapter 12 of the *User's Guide.*)

Or, you might want to use a different tool to create the structured file. For example, if the text file is patterned in a complex manner, you might prefer to write a program in a programming language such as BASIC or PASCAL to convert the text file into a structured file. For more information about what the structured file should contain, see Appendix B.

Creating a Definition File

This section describes one way to create a definition file. It presents a sample text file and describes the procedures you follow to create a corresponding definition file.

As you read this section, remember that there are usually several ways to convert the same text file. For example, you might want to collect different item text from the text file. Your Agenda file might organize information in a different manner than is outlined in this example, so you might need to create different categories.

Also remember that the definition file described in this section is tailored to the sample text file. You can use strategies described in this section in definition files that you create, but you need to adapt them to fit your own text files, whose contents and format will differ from the text file shown in this example.

To create a definition file that can convert a text file into items, categories, and notes, you must

- Analyze the text file to determine what text to convert
- Write the definition file, adding information that tells TXT2STF how to convert the text

Analyzing the Text File

The first step in creating a definition file is to analyze the text file to determine which text to convert into items, categories, and notes.

Figure 1-2 shows a sample text file that you might want to bring into an Agenda file. This sample text file contains a memo sent to an Agenda user by electronic mail.

```
SEND_VIA NODE_42

Date:    November 23
To:      Jill
From:    Linda
Re:      Lunch

Can't make lunch appointment today, can we reschedule for next week?
```

Figure 1-2 *A sample text file*

In the current example, you could convert

- The mail date into a When date
- The recipient and sender names into categories
- The mail subject line (labeled Re:) into an item
- The body of the memo into a note for the item

After you determine which text to convert, you must tell TXT2STF how to identify the text. To do this, look through the text file and locate text strings that label the text to convert. In the current example, this is easy to do, since electronic mail files contain standard labels that describe the information in the file. Date: labels the date, To: labels the memo recipient, and so forth.

As shown in Figure 1-3, you want TXT2STF to convert text following Date: into a When date, the names following To: and From: into categories, the text following Re: into the item, and the remaining text into a note for that item. By identifying these text strings, you give TXT2STF patterns it can use to locate the text you want to convert.

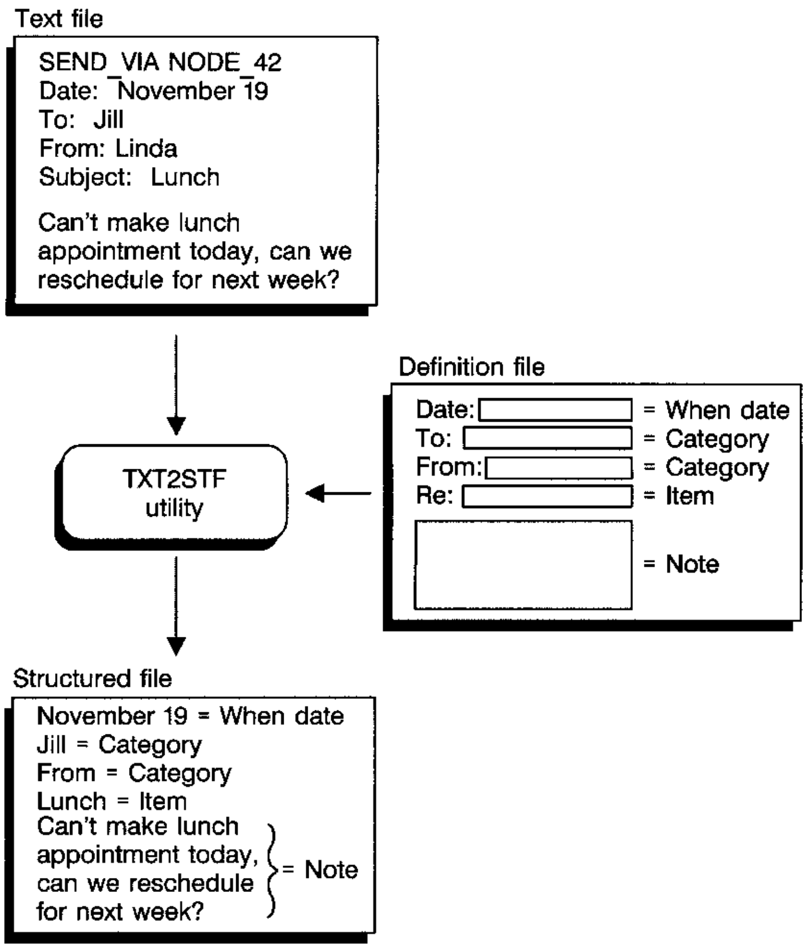


Figure 1-3 *Using a definition file to convert text to import into Agenda*

Writing the Definition File Patterns and Commands

Now that you know what information you want converted and have text strings that identify the text, you have enough information to write a definition file. For each text line you want to convert, you must

- Add a pattern that describes to TXT2STF the text strings you want TXT2STF to find
- Provide the commands required to convert the text located by the pattern into an item, category, note, or some combination

The combination of a pattern and its commands is called a statement. A definition file statement must start with a pattern and can include any number of commands.

Adding patterns

Patterns describe the text strings you want TXT2STF to find in the text file. By locating any of these lines in the text file, TXT2STF locates a line with information to be converted. To convert the electronic mail message in Figure 1-2, you can add the following patterns:

```
"^Date\:"
"^To\:"
"^From\:"
"^Re\:"
```

In the above patterns, the caret (^) tells TXT2STF that the text string begins a new line. The combination of backslash (\) and colon (:) in the above patterns tell TXT2STF that each colon (:) is a regular text character (otherwise, the colon (:) starts a special pattern condition). You include the carat (^), backslash (\), and other characters in patterns to provide additional information about the text string. For more information about patterns, see Chapter 4.

When TXT2STF matches a text line with a pattern in the definition file, TXT2STF executes all commands related to the pattern. So far, you have created the patterns that TXT2STF can use to locate text to be converted. Now you can add commands to each pattern to tell TXT2STF how to convert the text that matches the pattern.

Converting the When date

Since "Date:" is the first text pattern in the sample text file, you can start the process of determining how to convert text with this text line.

When TXT2STF finds "Date:" at the start of a line in an electronic mail file, you want TXT2STF to

- Begin gathering information for a new item, consisting of text for the item itself as well as assignments for the item
- Skip over the word Date and the colon (:) that follows it so that the string Date: does not occur in the When date
- Convert the date into a When date

You set aside some computer memory for TXT2STF to use while constructing your item and its note and categories by using the @start command. To skip over text on a line, use @skip. To make text into a When date, use @date. Therefore, to convert text into a When date, you add the following pattern and commands:

Pattern	*Commands*

`"^Date\:"`	`@start`
	`@skip`
	`@date`

Note To simplify the current discussion, this chapter shows definition file commands without their accompanying arguments. To see the complete commands with arguments, refer to Figure 1-4.

Converting the To and From categories

You can continue by creating the To and From categories, since To and From are the next strings of interest in the text file.

In this example, To: labels the electronic mail recipient name and From: labels the sender name. You can have the definition file organize sender names as children of the From category and recipient names as children of the To category. To do this, the definition file needs to keep track of both the parent category (To or From) and the child category (the name of the recipient or sender).

When TXT2STF finds the string To: or From: at the start of a line in an electronic mail file, you want TXT2STF to

- Convert the word To or From that starts the text line into a parent category
- Convert the recipient or sender name into a child category of To or From

You need to use two different portions of text from the same line: the parent category and the child category. One way to do this is to copy the parent and child categories into variables. Then, you can include these variables in the @custom_category command that assigns the child to the parent category. (For information about variables, see Chapter 5). For example:

Pattern	*Command*
`"^To\:"`	`@trim` (copies the parent category to a variable) `@trim` (copies the child category to a variable) `@custom_category`

Since the same processing is required for both the To and From categories, you can use a single statement to convert both types of text lines. To do this, you need to use a pattern that is generic enough to match both "From:" and "To:".

You can use the pattern "^.*\:" for this purpose. The carat (^) is the start-of-line character that tells TXT2STF that the string begins a new line in the text file. The combination of period (.) and asterisk (*) specifies that the string contains one or more characters. The combination of backslash (\) and colon (:) specifies that the string ends in a colon (:). The pattern "^.*\:" therefore matches any string that starts at the beginning of a text line and ends in a colon (:). The following single statement handles all To and From lines in the electronic mail message:

Pattern	*Commands*
`"^.*\:"`	`@trim` (copies the parent category to a variable) `@trim` (copies the child category to a variable) `@custom_category`

Converting the item and note

When TXT2STF finds the string "Re:" at the start of a text line, you want TXT2STF to save the rest of the line as item text.

The Agenda view into which you import electronic mail organizes electronic mail items as children under the parent category, Messages. So, when TXT2STF finds the string Re: you want TXT2STF to

- Specify that the new item will be assigned to a category called Messages
- Skip the word Re and the colon (:) that follows it so the string Re: does not occur in the item text
- Convert the remaining text in the line into an item
- Create a note, starting on the next line and continuing to the end of the file

You specify that the new item will be assigned to a category by using @custom_category. To skip over text on a line, use @skip. To make text into an item, use @item2 or @item. To make a note for an item, use @note.

After TXT2STF creates the note, the item is complete. To tell TXT2STF to stop converting text for the item, use an @end command. When TXT2STF executes @end, it copies the item, categories, and note TXT2STF created to a structured file that is ready to be imported to Agenda.

The statement that creates item and note text is as follows:

Pattern	*Commands*

`"^Re\:"`	`@custom_category`
	`@skip`
	`@item2`
	`@note`
	`@end`

Ordering the statements

Because of the way TXT2STF converts a text file, the order of statements in a definition file can be very important.

To convert a text file, TXT2STF steps through the text file, line by line. For each text line, TXT2STF compares the line with all of the statements in the definition file, starting at the first statement and working down to the end of the definition file. When a pattern in a statement matches text in the text line, TXT2STF executes the commands associated with the pattern. Then, TXT2STF continues to the next unconverted line in the text file and compares that line with statements, again starting at the top of the definition file.

When you order statements in a definition file, you must be sure that the first pattern in the definition file that matches the text line is the desired pattern. When TXT2STF finds a statement that matches the current text line, it executes the commands in that statement. Then, TXT2STF moves on to the next line in the text file. No other statements are executed for the text line.

In general, any statement with a pattern that precisely matches a specific text line, and no other, should be placed early in the definition file. Put statements whose patterns are likely to match more than one text line toward the end of the definition file. If you follow this ordering scheme, TXT2STF compares text lines to statements with specific patterns before those with generic patterns. Text lines for which specific patterns exist match their patterns earlier in the conversion process, which expedites the conversion of text to structured information, and prevents them from incorrectly matching the more generic patterns.

So far, this section has developed statements that

- Convert the When date
- Convert the To and From categories
- Convert the item and note

You might consider placing these statements in the same order in which the corresponding text strings occur in the text file. This results in the following order of patterns:

Pattern	*Purpose of statement that includes the pattern*
`"^Date\:"`	Converts the When date
`"^.*\:"`	Converts the To and From categories
`"^Re\:"`	Converts the item (and note)

However, if you look at this order you'll see that the generic pattern "^.*\:" occurs before the pattern "^Re\:". This causes TXT2STF to execute the wrong statement for the electronic mail subject line. When TXT2STF compares the electronic mail subject line with definition file statements, it matches the text string "Re:" with the generic pattern, and then execute the associated commands, which convert To and From categories.

To correct this problem, put the statement containing the generic pattern at the end of the definition file, which results in the following order of patterns:

Pattern	*Purpose*
`"^Date\:"`	Converts the When date
`"^Re\:"`	Converts the item (and note)
`"^.*\:"`	Converts the To and From categories

Figure 1-4 shows the actual definition file that formats the sample electronic mail message in Figure 1-2. The commands in this definition file include the arguments necessary to actually convert the sample electronic mail text file. For more information about definition file commands and arguments, see Chapter 5.

```
"^Date\:"     @start()
              @skip(" +",1)
              @date(W)

"^Re\:"       @custom_category("Messages",,)
              @skip(" +",1)
              @item2("$")
              @note(EOF)
              @end()

"^.*\:"       @trim("^","\:",N,"parent")
              @trim("\:","$",N,"cat")
              @custom_category(cat,parent,)
```

Figure 1-4 *Sample definition file*

The definition file in Figure 1-4 assumes that your text file contains only one electronic mail message, so @note specifies that the item note ends when TXT2STF reaches the end of the text file (specified by the argument EOF). If the text file contains more than one electronic mail message, you need to change this definition file slightly. (See "Creating More Than One Item" in Chapter 7.)

The definition file is now ready to convert the electronic mail text file in Figure 1-2. When TXT2STF uses this definition file to convert the electronic mail text file, TXT2STF:

1. Matches the date line in the electronic mail file with the statement containing the pattern "^Date\:".

   TXT2STF starts collecting information for a new item. TXT2STF then skips over the text string Date: and makes November 17 the When date.

2. Matches the recipient line in the electronic mail file with the statement containing the pattern "^.*\:".

   TXT2STF puts the string To into the variable called parent, puts the name Jill into the variable called cat, and then uses the contents of these variables to create a category.

3. Matches the sender line in the electronic mail file with the statement containing the pattern "^.*\:".

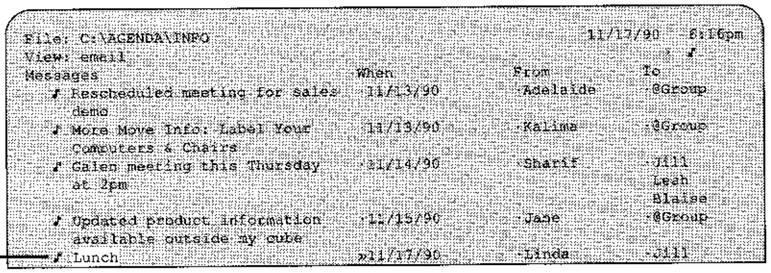
   TXT2STF puts the string From into the variable called parent, puts the name Linda into the variable called cat, and then uses the contents of these variables to create a category.

4. Matches the subject line with the statement containing the pattern "^Re\:".

   TXT2STF creates the Messages category, and then skips the text string Re: and makes the rest of the current line the item text. The caret (^) in the @item2 command tells TXT2STF to add text to the item until it reaches the start of the next line.

   TXT2STF then creates an item note, starting with the line after the electronic mail subject line continuing until the end of the file (EOF). Finally, TXT2STF ends the item and copies the item, with its categories and note, into a structured file.

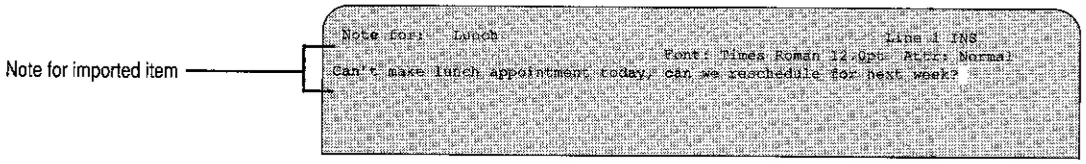
After you run TXT2STF to create a structured file, you can import the structured file into Agenda whenever you're ready. Figure 1-5 shows the converted electronic mail memo after it's imported into Agenda.



```
File: C:\AGENDA\INFO                                      11/17/90   6:16pm
View: email                                                         › ♪
Messages                          When              From           To
   ♪ Rescheduled meeting for sales ·11/13/90        ·Adelaide      ·@Group
     demo
   ♪ More Move Info: Label Your    ·11/13/90        ·Kalima        ·@Group
     Computers & Chairs
   ♪ Galen meeting this Thursday   ·11/14/90        ·Sharif        ·Jill
     at 2pm                                                          Leah
                                                                     Blaise
   ♪ Updated product information   ·11/15/90        ·Jane          ·@Group
     available outside my cube
   ♪ Lunch                         »11/17/90        ·Linda         ·Jill
```

Figure 1-5 *Converted memo text imported into Agenda*

The musical note symbol (♪) beside each item in this view indicates that the item has a note associated with it. Figure 1-6 shows the note for your imported item.



```
Note for:   Lunch                                              Line 1 INS
                                        Font: Times Roman 12.0pt  Attr: Normal
Can't make lunch appointment today, can we reschedule for next week?
```

Figure 1-6 *Note for the imported item*

Chapter 2
Creating a Definition File

You create a definition file to tell TXT2STF how to process text files to be imported into Agenda. The definition file must be created before it can be used with TXT2STF.

In this Chapter

This chapter explains

- How to create a definition file
- How to analyze the text file that you plan to import into Agenda
- How to write the patterns and commands that make up the definition file
- How TXT2STF uses the definition file to format a text file

Creating a Definition File

When you create a definition file, you perform these general procedures in the following order:

- Analyze the text file to be structured.

  You must decide how you want to convert the information in your text file into items, categories, and notes.

- Write a definition file to manage the structuring process.

  You write patterns to match specific lines in the text file, and commands to convert the text in those lines to Agenda items, categories, and notes.

- Debug the definition file.

  You run TXT2STF with your definition file and determine whether it formats information the way you want. If it doesn't, revise the definition file and test it again with TXT2STF.

This chapter provides details about the first two procedures listed above. For more information about debugging a definition file, see Chapter 7.

Analyzing the Text File

Before you write a definition file, you should become familiar with the format and contents of the text files that you want to convert.

You need to analyze your text file to

- Decide how you want to use information from the text file in Agenda.

  You need to determine which text in the text file you want to use as items, categories, and/or notes. For example, when converting a memo sent through electronic mail, you might want the memo's date to be a When date, the names of the sender and recipient of the memo to be categories, the subject of the memo to be the item, and the body of the memo to be a note for that item.

- Identify a unique text string for each part of the text file to be converted to items, categories, and notes.

  For example, in an electronic mail memo, the memo date might be preceded by an identifying label, such as Date.

- Determine how many types of information the text file contains.

  Check the text file to see whether you have a single type of information in the files you plan to convert (for instance, the file contains only memos), or whether you have more than one type of information (for example, memos and telexes in the same file). If you have more than one type of information, see "Using @start and @end" later in this chapter.

Writing the Definition File Patterns and Commands

After you analyze the text file to be structured, you can specify the patterns and commands that tell TXT2STF how to convert the text file. In this way, you construct the definition file.

You can use any text editor that creates unformatted text files to create a definition file. You must give the definition file the extension .DEF.

Tip You can write the definition file patterns and commands in an Agenda note, and export it to a text file.

What a Definition File Contains

A definition file contains statements that govern the conversion of the contents of a text file into a structured file. The statements, and the text conversions they produce, are tailored to the requirements of the text file.

For example, a definition file that converts electronic mail files contains different statements from a definition file for files that contain legal abstracts due to differences in the layout of the files and in the information to be converted.

Each statement begins with a pattern and can include one or more commands:

- Each pattern describes a line of text to be converted. Each pattern is enclosed in a set of double quotation marks (" ") that indicate where the pattern begins and ends. The quotation marks are not part of the pattern.
- Each pattern is followed by one or more commands that tell TXT2STF how to convert any text line that matches the pattern. Each command begins with an at (@) character.

Guidelines for Writing Statements

This section provides guidelines for writing statements in a definition file. For specific information about patterns, see Chapter 4. For specific information about command syntax, see Chapter 5.

Follow these guidelines when you write definition file statements:

- To make it easier to differentiate patterns from commands, add spaces or tabs after the pattern to separate it from the commands that follow it.

If the statement includes multiple commands, you also can add spaces or tabs to separate commands from each other on a single line. However, consider breaking the statement into multiple lines, and lining up commands in a column, as described below.

- To continue a statement onto another line, break the statement between commands.

  This strategy makes it much easier to read and debug statements that contain more than one command.

  Note Always break statements *between* commands; never break a statement in the middle of a command.

- To make it easier to read statements that contain multiple commands, line up commands one under the other in a column.

  Version 1.0 of TXT2STF required you to put a backslash (\) in the first column of each continued line in the larger statement. Backslashes (\) are no longer required, but you can include them in definition files if you want.

Note The maximum length of a given definition file statement is determined by the amount of standard memory available when TXT2STF executes to the command. If TXT2STF repeatedly encounters memory problems, see if you can shorten long statements.

The definition file in Figure 2-1 follows the above guidelines.

Patterns	**Commands**

`"^Date\:"`	`@start()`
	`@skip(" +",1)`
	`@date(W)`

`"^Re\:"`	`@custom_category("Messages",,)`
	`@skip(" +",1)`
	`@item2("$")`
	`@note(EOF)`
	`@end()`

`"^.*\:"`	`@trim("^","\:",N,"parent")`
	`@trim("\:","$",N,"cat")`
	`@custom_category(cat,parent,)`

Figure 2-1 *Definition file with statements on more than one line*

The sample definition file in Figure 2-1 contains three statements, each of which begins with a pattern and includes commands to process text lines. For example, the first statement begins with the pattern "^Date\:" and includes @start, @skip, and @date commands. Tabs separate patterns and commands in this sample definition file. Statements with multiple commands are continued to the next line after each command. The commands are lined up in the right hand column.

Including Comments in a Definition File

Comments let you describe what a particular statement or command accomplishes and can help you read through the definition file at a later time. You add comments for your own use; TXT2STF ignores comments when it uses the definition file.

It is a good idea to

- Start a definition file with a comment that describes the purpose of the definition file

  Since TXT2STF copies the first line of the definition file to the structured file, making the first definition file a comment line also causes the structured files it creates to start with a descriptive line.

- Include comments throughout the definition file to describe specific commands and statements

  Comments that describe specific processing performed by a definition file can be particularly useful when you debug the definition file or update it to reflect changes in text file structure.

To include a comment in your definition file, insert a number sign (#) at the beginning of a line. This character must be in the first column of the line, or TXT2STF cannot recognize the line as a comment.

Note Including many lines of comments can occasionally increase the time it takes TXT2STF to process your text.

Using @start and @end

Each definition file must include at least one pair of @start and @end commands:

- @start sets aside and initializes memory so that TXT2STF can temporarily store information that it converts.

  The patterns and commands that occur after @start and before @end convert text lines and add the resulting structured information to memory. The commands executed between each @start and @end can create a single item, a note, and any number of categories with or without category notes.

- @end bundles together the information in memory and copies it to the structured file.

  For example, if memory currently contains two categories and an item, @end assigns the item to the categories and copies them to the structured file as a single item specification. After copying information to the structured file, @end initializes memory.

If the text file contains several occurrences of the same type of information, such as several electronic mail messages all formatted in basically the same way, the same block of definition file commands can be executed over and over, each time creating a new item and copying it to the structured file. After copying an item to the structured file, the @end command initializes memory so TXT2STF can construct a new item. Then, TXT2STF progresses to the next electronic mail message and converts it, starting from the beginning of the definition file.

Figure 2-2 illustrates how @start and @end work. If you omit @start, TXT2STF will not have any memory in which to store converted information. If you omit @end, TXT2STF won't put anything in the structured file.

You need more than one @start or @end in a definition file when the file to be converted:

- Can begin or end in different ways

  For example, if an electronic mail file can begin with different labels, such as either Date: or Today:, you should include a @start for each possible file beginning and an @end for each possible file ending.

- Can contain more than one type of information

  For example, a text file might include some messages received from an internal electronic mail system and some received as telex messages. In this case, the messages will have different layouts. You must include a @start and @end pair for statements that convert the electronic mail messages, and another @start and @end for statements tailored to convert the telex messages.

Caution If multiple types of information have patterns in common, unexpected structuring can result.

For more information about @start and @end, see "How TXT2STF Builds Items, Categories, and Notes" at the end of this chapter.

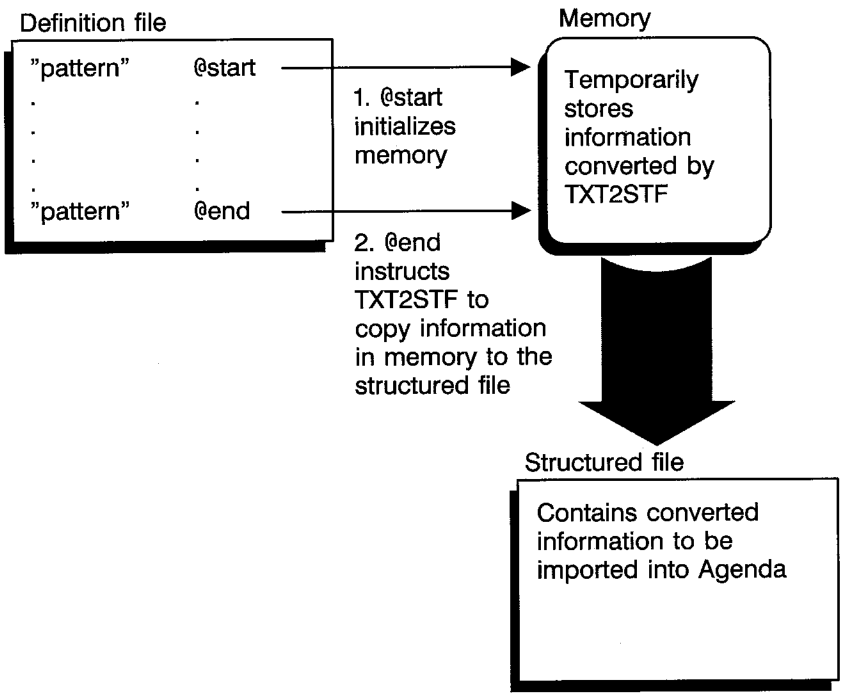

Definition file
"pattern" @start
"pattern" @end
1. @start initializes memory
2. @end instructs TXT2STF to copy information in memory to the structured file
Memory
Temporarily stores information converted by TXT2STF
Structured file
Contains converted information to be imported into Agenda

Figure 2-2 *How @start and @end work*

How TXT2STF Uses the Definition File

To write efficient definition files, it helps to understand how TXT2STF uses the definition file when processing your text file. This information also helps you when you debug the definition file to determine why information in the structured file is different from what you expected.

TXT2STF processes your text file one line at a time, starting with the first line in the text file. When it finishes with the current text line, TXT2STF proceeds to the next line in the text file. When there are no more lines in the text file, TXT2STF stops running.

The following sections provide details about

- How TXT2STF processes each line from the text file

- How TXT2STF builds items, categories, and notes from text in the text file

How TXT2STF Processes Each Text Line

TXT2STF processes each text line by copying it into memory, searching the definition file for a pattern that matches the line and, if it finds a pattern, executing the commands associated with the pattern.

Note TXT2STF *always* uses standard memory, and not expanded or extended memory.

For each line in the text file, TXT2STF performs the following steps:

1. Copies a line of the text file into memory

   In this document, this line is often referred to as the current text line in memory, or the current text line.

   TXT2STF determines that it has copied the entire text line when it encounters a carriage return character (ASCII decimal 13) or the alternate separator character as specified by using the TXT2STF /S option. (See Chapter 6). The maximum length for a text line is 512 characters.

   Caution If a text line is longer, TXT2STF treats it as more than one text line, stopping the first line at 512 characters and using the remaining characters as a second text line. If the current statement ends when TXT2STF is only part-way through this second text line, TXT2STF discards the unprocessed text characters and continues to the next unprocessed text line in the text file.

2. Searches the definition file from the top, looking for a pattern that matches the current text line in memory

   As soon as TXT2STF finds a pattern that matches the current text line, it stops searching the definition file. (TXT2STF does not later search for other patterns that might match the current text line.)

3. Executes the commands associated with the pattern that matched the text line, one after the other

   The commands convert the current text line. TXT2STF executes all commands associated with the current pattern. When there are no more commands for the pattern, TXT2STF continues to Step 4.

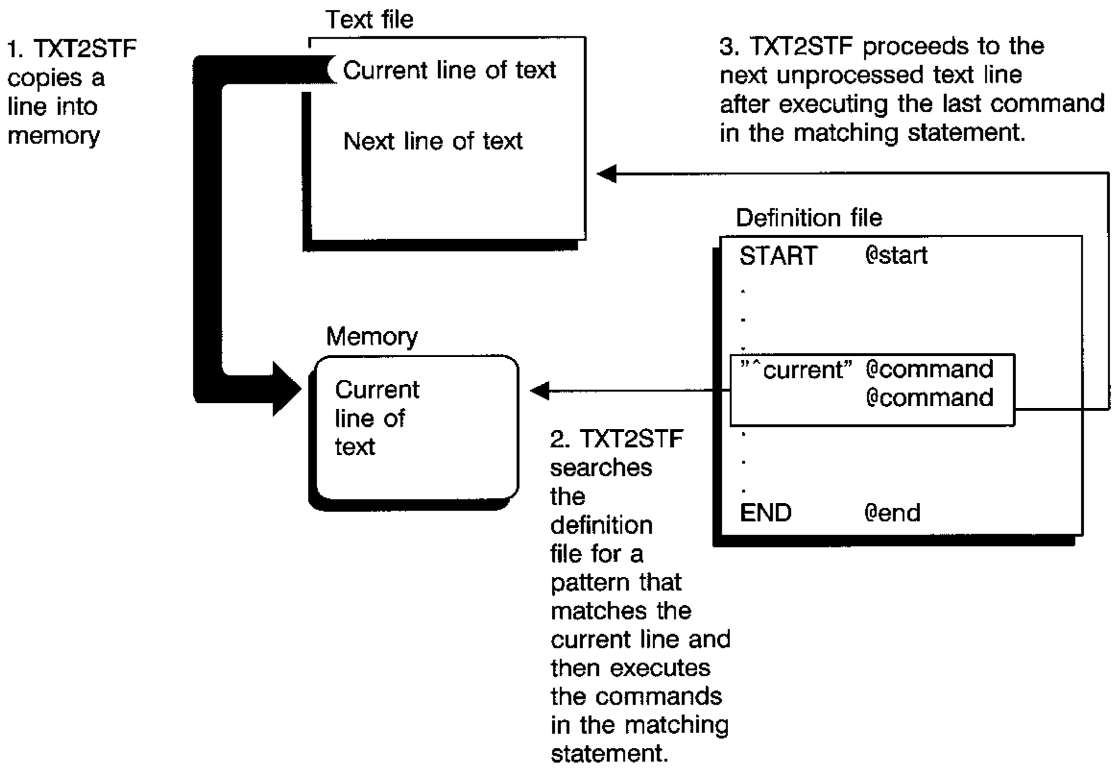

Text file
1. TXT2STF copies a line into memory
Current line of text
Next line of text
3. TXT2STF proceeds to the next unprocessed text line after executing the last command in the matching statement.
Definition file
START @start
"^current" @command
@command
END @end
Memory
Current line of text
2. TXT2STF searches the definition file for a pattern that matches the current line and then executes the commands in the matching statement.

Figure 2-3 *How TXT2STF processes text file lines when using a definition file*

How TXT2STF Builds Items, Categories, and Notes

As TXT2STF processes lines from the text file, TXT2STF formats the lines into items, categories, and notes. The commands in your definition file tell TXT2STF which text to use and which to ignore.

TXT2STF builds each item, category, and note in a separate buffer in memory. These buffers are special portions of the computer's memory that only TXT2STF can use. As it builds items, categories, and notes, TXT2STF maintains

- A single item buffer
- A single note buffer
- A separate buffer for each category

Some commands can also convert subsequent lines from the text file. For example, @note can make several consecutive text lines into a note. In this case, TXT2STF reads each text line into memory, one after the other, and converts it as instructed by the command. When the command finishes converting text lines, TXT2STF continues to the next command in the statement.

Note If a *command* reads additional lines from the text file, these additional text lines are not searched and compared to other patterns in the definition file as described in Steps 1 and 2.

4. Proceeds to the next unprocessed line in the text file

   In general, depending on the commands you use, the next line in the text file is the line immediately after the line most recently copied into memory. When it locates the next line in the text file, TXT2STF returns to Step 1 above.

TXT2STF repeats these steps until it has processed the last line from the text file.

Figure 2-3 illustrates the preceding process. The first operation that TXT2STF performs on any text line is to copy the line into memory. TXT2STF never alters the original text file.

For more information about how commands influence which line in the text file is the next line to be processed, see Chapter 5. For information about how to run TXT2STF, see Chapter 6.

TXT2STF adds to the end of the appropriate buffer each time it executes a command that identifies text to be used as item, category, or note text.

An @end command causes TXT2STF to bundle together the contents of all buffers and then copy the bundled information to the end of the structured file. If the item buffer contains text, TXT2STF creates an item. If the note buffer or any of the category buffers contain text, TXT2STF adds them (it adds the note to the item and assigns the item to the categories).

If the item buffer is *empty*, TXT2STF can create

- One or more independent categories (not assigned to items)
- An independent note (assigned to a blank item on import)

For example, if the buffers contain two categories and an item, TXT2STF adds the item to the structured file, and adds the categories as assigned to the item. If the buffers contain three categories but no item, TXT2STF creates three categories in the structured file.

When it copies information to the structured file, TXT2STF inserts tags in the converted text that tell Agenda how to import the structured information, and how to relate various elements. For example, tags indicate where each new item starts, and specify the category values and notes that belong to each item. (See Appendix B.)

When TXT2STF terminates, it clears all buffers. It only copies the buffers to the structured file if an @end is executed before TXT2STF terminates.

In Figure 2-4, TXT2STF adds converted text to buffers in response to commands in the definition file. The @custom_category command causes TXT2STF to put a category value in a buffer. Then, TXT2STF puts item text in a buffer in response to the @item2 command. When TXT2STF executes the @note command in a later statement, it puts note text in a buffer. When it executes the @end command, TXT2STF copies categories, items, and notes to the structured file.

1. TXT2STF converts the current line of text into a category and an item by executing commands in the statement that matches the current line.

2. TXT2STF places the category in a category buffer and the item in the item buffer.

3. TXT2STF copies the current contents of the buffers to the structured file when it executes @end.

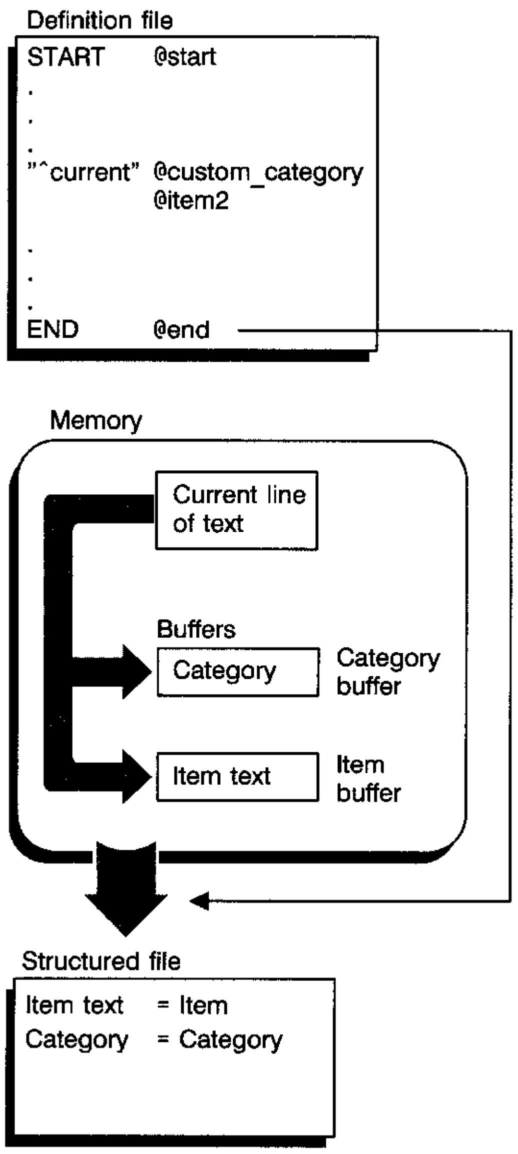


Figure 2-4 *TXT2STF uses buffers to store text*

A text file can contain more than one piece of information to be converted. For example, a text file can contain several electronic mail messages, each one to be converted to a new item. You must make sure TXT2STF executes an @end command for each item to be added to the structured file. This means you must use @end to copy TXT2STF buffers to the structured file when TXT2STF reaches the end of each occurrence of the information type (for example, each new electronic mail message).

Chapter 3
Using Definition Files

This chapter shows how you can use definition files by providing some examples. You can adapt these examples for your use, changing patterns to suit your own text files, and changing commands to convert text as appropriate for your needs.

About this Chapter

This chapter presents detailed examples of how to use definition files to

- Convert electronic mail messages
- Convert a table of information

For each example, this chapter describes how text in the text file is converted and presents a sample definition file that accomplishes this conversion. Each example concludes by showing you how the sample converted information looks after you import it to Agenda.

These examples give you a starting point for creating your own definition files. To become familiar with how definition files work, it is a good idea to read through both examples. They help prepare you for writing your own definition files.

When TXT2STF converts a text file, it puts the converted information into a structured file, which is the file you actually import into Agenda. Because reading through structured files requires some practice, this chapter omits the structured files from the examples. This helps you focus on definition files. When you become more familiar with definition files, consult Appendix B for information about structured files. Appendix B contains the structured files for the examples in this chapter.

Converting an Electronic Mail Message

The example described below shows one way to convert a text file that contains an electronic mail message. In this example, you'll see the original text file to be converted, the definition file that converts it, and an Agenda view displaying the converted information after it is imported.

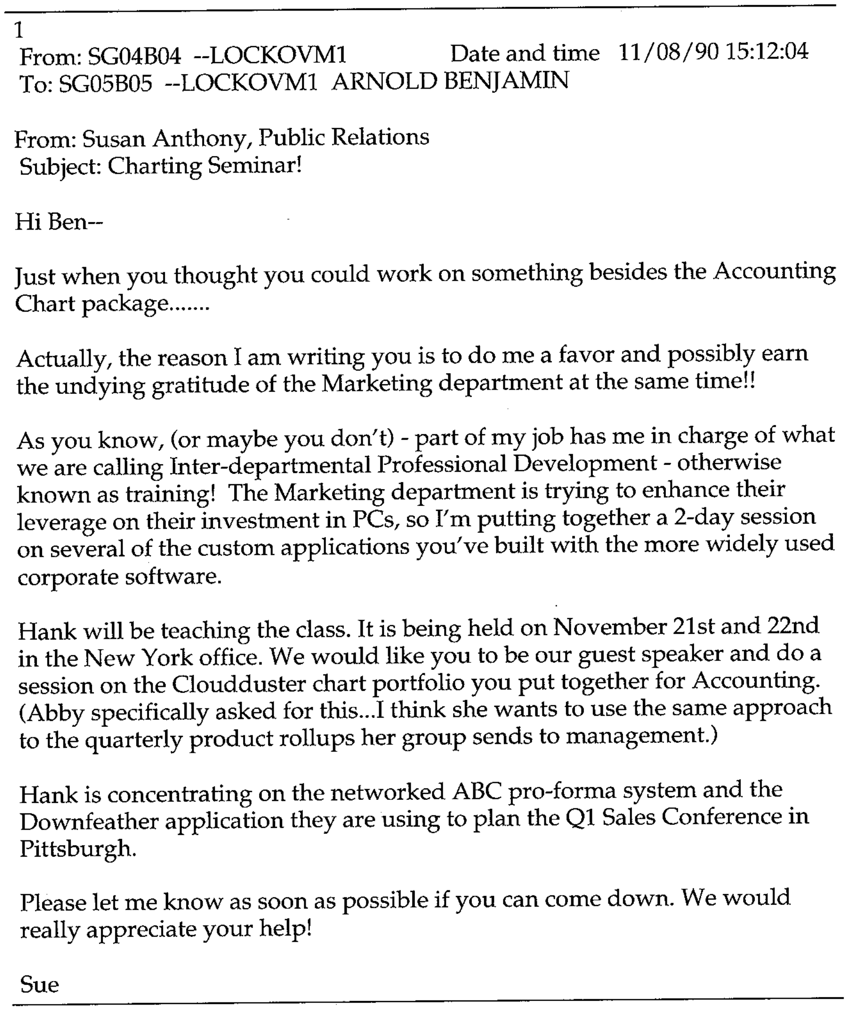
Sample Text File

Figure 3-1 shows the text file to be converted. This text file contains a generic example of a PROFS® mail message after it is exported to an ASCII text file.

In this example, the electronic mail message will be converted so that

- The subject of the message (in this example, Charting Seminar!) becomes an item
- The date and time at the top of the message (11/08/90 15:12:04) become an Entry date
- The sender name (Susan Anthony) becomes a category, with the sender's department (Public Relations) as a category note

Also, the original text file containing the entire electronic mail message is attached to the item as a note file. You can do this because TXT2STF converts and imports information from text files *without* altering the original text file. When you look at the note in Agenda, you see the actual contents of the original text file, since it is used as a note file. If you make any changes to the note, Agenda saves the changes in the external note file. For more information about note files, see Chapter 12 in the *User's Guide*.



1
From: SG04B04 --LOCKOVM1 Date and time 11/08/90 15:12:04
To: SG05B05 --LOCKOVM1 ARNOLD BENJAMIN

From: Susan Anthony, Public Relations
Subject: Charting Seminar!

Hi Ben--

Just when you thought you could work on something besides the Accounting Chart package.......

Actually, the reason I am writing you is to do me a favor and possibly earn the undying gratitude of the Marketing department at the same time!!

As you know, (or maybe you don't) - part of my job has me in charge of what we are calling Inter-departmental Professional Development - otherwise known as training! The Marketing department is trying to enhance their leverage on their investment in PCs, so I'm putting together a 2-day session on several of the custom applications you've built with the more widely used corporate software.

Hank will be teaching the class. It is being held on November 21st and 22nd in the New York office. We would like you to be our guest speaker and do a session on the Cloudduster chart portfolio you put together for Accounting. (Abby specifically asked for this...I think she wants to use the same approach to the quarterly product rollups her group sends to management.)

Hank is concentrating on the networked ABC pro-forma system and the Downfeather application they are using to plan the Q1 Sales Conference in Pittsburgh.

Please let me know as soon as possible if you can come down. We would really appreciate your help!

Sue

Figure 3-1 *Sample electronic mail text file*

Sample Definition File

Figure 3-2 shows sample definition file that converts the sample text file shown in Figure 3-1.

```
#definition file for electronic mail

"^ *From\:.*Date"  @start()
                   @skip("Date and time",1)
                   @date(E)

"^ *From\:"        @skip("\: ",1)
                   @set("save2")
                   @trim(", ¦\(","$",)
                   @category("$","From",)
                   @reset(save2)
                   @trim("^",",  ¦\(",)
                   @category_note("^")

"^ *Subject\:"     @skip("\: ",1)
                   @item("^",80,N)
                   @note_file(FILENAME)
                   @skip_lines(EOF)
                   @end()
```

Figure 3-2 ***Definition file for converting the sample electronic mail text file***

This definition file contains three statements:

- The first statement creates the Entry date.
- The second statement creates a category using the sender name and category note using the sender's department.
- The third statement creates the item and attaches the original text file as an item note.

When the resulting structured file is imported into Agenda, the item is assigned to the Entry date and sender name (Susan Anthony) categories.

The sample definition file in Figure 3-2 is tailored to work with the sample text file shown in Figure 3-1 and might not work with all PROFS messages. For example, differences in the header information for the message or in how messages are formatted might require changes in how the definition file matches and converts text lines.

The sample definition file in Figure 3-2 also assumes that each file to be processed contains a single electronic mail message, as in the sample text file. To convert a text file containing multiple electronic mail messages, you need to make some changes to the sample definition file. For example, you need to change @note so it specifies something other than EOF for the end-of-note string. See "Creating More Than One Item" in Chapter 7.

Creating the Entry date

The first statement in the sample definition file creates an Entry date. This statement begins with the pattern "^ *From\:.*Date", which matches a text line whose first word is From: *and* that contains the word Date somewhere later in the same text line. To specify this, the pattern

- Starts with a caret (^), a special match-control character that specifies that the pattern only matches a string that starts a text line
- Includes a space ( ) followed by an asterisk (*)

  The asterisk (*) is a match-control character that tells TXT2STF to match the immediately preceding character any number of times that it occurs in a row. The combination of space ( ) and asterisk (*) specifies that the text line can start with any number of space characters.
- Includes From\:

  This tells TXT2STF to match the word From followed by a colon (:). The backslash (\) tells TXT2STF that the colon (:) is a regular text character (otherwise, the colon (:) starts a special pattern condition).
- Includes .*

  The period (.) is a match-control character that matches any text character, including the space character. The combination of period (.) and asterisk (*) tells TXT2STF to match any number of text characters.
- Includes Date

  This tells TXT2STF to match the word Date. Because the pattern concludes with this specification, the entire pattern matches a text line that starts with From and ends with Date.

For more information about patterns, see Chapter 4.

A pattern and its associated commands together form a single statement. When TXT2STF finds a text line that matches pattern "^ *From\:.*Date", TXT2STF executes the commands in the statement:

- @start reserves memory that TXT2STF can use while converting a text file into items, categories, and notes.
- @skip removes the first portion of the text line, leaving the date and time information in the text line
- @date makes the remaining current text line into an Entry date (as specified by the uppercase E in @date)

  When Agenda imports the text line, it eliminates extra words, and retains only the date and time information as the Entry date.

For details about definition file commands and the arguments specified in them, see Chapter 5.

Creating the sender name category

The second statement in definition file creates a category from the electronic mail sender name (Susan Anthony). This statement begins with the pattern "^ *From\:", which matches any text line that starts with any number of space characters, followed by the word From, followed by a colon (:).

When TXT2STF finds a text line that matches pattern "^ *From\:", TXT2STF executes the commands in the statement:

- @skip deletes the string From:, including the colon (:) from the text line.

  To do this, @skip divides the text line into separate values, using the colon character followed by a space (\:) as the dividing string. Then @skip deletes the first value from the field, which is everything on the line through the word From, followed by a colon (:), followed by a space character.
- @set puts the current contents of the text line (after From: is deleted) into a variable named save2.
- @trim deletes the words Public Relations from the text line, starting with the comma (,) and continuing to the end of the text line.

  To do this, @trim searches the current text line for either a comma (,) or an open (lefthand) parenthesis. If it finds either of these characters, @trim deletes the character and everything after it from the text line.

The pattern ",¦\(" tells @trim to search for either a comma (,) or an open (lefthand) parenthesis. The vertical bar (¦) tells @trim to search for either of the *two* characters on either side of the vertical bar (¦). In this example, @trim searches for either the comma (,), which is on the left side of the vertical bar (¦) *or* the open parenthesis, which is to the right of the vertical bar (¦). (The backslash (\) tells TXT2STF to search for the open parenthesis as a regular character and to ignore its special meaning in patterns.)

- @category makes the current text line (after the end is deleted) into a child category of the From category.

  The dollar sign ($) is the end-of-line match-control character. Since the dollar sign ($) is specified as a pattern, @category makes the entire current line through the end of the line into a category. The second argument in @category is "From", which is the name of the parent category.

- @reset replaces the current text line with the contents of the variable save2, which contains the text line as it existed when @set stored the text line in save2, above.

- @trim deletes the name Susan Anthony and the comma (,) from the text line.

  This @trim is similar to the @trim discussed above. The current @trim starts from the beginning of the text line, as specified by the start-of-line character, the caret (^), and deletes everything up to the first comma *or* open (lefthand) parenthesis, as specified by the pattern ",¦\(".

- @category_note makes the current line into a category note for the sender name (Susan Anthony) category.

  The pattern "^" tells @category_note to make everything into a category note until it reaches the start of the *next* line in the text file.

The statement that starts with the pattern "^ *From" only processes the *second* line that starts with From in the electronic mail text file. This is because the pattern that starts the statement matches the text line, and also because the statement is *not* the first statement in the definition file. To see how statement order affects which statement matches a text line, let's examine how TXT2STF uses the current definition file to convert text lines in the text file:

- TXT2STF starts with the first line in the text file and compares that line to patterns in the definition file, starting from the top of the definition file and working down.

The first line in this text file contains only the number 1, and does not match any patterns in the definition file.

- TXT2STF then moves to the second line in the text file and compares that text line to definition file patterns, starting from the top of the definition file and working down.

  The second line starts with From and contains the word Date, so it matches the pattern in the first statement, which is "^ From\:.*Date". The commands in the matching statement formats the text line into an Entry date as described earlier in this section.

- TXT2STF advances line by line through the text file, comparing text lines with patterns in the definition file, until TXT2STF reaches the second line that starts with the word From.

  This is the next text line that matches a pattern. TXT2STF compares this text line to patterns in the definition file, starting from the top of the definition file and working down. Though the text line starts with From it does not contain the string Date, and so does not match the first pattern in the definition file ("^ From\:.*Date"), but matches the second pattern ("^ From\:"). TXT2STF then processes the text line using the commands in the statement with pattern "^ From\:".

If the statement with pattern "^ From\:", which matches any text line that starts with From, were the first statement in the definition pattern, both text lines that start with From would match this pattern:

- TXT2STF compares the *first* text line that starts with From with patterns in the definition file, starting from the top of the definition file and working down.

  This text line matches the first pattern in the definition file, "^ *From\:". Therefore, TXT2STF uses this first statement to process the text line, and never compares the text line with pattern "^ From\:.*Date" since it occurs later in the definition file than the matching pattern "^ From\:".

- TXT2STF compares the *second* text line with patterns in the definition file, starting from the top of the definition file and working down.

  This text line also matches pattern " *From\:", and is processed by the same definition file statement.

Creating the item

The third statement in the sample definition file creates an item. This statement begins with the pattern "^ *Subject\:", which matches any text line that starts with any number of space characters, followed by the word Subject, followed by a colon (:).

When TXT2STF finds a text line that matches pattern "^ *Subject\:", TXT2STF executes the commands in the statement:

- @skip deletes the string Subject:, including the colon (:) from the text line.

- @item creates a note from the remaining text on the text line.

  The pattern "^" tells @item to make an item that includes all text up to the start of the *next* text line. 80 specifies that the item can include a maximum of 80 characters. The N tells @item to *not* put any remaining characters from the text line (if any) in a note for the item.

  Note @item2 is an alternative to @item. The equivalent @item2 argument list would be @item2("$",80,N). For more information, see Chapter 5.

- @note_file attaches the current text file, which contains the electronic mail message, as a note file for the current item.

  FILENAME is a predefined variable that tells @note_file to use the current text file name as the note file name.

- @skip_lines moves TXT2STF to the end of the text file.

  EOF (end of file) is a predefined variable that tells @skip_lines to move to the end of the current text file.

- @end adds the categories, notes, and item created by this definition file to a structured file.

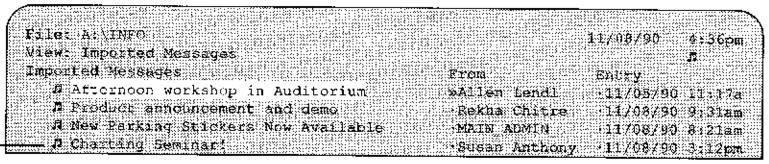
  @end is the only way to add this information to the structured file, which is the file that Agenda can import. If @end is omitted, the structured file won't contain the categories, notes, and item created in this example.

After @end, the definition file terminates since it is at the end of the text file, and there is no more text to convert.

After the Structured File Is Imported

The sample definition file in Figure 3-2 puts the item and categories it creates into a structured file. To import the item and categories into an Agenda file, you start Agenda and use the **File Transfer Import** command to import the structured file.

Figure 3-3 shows how the item and categories created by the sample definition file look after you import the structured file into Agenda. The view in Figure 3-3 shows imported messages.



```
File: A:\INFO                                            11/08/90   4:36pm
View: Imported Messages                                              ♫
Imported Messages                              From             Entry
   ♫ Afternoon workshop in Auditorium          »Allen Lendl     ·11/05/90 11:17a
   ♫ Product announcement and demo             ·Rekha Chitre    ·11/08/90 9:31am
   ♫ New Parking Stickers Now Available        ·MAIN ADMIN      ·11/08/90 8:21am
   ♫ Charting Seminar!                         ·Susan Anthony   ·11/08/90 3:12pm
```

Figure 3-3 *The imported item and categories*

The double musical note symbol (♫) before the imported item indicates that the item has an external note file attached to it. In the current example, the note file is the text file containing the original electronic mail message. If you highlight the Susan Anthony category, a musical note symbol (♪) displays in the upper righthand corner of the screen to indicate that the category also has a note. In this example, the note contains the text Public Relations.

Converting a Table of Information

The example described below shows one way to convert information that is arranged in a table format. This example converts a text file that contains a report derived from Lotus CD/Corporate®, a member of the Lotus One Source® family of CD-ROM-based information products. CD/Corporate provides business news, statistics, and financial information through monthly CD-ROMs and online updates.

The report shown in this example was generated and saved (transferred to an ASCII text file by using CD/Corporate). After saving a CD/Corporate report in an ASCII text file, you can use TXT2STF to process it.

The strategy for converting the table shown in the following example applies to *any* ASCII text file that contains information in a table. For example, you could apply this strategy to convert a table of information from a Lotus 1-2-3 worksheet that has been printed to a text file. Keep in mind, however, that you need to change the definition file shown in this chapter to work with your table and your Agenda file requirements.

Note Remember when using TXT2STF to convert tables (as well as any other information) is that the information *must* be in a text file (ASCII format). You cannot directly convert a CD/Corporate report or a 1-2-3 worksheet because they are not in ASCII format. You must first create an ASCII text version. For CD/Corporate, transfer the report to a text file. For a 1-2-3 worksheet, print the spreadsheet to a text file.

Sample Text File

The text file for the current example contains information from the Lotus CD/Corporate Industry Participants report, which is standard CD/Corporate output. This report lists all companies assigned to a particular industry grouping, ranked according to sales.

Figure 3-4 shows the beginning of the sample report derived from CD/Corporate. The report has already been saved in a text file, and so is already in ASCII format and ready to be converted. The table in this report lists companies in the Food Processing industry. The original report is long, so Figure 3-4 shows only a portion of the report.

The sample table in Figure 3-4 will be converted so that

- The text on the Industry line (Food Processing) is added as a child category under the category Industry.
- The table headings (Rank, Company, FYE, Sales, and Income) each become categories.
- Each company name (for example, Sara Lee Corp) becomes an item and is assigned to the Company category.
- Each company name is assigned to the Rank, FYE, Sales, and Income categories by the corresponding date and numeric values listed for the company in the table.

```
                 INDUSTRY PARTICIPANTS
                 =====================

 Industry:  Food Processing

Rank  Company                        FYE       Sales        Income
1   Sara Lee Corp                    7/01/89   11,717,678   410,492
2   Conagra Inc                      5/28/89   11,340,414   197,878
3   IBP Inc                          12/31/88  9,066,101    62,328
4   Archer Daniels Midland Co        6/30/89   7,928,836    424,673
5   Borden Inc                       12/31/88  7,243,526    311,882
6   Ralston Purina Co                9/30/88   5,875,900    387,800
7   H J Heinz Co                     5/03/89   5,800,877    440,230
8   Quaker Oats Co                   6/30/89   5,724,200    203,000
9   Campbell Soup Co                 7/30/89   5,672,100    13,100
10  General Mills Inc                5/28/89   5,620,600    414,300
11  CPC International Inc            12/31/88  4,700,000    289,100
12  Kellogg Co                       12/31/88  4,348,800    480,400
13  Beatrice Co                      2/28/89   4,066,000    325,000
14  Swift Independent Packing Co     10/31/87  3,692,147    -18,437
15  Whitman Corp                     12/31/88  3,582,500    233,500
```

Figure 3-4 ***Sample text file containing information in a table***

The table shown in Figure 3-4 is slightly different from the original table. In the original table, the column heads spanned two lines rather than one, and the heads were separated from information by a line of dashes (-----). For example, the beginning of the table looked like this:

```
                                        Corporate   Net
Rank  Company                   FYE      Sales      Income
-----------------------------------------------------------------
1   Sara Lee Corp               7/01/89  11,717,678  410,492
```

To make the table easier to process:

- The two-line headings, Corporate Sales and Net Income, were shortened so that all headings could fit on a single line.
- The line of dashes (-----) that separated column headings from the data was deleted.

In addition, an extra space was added between the Rank and Company heads, so that each column head is separated from the next column head by at least *two* space characters. All of these changes make it easier to write a @table command to process the table.

Sample Definition File

Figure 3-5 shows the sample definition file developed for the sample CD/Corporate text file in Figure 3-4. As is typical of definition files, this definition file is tailored to work with the sample text file in Figure 3-4. This definition file, however can be used for any Industry Participants report extracted from CD/Corporate, put into a text file, and modified as described in the previous section.

```
#Table-reading DEF file sample for CD/Corporate participant table

"Industry\:"  @start()
              @skip("\: ",1)
              @category("$","Industry",)
              @end()
              @skip_lines(1)

#@table starts in the left so the argument list stays on one line
# (required for all commands)
#
@table("  +",EOF,1,"unindexed",2,"item",3,"date",4,"numeric",5,"numeric")
```

Figure 3-5 *Definition file for converting the sample table*

The sample definition file in Figure 3-5 contains a single statement that starts with the pattern "Industry\:". When TXT2STF finds a text line that contains the word Industry followed by a colon (:), it executes the commands in the statement.

The statement first executes a @start to reserve memory for TXT2STF to use while converting the text file. Next, remaining commands process the sample text file, starting with the text line that contains the word Industry.

Commands in the statement

- Create a category from the industry (in this example, Food Processing) listed in the Industry text line
- Process each row in the table

Creating the Food Processing category

The definition file statement creates a Food Processing category as a child of the category Industry. The statement executes the following commands:

- @skip deletes the word Industry and the colon (:) that follows it from the text line.

To do this, @skip divides the text line into separate values, using the colon (:) followed by a space ( ) as the dividing string. The backslash (\) before the colon (:) tells @skip to use the colon (:) as a regular text character (otherwise the colon (:) starts a special pattern condition). Then @skip deletes the first value from the field, which is everything on the line through the colon (:).

- @category makes the current text line (after the word Industry and the colon (:) are deleted) into a child category of Industry.

  The pattern "$" tells @category to make the entire current line through the end of the line into a category. The second argument in @category is "Industry", which is the name of the parent category.

- @end saves the category to the structured file.

  It is important to execute an @end before starting to process a table since the first thing the @table command does is clear the memory where items and categories are saved prior to being copied to the structured file. If this @end is omitted, the structured file won't contain the Food Processing category.

Since there are unprocessed text lines in the text file after the Industry line, TXT2STF continues processing the text file after executing @end.

Processing the table

In order to process the table in Figure 3-4 and create items and categories, the definition file statement in Figure 3-5 executes the following commands:

- @skip_lines skips over the blank line between the Industry line and the table.

- @table processes each line in the table, creating categories from the first line in the table and creating items from remaining lines.

@table is a powerful command. The single @table command in Figure 3-5:

- Processes the row containing table headings to create categories, making Rank an unindexed category, Company a standard category, FYE a date category, Sales a numeric category, and Income a numeric category

- Processes each line in the table, creating an item from the company name and making date and numeric values in the same rows into category assignments for the item

When the resulting structured file is imported into Agenda, each item is assigned to the Company category and is also assigned by date/numeric value to the Rank, FYE, Sales, and Income categories.

@table converts a table one line at a time. @table assumes that the first line in a table provides header or label information for each column, and so *always* starts by making text in the first table line into categories. Then @table converts the remaining text lines, creating an item and related categories from each text line. To do this, @table includes the following arguments:

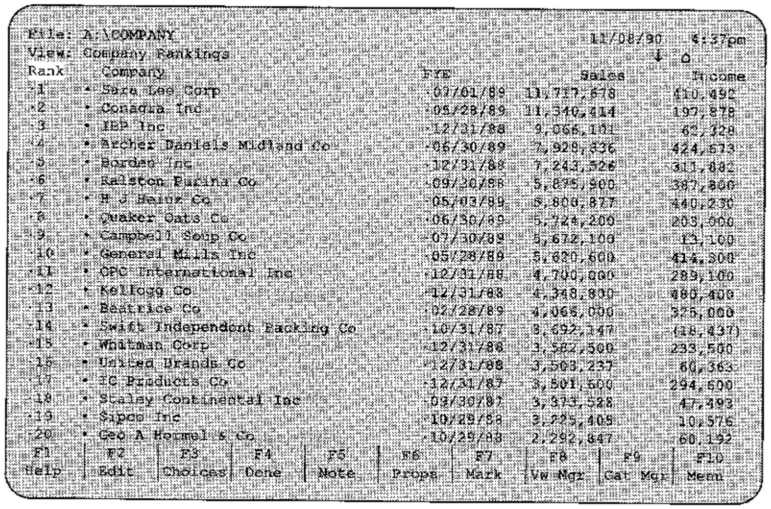
Argument	*What it specifies*
" +"	The pattern that @table uses to divide each text line into table columns. This pattern tells @table to end each column when it finds *two* or more spaces in the text line. This is why an extra space was added between the original Rank and Company headings (described earlier in this chapter).
EOF	The end of the table. @table finishes processing text lines when it reaches the end of the text file (EOF stands for end-of-file).
1,"unindexed"	How @table converts text in the *first* table column. In this case, the first column contains information for an unindexed category (Rank).
2,"item"	How @table converts text in the *second* table column. In this case, the second column contains items.
3,"date"	How @table converts text in the *third* table column. In this case, the third column contains information for a date category (FYE).
4,"numeric"	How @table converts text in the *fourth* table column. In this case, the fourth column contains information for a numeric category (Sales).
5,"numeric"	How @table converts text in the *fifth* table column. In this case, the fifth column contains information for a numeric category (Income).

When it starts processing each text line, @table internally executes @start to clear the memory where @table adds items and categories as it creates them. After it processes each text line, @table internally executes @end to add the current item and category information to the structured file. Then @table continues to the next line in the table.

After the Structured File Is Imported

The definition file puts the items and categories it creates from the table of food processing company information into a structured file. To import the items and categories into an Agenda file, you start Agenda and use the **File Transfer Import** command to import the structured file.

Figure 3-6 shows how the items and categories converted in this example look after you imported the structured file into Agenda. The view shown in Figure 3-6 displays the names of companies, their relative ranking in the industry, their fiscal year end dates, and their sales and income figures.



```
File: A:\COMPANY                                        11/08/90   4:37pm
View: Company Rankings                                          ↓   ⌂
Rank    Company                             FYE            Sales       Income
·1    • Sara Lee Corp                      ·07/01/89  11,717,678      410,492
·2    • Conagra Inc                        ·05/28/89  11,340,414      197,878
·3    • IBP Inc                            ·12/31/88   9,066,101       62,328
·4    • Archer Daniels Midland Co          ·06/30/89   7,928,336      424,673
·5    • Borden Inc                         ·12/31/88   7,243,526      311,082
·6    • Ralston Purina Co                  ·09/30/88   5,875,900      387,800
·7    • H J Heinz Co                       ·05/03/89   5,800,877      440,230
·8    • Quaker Oats Co                     ·06/30/89   5,724,200      203,000
·9    • Campbell Soup Co                   ·07/30/89   5,672,100       13,100
·10   • General Mills Inc                  ·05/28/89   5,620,600      414,300
·11   • CPC International Inc              ·12/31/88   4,700,000      289,100
·12   • Kellogg Co                         ·12/31/88   4,348,800      480,400
·13   • Beatrice Co                        ·02/28/89   4,066,000      325,000
·14   • Swift Independent Packing Co       ·10/31/87   3,692,147      (18,437)
·15   • Whitman Corp                       ·12/31/88   3,582,500      233,500
·16   • United Brands Co                   ·12/31/88   3,503,237       60,363
·17   • IC Products Co                     ·12/31/87   3,501,600      294,600
·18   • Staley Continental Inc             ·09/30/87   3,373,528       47,493
·19   • Sipco Inc                          ·10/29/88   3,225,405       10,576
·20   • Geo A Hormel & Co                  ·10/29/88   2,292,847       60,192
 F1   |   F2   |   F3    |  F4   |  F5   |  F6   |  F7  |   F8   |   F9    |  F10
Help  |  Edit  | Choices | Done  | Note  | Props | Mark | Vw Mgr | Cat Mgr | Menu
```

Figure 3-6 *The imported items and categories*

Chapter 4 Patterns

A pattern describes a string of text characters. The pattern specifies, character by character, the text string to match. The pattern also can include additional information about the text string, such as whether it starts or ends the text line.

In this Chapter

This chapter gives you information about how to use patterns by

- Describing how to use patterns in definition files
- Describing special patterns START and END that you can use with TXT2STF
- Detailing the match-control characters you can use when constructing patterns
- Detailing how TXT2STF uses patterns to match text strings in text lines

About Patterns

Patterns describe text strings in a text file. Whenever you need to match a text string in a text file, you specify a pattern in the definition file.

You use patterns in the following ways:

- You begin each definition file statement with a pattern that describes the text line where the statement begins converting text.

When TXT2STF finds a text line that matches the pattern, TXT2STF converts the text line by executing the commands in the statement.

- You also can include patterns as arguments in many definition file commands.

  For example, you can include a pattern in @replace to identify the text string to be replaced with another string. When TXT2STF finds the identified string in the text line, it replaces it with the substitute string also specified by @replace.

A pattern describes a text string. To create a pattern, you can

- Type the text string itself, surrounded by a pair of double quotation marks (" "); you *must* use double quotation marks (" ") and not single quotation marks (' ').

  For example, to search for the text string Subject, use the pattern "Subject". If the text string includes spaces, make sure the pattern specifies each space in the correct position. Do not add spaces where the text line has no spaces.

- Include match-control characters in the pattern.

  Match-control characters help provide extra information about the text string in a compact form. For example, match-control characters let you specify whether the text string is the first or last text on a line. You also can use match-control characters to specify a more generic pattern, such as a pattern that matches any grouping of four letter characters in a row.

  For more information, see "Match-Control Characters" later in this chapter.

Note TXT2STF ignores case when matching letters of the alphabet, unless you specify the /C option when you run TXT2STF. (See Chapter 6.)

Special Patterns

TXT2STF provides two special patterns, START and END, that identify the beginning and the end of a text file. These patterns can begin definition file statements. They *cannot* be included as patterns in commands. Do not enclose the patterns START and END in quotation marks.

Note Do not confuse these patterns with @start and @end, which are commands. (See Chapter 5.) You can, however, include @start in a statement that begins with START and @end in a statement that begins with END.

START

START causes a statement to execute only once, when TXT2STF begins processing the text file. You use START when the statement includes commands you want TXT2STF to run only when it begins processing the text file.

You can also use START to write a statement that specifies the format of dates in the structured file. Agenda uses this information when it imports the structured file. Use START as the pattern and put the @date_format command in the statement, followed by an @end command. @date_format specifies the format of numeric dates in the structured file (for example, MM/DD/YY format).

END

END causes a statement to execute only once, when TXT2STF finishes processing the text file. You use END when the statement includes commands you want TXT2STF to execute only when it finishes processing the text file.

Using START and END

You might use START and END if you have a group of files that don't all start and/or end in the same way. Use START as the pattern in the statement that contains the first @start command. Use END as the pattern in the statement that contains the final @end command.

You should *not* use END as the only pattern associated with the @end command if your input text file has more than one input group, for example, a number of electronic mail messages, each one of which should become a separate item. You need an @end command to copy each item to the structured file; if @end is not executed until TXT2STF closes the file, only the last item is copied to the structured file.

START and END do not cause TXT2STF to read text into memory, so you should only use them with commands that do not make use of the current text line, such as @start, @end, and @date_format.

Note You should *not* use these patterns in commands that convert text into items, categories, or notes, as those commands require that you have a text line in memory to be converted.

Match-Control Characters

Match-control characters let you write patterns that match a varied set of text strings in the text line by letting you specify additional information about the text strings to be matched.

For example, to match lines that start with the word Subject, begin the pattern with the caret (^) match-control character. Caret (^) tells TXT2STF to find the specified text string only when it occurs at the beginning of a line, resulting in the pattern "^Subject". To search for a line starting with *either* Subject or Re, you could use the pattern "^(Subject ¦ Re)", which includes the vertical bar (¦) as the "or" character.

The following table summarizes the available patterns. Each match-control character is discussed in greater detail later in this chapter.

Note These match-control characters are different from the match-control characters used in Agenda for text conditions.

Match-Control Character	*Description*
Asterisk (*)	Matches zero or more occurrences of the preceding pattern character
Backslash (\)	Specifies that the next character in a pattern is a text character and not a match-control character
Brackets ([ ])	Specifies a range or group of characters, and matches any single text character specified in the range or group
Caret (^)	As the first character in a pattern, causes the pattern to match the corresponding text only when it occurs at the beginning of a text line As the first character in a pair of brackets ([ ]), caret (^) negates the range or group specified in the brackets
Colon a (:a)	Matches any single uppercase or lowercase alphabetic text character
Colon d (:d)	Matches any single numeric text character from 0 through 9
Colon n (:n)	Matches any single alphabetic or numeric text character

continued

Match-Control Character	*Description*
Colon space (:)	Matches a single space or control character in the text file
Dollar sign ($)	As the last character of a pattern, causes the pattern to match the corresponding text string only when it occurs at the end of a text line
Parentheses ( )	Groups pattern characters into a string so you can apply a match-control character to the string
Period (.)	Matches any single character in the text file (except a carriage return or linefeed character)
Plus (+)	Matches one or more occurrences of the immediately preceding pattern character
Tilde (~)	Matches any text string that is *not* the pattern
Vertical bar (¦)	Matches one pattern *or* another (performs a logical OR)

Asterisk (*)

The asterisk (*) matches zero or more occurrences of the preceding pattern character. For example, "X*" matches a zero, one, or any number of uppercase X characters in a row.

To match zero or more occurrences of a string, enclose the entire string in parentheses, followed by the asterisk (*).

Tip If you need to match at least *one* text character, use the plus (+) and not the asterisk (*).

Examples

Sample pattern	*Matches in the text file*
`"ab*"`	a ab abbbb It does not match zb or fbbb
`"Hey(, Terry)*"`	Hey Hey, Terry Hey, Terry, Terry, Terry

For more examples of using the asterisk (*), see the section about the period (.) later in this chapter. For more information about how TXT2STF matches the asterisk (*), see "How TXT2STF Compares Patterns with Strings" later in this chapter.

Backslash (\)

The backslash (\) specifies that the immediately-following character is a text character and not a match-control character. You use the backslash (\) to match a character that TXT2STF uses as a match-control character, such as the asterisk (*) or dollar sign ($).

For example, to match the string "Amount in US$", you need to match the dollar sign ($) as a text character. To do this, use the following pattern: "Amount in US\$". The backslash (\) in this pattern tells TXT2STF to match the dollar sign ($) as a text character, and to ignore its special meaning as the end-of-line match-control character.

Use backslash (\) before each of these special characters to match them as literal text characters:

```
* . ~ : ] [ ^ $ \ ! ¦ ) ( # @ "
```

The backslash (\) applies to only one character at a time. To match a string of special characters as literal text, you must precede *each* of the special characters with a backslash (\).

Tip The backslash (\) cannot be used for control characters (ASCII decimal 1 through 32). To match these characters, use the combination of colon (:) and space ( ), described later in this chapter.

Examples

Sample pattern	*Matches in the text file*
`"\$\$"`	Two consecutive dollar signs ($$) in the text file
`"Col1\\Col2"`	The string Col1\Col2

Brackets ([ ])

Brackets ([ ]) define a range or group of characters, and match any *single* character in that range or group. You can include range and group specifications in the same pair of brackets ([ ]).

You can specify any text characters (characters *above* ASCII 031), including the space character ( ).

Note TXT2STF ignores the /C option when matching characters specified in brackets ([ ]), and matches those characters exactly as specified in the range or group.

Ranges

A range is a contiguous set of characters or values, for example 1 through 5. You specify a range to match any one of the characters in the range.

To specify a range, identify the lowest and highest values in the range, separating these values with a hyphen (-). For example, [1-5] matches any character in the range from 1 through 5. [a-z] matches any single lowercase character in the range from a through z.

The characters that define the range must be specified with the smaller ASCII decimal value first and the highest ASCII decimal value last. Use the standard character representation (for example, A or 9) and not the equivalent ASCII decimal representation.

Examples

Sample pattern	*Matches in the text file*
`"[A-Z]"`	Any single uppercase letter
`"[a-zA-Z]"`	Any single character in the alphabet, including both uppercase and lowercase values
`"[0-9]"`	Any single numeric character from 0 to 9, including 0 and 9
`"[a-c5-9]"`	Any single character in the alphabetic range of lowercase a through c or in the numeric range 5 through 9

Groups

A group is a list of characters, each of which is valid. You specify a group of characters to match any one of a specific list of characters.

Examples

Sample pattern	*Matches in the text file*
`"[SML]"`	Any of the uppercase characters S, M, or L
`"[RGB]"`	Any of the uppercase characters R, G, or B
`"[a-z123]"`	Any of the lowercase characters specified in the range a through z, or any of the numbers 1, 2, or 3

Special character guidelines

A match-control character that immediately *follows* a range or group applies to the entire range or group. For example, [A-Z]+ matches one or more characters in the range A through Z.

When included *inside* of brackets ([ ])

- Match-control characters lose their special meaning when included in range or group specifications. For example, the asterisk (*), backslash (\), period (.), and left bracket ([) represent actual characters when included in a pair of brackets ([ ]).
- The caret (^) negates the remaining specification when included as the first character inside the brackets ([ ]). The pattern matches any character *except* a character specified in the range or group. The pattern does not, however, match linefeed (ASCII decimal 10) or carriage return (ASCII decimal 13).
- The hyphen (-) indicates a range specification *unless* it occurs as the first character after the left bracket ([ ), or after the caret (^) in negative patterns. In these two cases, the hyphen (-) is an actual character and not a special character.
- The right bracket ( ]) ends the range or group specification *unless* it occurs as the first character after the left bracket ([ ), or after the caret (^) in negative patterns. In these two cases, the right bracket ( ]) is an actual character and not a special character.

Examples

Sample pattern	*Matches in the text file*
`"[a-zA-Z]+"`	One or more alphabetic characters, uppercase or lowercase, in a row
`"[^0-5]"`	Any character that is not in the range 0 through 5; for example, it matches A or 6, but does not match 5
`"[^XYZ]"`	Any character that is not an uppercase X, Y, or Z; for example, it matches A, 1, and lowercase x, y, or z
`"[]1-7]"`	Either the right bracket ( ]) or any character in the range 1 through 7
`"[*-/]"`	Any character in the range from the asterisk (*), which is ASCII decimal 042, through the slash (/), which is ASCII decimal 47

Caret (^)

The caret (^), when included as the first character of a pattern, causes the pattern to match the corresponding text string only when it occurs at the beginning of a text line. The caret (^) *must* be the first character in the pattern.

To match the caret (^) as a normal text character, precede it with a backslash (\).

When the caret (^) is the first character in a pair of brackets ([ ]), as in "[^1-9], the caret (^) negates the range or group specified in the brackets ([ ]). For more information, see "Brackets ([ ])" earlier in this chapter.

Example

Sample pattern	*Matches in the text file*
`"^The"`	Any line that begins with the string The It does not match either of these lines: 1. The However, the...
`"^(To\:¦From\:)`	Any line that begins with either To: or From:

For more information about how TXT2STF matches the caret (^), see "How TXT2STF Compares Patterns with Strings" later in this chapter.

Colon a (:a)

The combination :a matches any single uppercase or lowercase alphabetic character.

Example

Sample pattern	*Matches in the text file*
`":a"`	W z B or any other single alphabetic character

Note The :a match-control character is not case sensitive even if you use the /C option when you run TXT2STF. (See Chapter 6.) To match only uppercase or only lowercase characters, see "Brackets ([ ])" earlier in this chapter.

Colon d (:d)

The combination :d matches any single numeric character from 0 to 9.

Examples

Sample pattern	*Matches in the text file*
`":d:d:d-:d:d:d:d"`	555-8500 555-1212 or any other seven-digit phone number with a hyphen in the middle
`"Order\: :d:d:d"`	The word Order, followed by a colon (:), followed by any three-digit order number

Colon n (:n)

The combination :n matches any single uppercase or lowercase alphabetic character or numeric character.

Example

Sample pattern	*Matches in the text file*
`":n:n:n"`	b5h 897 xxx or any other combination of three alphabetic and/or numeric characters

Colon Space (:)

The combination of colon (:) followed by one space ( ) matches one space or control character.

Specifically, it matches any character from ASCII decimal 1 through 32. This list includes tab (ASCII decimal 9), linefeed (ASCII decimal 10), carriage return (ASCII decimal 13), and other control characters as well as the standard space ( ) character (ASCII decimal 32).

Example

Sample pattern	*Matches in the text file*
`":d: "`	Any two-character combination consisting of a single numeric character followed by one space or control character, such as tab

Dollar Sign ($)

The dollar sign ($), when included as the last character of a pattern, causes the pattern to match the corresponding text string only when it occurs at the end of a text line. The dollar sign ($) matches the actual end of a line (not the carriage return, linefeed, or alternate separator).

To use the dollar sign ($) as a normal text character, precede it with a backslash (\).

Example

Sample pattern	*Matches in the text file*
`"The end.: *$"`	Any line that ends with The end It does not match a line that ends with: The end. Display film credits.

For more information about how TXT2STF matches the dollar sign ($), see "How TXT2STF Compares Patterns with Strings" later in this chapter.

Parentheses ( )

Parentheses group characters into a string. This lets you apply a match-control character to an entire string rather than to just a single character.

Each set of parentheses in a pattern begins a new level of nesting. When you include a set of parentheses *inside* another set of parentheses, the inner parentheses define a new nesting level that is one level deeper than the level defined by the outside set. A pattern can include up to 10 nesting levels. If you end one set of parentheses before starting a second set, both parentheses are at the same level of nesting.

Examples

Sample pattern	*Matches in the text file*
`"(HLQ)*"`	Zero or more consecutive occurrences of the entire string HLQ, for example, HLQhlq
`"(:n:n:n)+"`	A line containing one or more strings of three numeric characters, for example 123
`"^(:a:a(:d:d)+)+"`	A line starting with two alphabetic characters (specified by :a:a) followed an even number of numeric characters (specified by :d:d); the string can include any number of instances of this string

Period (.)

The period (.) matches any single character, including a space character. The period (.) also matches control characters (ASCII decimal 1 through 31, *except* for linefeed (ASCII decimal 10) or carriage return (ASCII decimal 13)).

Examples

Sample pattern	*Matches in the text file*
`"..."`	123 abc {}= ...
`".*"`	Any characters up to the start of the next new line
`".*\:"`	Any characters up to and including the final colon (:) on the current line; the pattern does not match a text line if the line is missing a colon (:)

Plus (+)

The plus (+) matches one or more occurrences of the immediately-preceding character.

To match one or more occurrences of a character string, enclose the string in parentheses followed by plus (+).

Tip To match *zero* or more occurrences of a character, use the asterisk (*) and not the plus (+).

Examples

Sample pattern	*Matches in the text file*
`"b+"`	b bb bbbbbbb
`"0123+"`	0123 01233 012333333333
`"(0123)+"`	0123 01230123 0123012301230123

continued

Sample pattern	*Matches in the text file*
`".+"`	All characters up to the start of the next new line; the pattern does not match a text line if the text line lacks at least one character before a linefeed, carriage return, or alternate separator character
`".+\:"`	All characters up to and including the final colon (:) on the current line; the pattern does not match a text line if the line lacks a colon (:) preceded by at least one other character

For more information about how TXT2STF matches the plus (+), see "How TXT2STF Compares Patterns with Strings" later in this chapter.

Tilde (~)

The tilde (~) matches any string that is *not* the pattern.

Note Negative patterns typically provide less control over what the statement matches. As a general rule, use negative patterns only in text files in which the order and format of lines is very predictable.

Example

Sample pattern	*Matches in the text file*
`"^~Modem"`	Any line (or string) that does not *begin with* the word Modem
`"^~.*Modem"`	Any line (or string) that does not *contain* the word Modem

Note To make sure that the pattern does not match every line in the text file, it is important to anchor the pattern to some fixed characteristic in the string you want to match. For example, the pattern "^~Modem" matches lines that start with any string other than Modem. Without the caret (^), this pattern would also match lines that include the string Modem, since portions of this string (such as odem) are *not* the same as the string Modem, and so would match the pattern.

For more information about how TXT2STF matches the tilde (~), see "How TXT2STF Compares Patterns with Strings" later in this chapter.

Vertical Bar (¦)

The vertical bar (¦) represents a logical OR. This lets you match one pattern OR another. You separate the alternative patterns with the vertical bar (¦).

Examples

Sample pattern	*Matches in the text file*
`"a¦b"`	a OR b
`"dog¦cat"`	dog OR cat
`"[a-f]¦[q-z]"`	Any single letter between a and f OR any single letter between q and z

How TXT2STF Matches Patterns

TXT2STF compares a pattern to the characters in a text line to determine whether the text line matches the pattern. TXT2STF searches a single text line at a time. If the text line includes the entire string specified by the pattern, it matches the pattern. If the text line lacks the string, the line does not match the pattern.

When TXT2STF Matches Patterns

Patterns describe strings in your text file that you want TXT2STF to match. By matching the string, TXT2STF finds a portion of text that you want to convert.

You use patterns in definition file statements in the following ways:

- Each definition file statement begins with a pattern that identifies the text line the statement converts.

  For example, to convert a text line that includes the string Personnel Record, a definition file statement begins with the string "Personnel Record".

- You can include patterns as arguments in definition file commands.

  For example, you can specify where an item ends by including a termination pattern in the @item command.

TXT2STF matches these patterns as described below.

Patterns that begin statements

Patterns that begin statements identify which text lines the commands in the statement convert.

TXT2STF converts a text file line by line. After reading a line of the text file into memory, TXT2STF compares it with patterns in the definition file, starting with the first pattern in the definition file. The text line matches the first pattern that describes a string in the text line.

For example, the following text line matches the second pattern ("Records") since TXT2STF encounters the second pattern before the third pattern ("^Personnel"):

Personnel Records For New Employees

Patterns:
```
"^Date\:"
"Records"
"^Personnel"
```

When it matches any portion of a text line with a pattern, TXT2STF uses the statement that contains the pattern to convert the text line. The entire text line is available when TXT2STF executes the first command in the statement.

After converting a text line, TXT2STF continues to the next unconverted line in the text file, and attempts to match that line with patterns in the definition file, starting again with the first pattern in the definition file. TXT2STF never returns to a converted text line, and never uses more than one statement to convert a text line. For more information, see "How TXT2STF Processes Each Text Line" in Chapter 2.

Tip If you start TXT2STF with the /M debugging option, TXT2STF displays each text string that matches a definition file pattern. (See Chapter 7.) Using this information, you can determine which text line matches each pattern, and therefore can determine whether the correct statement converts each text line.

Patterns that are arguments in commands

You can include patterns as arguments in definition file commands. These patterns are pattern constants that specify information to be used in the command. For example, you identify the locations where @trim starts and stops trimming text by specifying a start and end pattern.

When TXT2STF executes a command that includes pattern constants, it compares the pattern to the current text line, starting with the first character in the line. By matching a string with the pattern, TXT2STF navigates to a specific portion of the line. After it locates the appropriate place in the text line, TXT2STF performs the operation specified by the command.

For example, @trim("^"," +",) deletes text from the current text line. The pattern "^" tells TXT2STF to start deleting at the beginning of the text line. The pattern " +" tells TXT2STF to keep deleting text until it reaches the end of a string consisting of one or more space characters.

Note TXT2STF does *not* display text strings that match pattern constants in commands, even if you start TXT2STF with the /M debugging option.

How TXT2STF Compares Patterns with Strings

To see if a text string matches a pattern, TXT2STF compares the characters in the pattern with the characters in the text string, one character at a time. A pattern can consist of regular text characters exclusively, or can also include match-control characters.

If the pattern contains text characters, such as the pattern "Subject", TXT2STF starts by searching the text line for the first text character in the pattern (for example, the uppercase S in "Subject"). When it finds that character, TXT2STF compares the next character in the pattern (for example, the lowercase u in "Subject") with the next character in the text string. If the characters match, TXT2STF continues comparing characters in the pattern with characters in the text line, one by one, until the entire pattern matches a complete string. If at least one text character in the string differs from the pattern, the text string does *not* match the pattern.

The TXT2STF /C option determines whether TXT2STF matches or ignores the case (uppercase or lowercase) of text characters when it matches patterns. (See Chapter 6.) The /C option is ignored, however, when matching ranges or groups specified in brackets ([ ]). TXT2STF matches those characters exactly as they are specified in the range or group.

The caret (^) and dollar sign ($)

Match-control characters affect how TXT2STF matches patterns. Specifying a caret (^) as the first character in a pattern tells TXT2STF to match the pattern only when the corresponding text string begins the text line. Specifying a dollar sign ($) as the last character in a pattern tells TXT2STF to match the pattern only when the corresponding text string occurs at the end of a text line.

The asterisk (*) and plus (+)

The asterisk (*) and plus (+) tell TXT2STF to match as many occurrences of the preceding pattern character as possible. For example, "X*" matches zero or more uppercase X characters in a row. The pattern ".*" matches a string containing zero or more characters.

When TXT2STF matches a string with a pattern that ends with either asterisk (*) or plus (+), TXT2STF matches up through the last text character that matches the pattern. If the next character in the pattern does not match the next text character after the currently matched string, TXT2STF searches *backward* through the matched string until it finds the text character that matches the pattern character.

For example, the pattern ".*\:" matches a line containing zero or more characters, and matches up through the last character in the text line. Then, TXT2STF looks to see whether the next character is a colon (:), as specified by the combination of backslash and colon (\:) in the pattern. Since TXT2STF is at the end of the text line, it cannot find a colon *after* the matched string in the text line. So, TXT2STF starts searching backward through the matched string for a colon (:).

For example, the pattern "^.*\:" matches the following sample text line up through the colon (:) that ends the word Site:

From: KGY Site: 23

To match only up through the colon (:) that ends the word From in the above sample text line, you could use the pattern "^[^:]*\:". This pattern matches a string consisting of any characters that are *not* colons (:), as specified by [^:]*. The string matches the pattern up to the first colon in the line or the end of the line, whichever comes first. TXT2STF then compares the next character in the text line with the next pattern character, which is the colon character specified by the combination of backslash and colon (\:). If the next character is a colon, the text line matches the pattern up through the first colon in the line.

The tilde (~)

The tilde (~) begins a negative pattern. That is, it tells TXT2STF to match any string that is *not* the pattern. It is important to anchor a negative pattern to a fixed characteristic of the text line (for example, caret (^) anchors the pattern to the beginning of the text line). This keeps TXT2STF from trying to match the pattern in ways that, in this case, are undesired.

To match a negative pattern, TXT2STF starts at the beginning of the text line and searches through the entire text line for the negative pattern. The negative pattern *matches* the text line if TXT2STF cannot find the negative pattern in the text line.

The negative pattern *does not match* the text line if TXT2STF finds the negative pattern in the text line. If the larger pattern begins with caret (^), TXT2STF concludes the matching process and continues to the next argument, command, or statement, as appropriate.

Note Without the caret (^) to specify where TXT2STF must begin searching the text line, the negative pattern eventually matches the text line. Each time TXT2STF finds the negative pattern in the text line, TXT2STF moves forward in the text line by one character and tries again to locate the negative pattern from its new starting location in the text line. Eventually TXT2STF moves beyond the start of the text string that matches the negative pattern. At this point, TXT2STF can no longer find the negative pattern in the text line, and so the negative pattern matches the text line.

How TXT2STF Searches for Character Classes

Each of the following match-control characters matches any single character of a specific type, range, or group:

- Colon a (:a) matches an alphabetic character.
- Colon d (:d) matches a numeric character.
- Colon n (:n) matches an alphabetic or numeric character.
- Colon space (:) matches a space or control character (ASCII decimal 1 through 32).
- A range or group specification in brackets ([ ]) matches a character specified in the range or group.

Each of these match-control characters specifies a class of characters, any one of which matches the corresponding character in a text line. For example, the colon a (:a) combination matches any single uppercase or lowercase alphabetic character.

Before TXT2STF compares patterns with text in the file, it internally expands each of these character class specifications, replacing the match-control character with the full list of characters it can match. For example, TXT2STF internally changes the combination of colon a (:a) to the full list of uppercase and lowercase alphabetic characters. Then, TXT2STF compares the text line with the pattern.

A pattern can include up to eight different character class specifications:

- Each group or range specification constitutes a separate character class specification.
- Each different colon combination, such as colon a (:a) or colon d (:d), is a separate character class specification.

  The pattern can, however, include any number of the same type of colon combination. For example, since all colon a (a:) combinations expand in the same way, TXT2STF can re-use the first expanded colon a (:a) specification for all subsequent colon a (:a) specifications in the pattern.

Examples

This pattern	*Includes these character class specifications*
`":d:d/:d:d/:d:d"`	One character class specification, for the combination colon d (:d)
`"[a-f] ¦ [q-z]"`	Two character class specifications, one for each range
`"^:a+-:d:d"`	Two character class specifications, one for the combination colon a (:a) and one for colon d (:d)
`":d:a[3-5abc][3-5abc]:d:a"`	Four character class specifications, one for :d, one for :a, and one for *each* range specification (even though they specify the same characters)

Chapter 5
Definition File Commands

Agenda provides several commands that you can include in a definition file to tell TXT2STF how to process the contents of your text file.

In this Chapter

This chapter describes

- The types of definition file commands that Agenda offers
- Definition file command syntax
- Arguments and variables used with definition file commands
- The syntax and purpose of each definition file command

This chapter also includes one or more examples of each definition file command.

About Commands

Commands tell TXT2STF how to process the contents of your text file. For example, commands in your definition file can tell TXT2STF when it processes a text file containing electronic mail to make the text following Subject: into an item and the text following Date: the When date for the item.

Each definition file statement begins with a pattern, and includes one or more commands. A pattern and all of its commands belong to a single statement. (See Chapter 2 for more information.)

TXT2STF processes a text file by comparing lines in the text file, one at a time, with the patterns that begin statements in the definition file, starting from the top of the definition file. When TXT2STF matches text line with a pattern, TXT2STF executes the commands in the statement whose pattern matches the text line. This is how TXT2STF processes the text line.

Most commands in a statement perform an action that processes the current text line. For example, @item2 can take characters from the text line and put them into the item buffer. @skip and @trim can remove text from the text line. When a command finishes processing a portion of the text line, the next command in the statement starts where the previous line left off.

So that TXT2STF can progress through a text file, a statement uses up at least one text line. When a statement finishes executing, TXT2STF gets ready to continue processing the text file by copying the next unprocessed text line into memory. Then TXT2STF searches for a definition file pattern that matches the current text line in memory.

Caution Text remaining in the current text line when a statement finishes executing is discarded when TXT2STF copies the next unprocessed text line into memory in preparation for the next statement.

Commands operate on the current text line in memory, telling TXT2STF how to convert the line and whether to save the resulting text in the item buffer, a category buffer, or the note buffer. Commands can add text to variables and advance your position in the text file. Commands also can tell TXT2STF when to transfer the contents of these buffers to the structured file. This chapter describes, for each command, the changes the command makes to the current text line, to TXT2STF buffers, to variables, to your location in the text file, and to the structured file.

When you include more than one command in a statement, you can make the statement easier to read by placing each command on a new line, immediately under the preceding command. For example, the following statement includes three commands, one under the other, beginning with @start:

```
"^Date\:"   @start()
            @skip(" +",1)
            @date(W)
```

Note In Release 1.0, you had to begin each continued command line with a backslash (\) as the first character of the new line. The backslash (\) is no longer required, but is allowed in definition files. In this chapter, partial examples where commands are shown without the corresponding pattern (or other commands) begin with a backslash (\) to indicate that the command is part of a larger statement that is not shown. For example, a backslash (\) precedes the following @start since the command is shown without a preceding pattern:

```
\    @start()
```

Types of Commands

Agenda definition file commands fall into the following groups:

- Category commands add categories; each category command adds a new category to TXT2STF category buffers.
- Control commands initialize TXT2STF item, note, and category buffers and copy converted text from the buffers to the structured file.
- Date commands add date categories and describe the format of numeric dates in the text file.
- Item commands convert text into items.
- Note commands convert text into notes for items and categories.
- Table commands convert tables into items and categories.
- Trimming commands let you delete text from a text line before you convert it.
- Variable commands create new variables, put values into variables, and initialize variables.

The following tables list Agenda definition file commands by group and briefly describe the purpose of each command. For more information about each command, see the alphabetical description of these commands later in this chapter.

Category Commands	*Description*
@category	Creates one or more categories from the text line and optionally makes them children of a specified parent category
@custom_category	Creates a category from text that you specify and optionally makes them children of a specified parent category
@date	Creates assignments to date categories
@numeric	Creates assignments to numeric categories
@unindexed	Creates assignments to unindexed categories

Control Commands	*Description*
@end	Copies the contents of the item, note, and category buffers to the structured file, and then initialize the buffers
@start	Initializes the item, note, and category buffers *without* first copying them to the structured file; prepares the buffers for use

Date Command	*Description*
@date	Creates assignments to date categories
@date_format	Describes the format of the dates in the text file

Item Commands	*Description*
@append_item	Appends a string to the end of the item currently being constructed from the text file
@item	Creates an item using text from the text file; is an alternative to @item2
@item2	Creates an item using text from the text file; is an alternative to @item

Note Commands	*Description*
@append_note	Appends a string to the end of the current note
@category_note	Creates a category note and assigns it to the category created by the most recently executed @category or @custom_category command
@category_note_file	Identifies the external note file for the current category
@make_external_note	Creates an external note file and copies text from the text file into the file; on import the note file is added as an item note
@note	Creates a note and assigns it to the current item
@note_file	Identifies the external note file for the current item

Table Command	*Description*
@table	Makes information in a table into items and categories

Trimming Commands	*Description*
@replace	Substitutes one string for another
@skip	Divides the line into values and then deletes one value from the line
@skip_lines	Skips over the number of lines you specify
@strip	Deletes one or more occurrences of a specified string from the text line
@trim	Deletes all the text between one string and another, placing the deleted text in a variable

Variable Commands	*Description*
@append	Adds a string to the end of the current contents of the variable
@equate	Stores a pattern or a text string in a variable
@grab	Divides the current text line into several values and puts each value into a variable
@reset	Replaces the current text line with the specified value
@set	Stores the current text line in the specified variable
@trim	Deletes all the text between one string and another, placing the deleted text in a variable

Command Syntax

Each definition file command consists of a keyword, which is the command name. Most commands also require you to specify one or more arguments in parentheses. Arguments provide the information that TXT2STF needs to complete the command; the what and where of a particular action.

Syntax

Figure 5-1 shows the format for definition file commands:

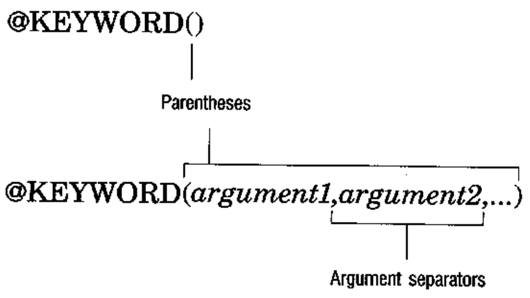


Figure 5-1 *Definition file command syntax*

When you specify a command without arguments, be sure to include the pair of parentheses at the end of the command without any intervening spaces.

Conventions

This chapter uses these conventions when showing command syntax:

- Arguments that you replace with your own information are in italic type.
- Optional arguments are enclosed in brackets ([ ]) in the syntax.

  If you omit an optional argument, include the comma (,) that precedes the argument only if the comma (,) *precedes* the brackets ([ ]) in the syntax.
- A pair of double quotation marks (" ") around an argument indicate that the quotation marks are required; you *must* use a pair of double quotation marks (" ") and not a pair of single quotation marks (' ').
- Ellipsis points (...) in commands indicate that you can specify the argument more than one time.

  Note In examples, vertical ellipses (a series of points, one above the other) indicate that lines are omitted to simplify the example, as shown below:

  ```
  "^Start_trans\:"     @start()
  .
  .
  "^QUOTE "            @item2("$")
  ```

At (@) character

Each definition file command starts with an at (@) character. Start the command name immediately after the at (@) character without an intervening space. For example, use @start(), not @ start().

Note Though definition file commands begin with the at (@) character, they differ from Lotus 1-2-3 at (@) functions, and perform different types of processing.

Keywords

The keyword in the command is like an action verb. It tells TXT2STF what action to perform.

Arguments

Arguments provide the information that Agenda needs to complete the command. For example, you specify the name of a date category, such as When, as an argument in a @date command to assign a date to that category.

About Arguments

Arguments in command syntax stand for specific types of information that you can include in the command.

Follow these guidelines when including arguments in commands:

- Enclose all command arguments in the pair of parentheses that follows the command.
- Separate the arguments with commas. You can include spaces after the commas that separate arguments — for example, @skip("\:", 1).
- If you omit an optional argument (indicated by brackets in syntax), you must still include the correct number of commas in the correct places. For example, the following @item2 command omits the optional second argument:

  ```
  @item2("^",)
  ```

Depending on the command, an argument can be any of the following six types:

Argument Type	*Description*	*Example*
String constant	A group of characters, such as the text of an item or a date value. A string contains standard alphabetic, numeric, and punctuation characters. The backslash (\) is a special character that instructs TXT2STF to ignore the next character. String constants must be enclosed in a pair of quotation marks (" ").	In **`@append_item("Derived from E-Mail")`**, the phrase "Derived from E-Mail" is a string that the @append_item command adds, without quotes, to the end of the current item.

continued

Argument Type	*Description*	*Example*
File name string constant	The name of a DOS file or directory. A file name string constant contains standard alphabetic characters. There are no special characters for file name string constants. File name string constants must be enclosed in a pair of quotation marks (" ").	In **`@make_external_note ("\agenda")`**, the phrase "\agenda" is the file name string constant that names the directory where @make_external_note stores a note.
Variable name string constant	The name of a variable. A variable name string constant defines a variable or references it by name. A variable name string contains standard alphabetic characters. There are no special characters for variable name string constants. Variable name string constants must be enclosed in a pair of quotation marks (" ").	In **`@append("dest_var", source_var)`**, the phrase "dest_var" is the name of the variable into which text is *appended.*

continued

Argument Type	*Description*	*Example*
Pattern constant	A group of characters that describes a string for which to search. A pattern can include standard string characters and special pattern-matching characters, such as the backslash (\) or asterisk (*). (See Chapter 4.) Patterns must be enclosed in a pair of quotation marks (" ").	In **`@note("^---*")`**, the pattern "^---*" describes the string that signals the end of note text in the text file; in this case, a text line starting with at least three hyphens.
Numeric constant	A number; this value cannot contain non-numeric characters.	In **`@skip_lines(3)`**, the number 3 specifies that TXT2STF should skip down three lines in the text file.
Type constant	A constant that identifies a type of category. Type constant can be "date", "exclusive", "numeric", "standard", "unindexed", or (@table only) "item". The type constant must be enclosed in a pair of quotation marks (" ").	In **`@category(",","Target Dates\:","date")`**, the "date" type constant instructs @category to make categories into dates.
Logical constant	A constant that specifies either N (no) or Y (yes). Command syntax shows these constants as *N* (italic uppercase N).	In **`@trim("^Who","How Many",N)`**, logical constant N instructs @trim to not trim the search strings Who and How Many when trimming the text between these strings.

continued

Argument Type	*Description*	*Example*
Variable	The name of a variable, *without* surrounding quotation marks. A variable name, without quotation marks, tells TXT2STF to use the *contents* of a variable. Variables can be used in the place of string constants, file name strings constants, and pattern constants. Variables *cannot* be used in place of variable name string constants, numeric constants, type constants, or logical constants.	In `@append("dest_var", source_var)`, source_var is the variable whose contents are copied into variable dest_var.

About Variables

You can specify variables as arguments in many definition file commands. You can use variables in commands to:

- Store text from one text line for use in a command that converts a later text line.
- Equate a long pattern to a short variable name and then specify the variable in your commands. This can make your definition file easier to read.

Note Variables can only be used as arguments for commands. You cannot use variables to represent the patterns that begin statements.

To create variables and store information in them, use variable commands. A variable can contain a maximum of 512 characters at a time.

Specifying Variables in Commands

A variable name can be up to 32 characters in length. Variable names can be more than one word and can include spaces. Variable names *cannot* include punctuation and cannot begin with a number. TXT2STF ignores the case of a variable when using it. This means that you can specify a variable using uppercase, lowercase, or a combination of the two.

When you identify a variable in a command

- Enclose a variable name in a pair of quotation marks (" ") when you need to simply *name* the variable.

  For example, you name a variable in @equate to associate the variable name with a specified value. To associate the string SOURCE-QIO-A with the variable Origin, you specify @equate("Origin","SOURCE-QIO-A").

- Specify the variable name without quotation marks when you want TXT2STF to use the *contents* of the variable.

  For example, to append the contents of the Name1 variable to the end of an item, specify @append_item(Name1).

Predefined Variables

Agenda supplies six variables whose values are predefined by Agenda: FILENAME, EOF, CR, D, E, and W. You can use these variables anywhere you need to use the Agenda-defined value. TXT2STF ignores the case of these variables. This means you can use uppercase, lowercase, or a combination of the two when naming any of these variables.

FILENAME

You can use FILENAME in commands to stand for the current text file name. For example, you can use this variable in @custom_category to assign each item to a category whose name is the file from which the item came.

As another example, you can use FILENAME to attach information from external files, such as electronic mail files, as note files. To do this, you can write a definition file to process the external files, creating a separate item for each external file. To attach an external file as the note file for an item, include FILENAME in @note_file. By specifying FILENAME as the external file name, you instruct TXT2STF to attach the current text file as a note file.

EOF

EOF stands for end of file. You can use this variable to specify the termination point in an @item2, @note, @category_note, @make_external_note, or @skip_lines.

CR

CR stands for carriage return. You can use this variable whenever you want TXT2STF to use the combination of carriage return (ASCII decimal 13) and linefeed (ASCII decimal 10) as a value.

For example, you can use the variable CR in an @append_note command to specify where a new text line starts in the note text currently in the note buffer. When the note is imported into Agenda, the note text will break to a new line where the CR adds the carriage return and linefeed.

CR is not affected by the TXT2STF /S command-line option. (See Chapter 6.)

D, E, and W

You can use these variables to represent Agenda-defined date categories in definition file commands:

Variable	*Means*
D	Done date
E	Entry date
W	When date

@append

@append adds a string to the end of the contents of a specified variable.

Syntax

@append("*variable-name*",*string*)

variable-name identifies the variable to which to append the string. *variable-name* is a variable name string constant.

string is the string to be appended. *string* is a string constant. You can specify the actual string value, in a pair of quotation marks (" "), or the name of a variable that contains the string.

Example

The following sample command adds the text Company to the end of the variable tempvar:

```
\                    @append("tempvar","Company")
```

In this example, if tempvar contains the value One (the word One followed by a space) before @append is executed, it contains the phrase One Company after this command.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	None
Variables	Can create new variables or add to the contents of existing ones
Location in text file	None
Structured file	None

@append_item

@append_item adds a string to the end of the current contents of the item buffer. If TXT2STF is not currently constructing an item, this command also begins the item.

Syntax

@append_item(*string*[,*N*])

string is the string to be added to the end of the current item buffer. *string* is a string constant. You can specify an actual string value, in a pair of quotation marks (" "), or the name of a variable that contains the string.

N (optional) prohibits TXT2STF from adding text to the note buffer if the item buffer becomes full.

Example

This example shows one way to convert the following memo:

To: Jill
From: Linda

Subject: Your proposal

> Jill, I liked your proposal
> but I had a few questions,
> could we get together next Wednesday
> to talk about it?

The sample definition file shown below converts the names Jill and Linda into categories and converts the remaining memo text into an item.

The definition file creates the item in two stages. First, it puts the subject text (Your proposal) into the item buffer and uses @append_item to add a space character ( ) to the end of the item buffer.

Next it processes each line of the indented text. It executes @trim to remove the leading spaces from the text line and puts the remaining text into the variable called LINE. Then it executes @append_item to add the text in variable LINE to the item buffer, followed by another @append_item that adds a space character ( ) to the end of the item buffer. These changes eliminate the existing line breaks in the message. On import, Agenda wraps the item text to fit the item column width.

```
#Converting memo text using @append_item

START                 @equate("SPACE"," ")
                      @equate("PERSON","")
                      @start()

#Create To category ("To:" starts each memo in the file)

"^To\:"               @end()
                      @trim("^To\:", "$", N, "PERSON")
                      @custom_category(PERSON, "To",)

#Assign a person to the From category

"^From\:"             @trim("^From\:", "$", N, "PERSON")
                      @custom_category(PERSON, "From",)
```

continued

```
#Add subject line to item buffer
"^Subject\:"        @trim("^","Subject\:",,)
                    @item2("$")
                    @append_item(SPACE)
#Discard blank lines so "~To\:"(NOT To:) won't match them
"^[ ]*$"            @equate("blanks","")
#Add each remaining line to item buffer
"^~To\:"            @trim("^: *","$", N, "LINE")
                    @append_item(LINE)
                    @append_item(SPACE)
END                 @end()
```

When the resulting structured file is imported into Agenda, To and From are added as categories under the category specified when you use the Agenda **File Transfer Import** command. The name Jill is added as a child of To, and Linda is added as a child of From. The item, which consists of the text from the Subject line plus the four-line memo, is assigned to the categories Jill and Linda.

To see what the structured file for this example looks like, see "A Simple Electronic Mail Example" in Appendix B.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Adds to the contents of the item buffer. If the item buffer becomes full, the rest of the string specified is added to the end of the note buffer, unless you specify other-wise. TXT2STF issues a warning when the item buffer becomes full.
Variables	None
Location in text file	None
Structured file	When the next @end is executed, the appended string is part of the item added to the structured file.

@append_note

@append_note adds a string to the end of the current note, as stored in the note buffer. If TXT2STF is not currently constructing a note, this command also begins a note for the current item.

Syntax

@append_note(*string*)

string is the string to be added to the end of the note buffer. *string* is a string constant. You can specify an actual string value, in a pair of quotation marks (" "), or the name of a variable that contains the string.

Example

This example shows one way to convert the following memo:

```
To:  Jill
From: Linda

Subject:  Your proposal

                Jill, I liked your proposal
                but I had a few questions,
                could we get together next Wednesday
                to talk about it?
```

The sample definition file shown below converts the names Jill and Linda into categories and converts the text on the Subject line into an item. Then it converts the remaining memo text into an item note. It executes @trim to put the memo text into a variable (without the leading spaces) and executes @append_note to add the text to the note buffer, followed by an @append_note to add a space character ( ) to the end of the note buffer. These changes eliminate the existing line breaks in the message. On import, Agenda wraps the note text to fit the note screen.

```
#Converting memo text using @append_note

START                @equate("SPACE"," ")
                     @equate("PERSON","")
                     @start()

#Create To category ("To:" starts each memo in the file)

"^To\:"              @end()
                     @trim("^To\:", "$", N, "PERSON")
                     @custom_category(PERSON, "To",)

#Assign a person to the From category

"^From\:"            @trim("^From\:", "$", N, "PERSON")
                     @custom_category(PERSON, "From",)

#Add subject line to item buffer

"^Subject\:"         @trim("^","Subject\:",,)
                     @item2("$")

#Discard blank lines so "~To\:"(NOT To:) won't match them

"^[ ]*$"             @equate("blanks","")

#Add each remaining line to note buffer

"^~*To\:"            @trim("^: *","$", N, "LINE")
                     @append_note(LINE)
                     @append_note(SPACE)

END                  @end()
```

When the resulting structured file is imported into Agenda, To and From are added as categories under the category specified when you use the Agenda **File Transfer Import** command. The name Jill is added as a child of To and Linda is added as a child of From. The item, which consists of the text from the Subject line, is assigned to the categories Jill and Linda. The note, which consists of the body of the message, is added to the item.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Adds to the contents of the note buffer. If the note buffer becomes full, the remaining portion of the string is ignored. TXT2STF issues a warning that the note buffer is full.
Variables	None
Location in text file	None
Structured file	When the next @end is executed, the appended string is part of the note added to the structured file.

@category

@category creates one or more categories from the input text, and optionally makes them children of a specified parent category. The category is added as a child of the current category, as specified when you use the Agenda **File Transfer Import** command, unless you specify that the parent category is a child of MAIN.

If the specified category already exists, Agenda retains the existing category and does not create a duplicate category. If you specify a parent, the category created by @category becomes its child when the structured file is imported into Agenda. If the specified parent does not already exist, Agenda creates it on import.

The current item is assigned to this category. The item and category must both be created before an @end copies the item and its categories to the structured file.

The item buffer can be empty when TXT2STF copies the category to the structured file. When you import the resulting structured file, Agenda adds the category to the category hierarchy but does not assign the any items to the category.

Syntax

@category(*separator-pattern*,[*parent*],["*type*"]) *separator-pattern* identifies the character or characters that TXT2STF uses to determine where a given value ends and the next value begins. *separator-pattern* is a pattern constant. This pattern can include the space character ( ). You can specify the pattern itself, in a pair of quotation marks (" "), or the name of the variable that contains the separator pattern.

TXT2STF makes each value into a separate category. The categories do not include the characters that match the separator pattern.

Tip Specifying the dollar sign ($) (the end-of-line character) as the separator pattern tells TXT2STF to make the entire text line a single category.

parent (optional) is the parent of the new categories. *parent* is a string constant. You can specify the parent category name, in a pair of quotation marks (" "), or the name of a variable that contains the category name.

You can specify the entire name of the parent category, including its parents. Use *two* backslashes (\\) to separate category names, without including any spaces. (A single backslash (\) is a special character that tells TXT2STF to use the *next* character as a regular character and not as a special character.) For example, to identify a parent category named Sales that is a child of Personnel in the category hierarchy, specify "Personnel\\Sales".

If a category name string begins with two backslashes (\\) the first category in the string is added as an immediate child of MAIN.

type (optional) specifies the type of category to be created. *type* is a type constant. You can specify "date", "exclusive", "numeric", "standard", or "unindexed". If you omit *type*, TXT2STF uses "standard". If the category already exists and is a different type, Agenda retains the original category and type on import.

Example

This example shows how to convert the following text in order to import the memo recipient names as child categories under the parent category To, which is itself a child of People, and the company name under the parent category Company:

```
To: Todd, Lynne, Sam, Julie
Company: ABC Corporation
```

That is, it converts the above text so you can import the following category hierarchy, with People and Company both children of MAIN:

```
People
   To
      Todd
      Lynne
      Sam
      Julie
Company
   ABC Corporation
```

The sample definition file converts the text file by first processing the To line and then the Company line. The statement that starts with the pattern "^To\:" processes the first text line. @trim removes the string To:, which is the word To followed by a colon (:), from the line. Then @category divides the remaining text line into separate values using a comma (,) or space character ( ), or a combination of the two (as specified by the pattern "[,]+"). Then @category makes each of the resulting values into categories.

To process the second text line, the definition file trims the string Company: from the line. @category makes the rest of the current line a category under the parent category Company.

```
#Definition file that creates categories

START                 @start()

#Convert names on "To" line into multiple categories

"^To\: "              @trim("^","\:",,)
                      @category("[, ]+","\\People\\To",)

#Convert Co. name on "Company" line into one category

"^Company\: "         @trim("^","\:",,)
                      @category("$","\\Company",)

END                   @end()
```

When the resulting structured file is imported into Agenda, the People and Company categories are added as children of MAIN.

See @custom_category for an example of using @category to create a numeric category and for an example of using @custom_category to create a single category from a text line.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Adds one or more categories to TXT2STF category buffers
Variables	None
Location in text file	None
Structured file	When the next @end command is executed, TXT2STF copies a category definition for each category buffer created by @category.

@category_note

@category_note creates a category note and assigns it to the most recently created category. The category and category note must both be created before an @end copies the category and its note to the structured file.

Syntax

@category_note(*termination-pattern*)

termination-pattern describes the string that ends the category note. *termination-pattern* is a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

The note starts with the current text line and ends the note when it finds the specified termination pattern. The note does not include the line containing the termination pattern; that line is the next text line to be processed.

Tip To use only the current text line as the note, specify caret (^) as the termination pattern. This causes TXT2STF to terminate the note when it reaches the start of the *next* text line.

Example

This example shows how to convert a file that contains these lines and make the name Mia Fischer into a category, and the company name and address into a note for the category:

```
Send To:        Mia Fischer
                Toys Toys Toys Inc.
                1234 Pinkham Road
                Storyville, WA 02138
-------------------------------------------
```

The following sample definition file searches for the string Send To: at the start of a text line and trims the string from the line. Then, @category adds the name Mia Fischer as a child of the To category. @skip_lines then instructs TXT2STF to skip to the next line, which becomes the first line in a category note created by @category_note. The category note ends when TXT2STF finds a row of dashes in the text file.

```
START               @start()

#Trim out "Send To:" and make rest of line a category

"^Send To\:"        @trim("^","\:",)
                    @category("$","\\To",)

#Skip to next line and make rest of text a category note

                    @skip_lines(1)
                    @category_note("^---*")

END                 @end()
```

When the resulting structured file is imported, the To category is added as a child of MAIN. The note is added as its category note.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	The first line that contains the termination pattern.
Item, category, and note buffers	The note created by this command is associated with the category most recently added to the category buffer.
Variables	None

continued

Elements that commands can change	*Changes made by this command*
Location in text file	The new location is the beginning of the first line from the old current line that matched the pattern *termination-pattern*.
Structured file	When the next @end is executed, a category note is defined in the structured file for the category mentioned above.

@category_note_file

@category_note_file identifies the external note file for the current category. Agenda attaches the note to the most recently created category. The category and category note must both be created before an @end copies the category and its note to the structured file.

Note If you include both @category_note_file and @category_note for a category, TXT2STF ignores the @category_note and copies only the name of the note file specified by @category_note_file to the structured file.

Note TXT2STF does not check whether the file exists. If the file does not exist, Agenda asks whether you want to create it when you ask to look at the note in the Agenda file.

Syntax

@category_note_file(*file-name*)

file-name is the note file name. *file-name* is a file name string constant. You can specify the name itself, in a pair of quotation marks (" "), or the name of a variable that contains the file name.

Example

This example creates a category called Quarter2 under the parent category Fiscal Year. It specifies the file QUARTER.DAT as a note file for that category.

```
\           @custom_category("Quarter2","Fiscal Year",)
\           @category_note_file("quarter.dat")
```

When the resulting structured file is imported, the Fiscal Year category is added as a child of the category specified when you use the Agenda **File Transfer Import** command. Quarter2 is added as a child of Fiscal Year. The note file QUARTER.DAT is attached as the category note for Quarter2.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	The note for the last category defined is set to the file name specified.
Variables	None
Location in text file	None
Structured file	When the next @end is executed, a category note is defined in the structured file for the category mentioned above.

@custom_category

@custom_category creates a category from text that you specify and optionally makes them children of a specified parent category. The category is added as a child of the current category, as specified when you use the Agenda **File Transfer Import** command, unless you specify that the parent category is a child of MAIN.

If the category already exists, Agenda retains the existing category and does not create a duplicate category. If you specify a parent, the category created by @custom_category becomes its child when the structured file is imported into Agenda. If the specified parent does not already exist, Agenda creates it on import.

The current item is assigned to this category. The item and category must both be created before an @end copies the item and its categories to the structured file.

The item buffer can be empty when TXT2STF copies the category to the structured file. When you import the resulting structured file, Agenda adds the category to the category hierarchy but does not assign any items to the category.

Syntax

@custom_category(*category*,[*parent*],["*type*"])

category is the category name. *category* is a string constant. You can specify the category as a string, in a pair of quotation marks (" "), or the name of a variable that contains the string.

parent (optional) is the parent category to which the new category is added. *parent* is a string constant. You can specify a parent category name, in a pair of quotation marks (" "), or the name of a variable that contains a category name.

You can specify the entire name of the parent category, including its parents. Use *two* backslashes (\\) to separate category names. (A single backslash (\) is a special character that tells TXT2STF to use the *next* character as a regular character and not as a special character.) For example, to identify a parent category named Sales that is a child of Personnel in the category hierarchy, specify "Personnel\\Sales".

If a category name string begins with two backslashes (\\) the first category in the string is added as an immediate child of MAIN.

type (optional) specifies the type of category to create. *type* is a type constant. You can specify "date", "exclusive", "numeric", "standard", or "unindexed". If you omit *type*, TXT2STF uses "standard". If the category already exists and is a different type, Agenda retains the original category and type on import.

Example

This example shows one way to convert the following sample text lines to make ABC Corporation an item that is assigned to the category Lyle Clark and also is assigned to the numeric category named Company # with the number 5012.

```
Company Number: 5012
Company: ABC Corporation
Contact: Lyle Clark
```

The following sample definition file converts the above text file. The first statement is executed when TXT2STF finds the string Company Num at the start of a text line. @skip deletes the start of the text line up through the colon(:). @category makes the rest of the text line (the number 5012) a numeric value for the category Company #, which is a numeric category.

The next statement is executed when TXT2STF finds the string Company: at the start of a text line. @skip deletes the string Company: from the line. @item2 makes the remaining text line (ABC Corporation) into an item.

The final statement is executed when TXT2STF finds the string Contact: at the start of a text line. @trim puts the entire text line into the variable contact_var, after first removing the string Company:, through the colon (:), from the line. Then @custom_category adds the contents of contact_var (Lyle Clark) as a category under the parent category Contact. Since @custom_category omits the third argument, which specifies the type of category to create, the name ABC Corporation is added as a standard category.

```
START               @start()

"^Company num"      @skip("\:",1)
                    @category("$","Company #","numeric")

"^Company\:"        @skip("\:",1)
                    @item2("$")

"^Contact\:"        @trim("^.*\:","$",N,"contact_var")
                    @custom_category(contact_var,"\\Contact",)
                    @end()
```

When the resulting structured file is imported, the categories Contact and Company # are added as children of MAIN. Lyle Clark is added as a child of Contact. The item ABC Corporation is assigned to the category Lyle Clark and also assigned to the category Company # with the numeric value 5012.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Creates a new category
Variables	None
Location in text file	None
Structured file	When the next @end is executed, the category created is added to the structured file.

@date

@date lets you create assignments to date categories. @date specifies the category name and optionally specifies a date value, such as 11/12/90. The date category can be a category defined by Agenda (Entry, When, or Done) or any other date category you create.

When @end copies the category specified by @date to the structured file, it also copies the item in the item buffer to the structured file. When the structured file is imported, Agenda uses the date value specified by @date to assign the item to the date category.

If the date category already exists, Agenda retains the existing category and does not create a duplicate category.

Caution If there is no item in the item buffer when the next @end is executed, then on import Agenda creates the category but does not import the date value since there is no item to assign to the date category.

To assign the same date value to more than one date category, include a separate @date for each date category.

By default, Agenda assumes that the dates in the structured file are in the format MM/DD/YY. To specify a different format, use @date_format.

Syntax

@date(*category*[,*value*])

category is the name of the date category. This value must be a string constant. You can specify either the category name, in a pair of quotation marks (" "), or a previously defined variable that contains the name of the category.

value (optional) is the date value used in the assignment, for example 11/12/90. This value must be a string constant. You can specify either the value itself, in a pair of quotation marks (" "), or a previously defined variable that contains the date value.

If you omit *value*, TXT2STF uses the entire current text line in the assignment. On import, Agenda discards any portion of the line that is not part of a valid date.

Example

This sample locates a text line that starts with a date value in standard MM/DD/YY format, and makes the date value a When date. The sample also creates an item from a text line that starts with the word What.

```
"^(:d:d/:d:d/:d:d)"    @date(W)

"^What"                @item2("$")
                       @end()
```

When the resulting structured file is imported, the item created by the @item2 command is assigned to When.

See @date_format for another example.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Adds a date category to TXT2STF category buffers
Variables	None
Location in text file	None
Structured file	When the next @end is executed, the date category is added to the structured file. If an item exists in the item buffer at that time, TXT2STF also adds the item to the structured file and assigns it to the date category using the value specified by the @date. Otherwise, there is no item to assign to the category, and so the category is imported into Agenda but the associated date value is ignored.

@date_format

@date_format describes the format of the dates and times in the structured file. On import, Agenda uses this date format specification to interpret dates in the structured file. @date_format is particularly useful if you are likely to have dates in international date formats.

By default, Agenda assumes that the dates in the structured file are in the format MM/DD/YY and that times are specified using a 24-hour clock. Use @date_format if dates in the text file have a different format. You can omit @date_format if all dates in the structured file are specified either in the default format (MM/DD/YY) or in a combination of words and numbers, as in "Thursday November 8, 1990".

You can use @date_format once at the beginning of the definition file if your text file contains dates in only one format. If the date format changes at intervals in your text file, include a new @date_format immediately *before* any commands in a statement that process dates in the new date format.

TXT2STF copies the date format to the structured file when an @end is executed. If more than one @date_format is executed between a given @start/@end (or @end/@end) pair, only the last @date_format is used.

Syntax

@date_format(*format-number*)

format-number is a numeric constant that indicates the format of the dates in the text file. Use any of the following format numbers (format 1 is the default):

Format number	*Associated date format*	*Associated time format*
1	MM/DD/YY	24-hour clock
2	DD/MM/YY	24-hour clock
3	DD.MM.YY	24-hour clock
4	YY-MM-DD	24-hour clock
5	DD-MMM	24-hour clock
6	DD-MMM-YY	24-hour clock
7	MM/DD/YY	12-hour clock
8	DD/MM/YY	12-hour clock
9	DD.MM.YY	12-hour clock
10	YY-MM-DD	12-hour clock
11	DD-MMM	12-hour clock
12	DD-MMM-YY	12-hour clock

Note These are only a subset of the date formats supported by Agenda. (See Chapter 7 in the *User's Guide.*) For another way to convert dates into different formats, see @replace.

Example

This example converts the dates in electronic mail messages when you get mail from mail nodes in both the United Kingdom (UK) and the United States (US). If the mail has the string "UK_Node" (which in this example identifies it as being from the UK), the date format is set to DD/MM/YY. If it has the string "US_Node" (which identifies it as being from the US), the date format is set to MM/DD/YY.

```
"^Via"              @start()
.
.
"UK_Node"           @date_format(2)
"US_Node"           @date_format(1)
.
.
":d:d/:d:d/:d:d"    @date(W)
"Subject"           @item2("$")
                    @end()
```

When the resulting structured file is imported, the item created by @item2 is assigned to the When date created by @date.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	None
Variables	None
Location in text file	None
Structured file	When the next @end is executed, TXT2STF copies the most recent date format specification to the structured file, which tells Agenda the date format to use to interpret all subsequent dates in the structured file.

@end

Copies the contents of TXT2STF item, category, and note buffers to the structured file as items, notes, and categories. @end then initializes the buffers.

An @end must be executed for each item to be copied to the structured file. You do not, however, need to include an @end for items created by @table because @table internally executes an @end (and @start) for each item it creates.

@end also should be the last command executed in the definition file. There are two ways to ensure that @end executes last:

- Add a statement of the following form to the definition file:

  END   @end()

- Make @end the last command in the statement that matches the last line of your text file.

Caution If the definition file ends with a command other than @end, the structured file might not contain all the information it is supposed to.

@end does not initialize the contents of variables. To initialize a variable you can use @append, @equate, @grab, @set, and @trim.

Syntax

@end()

@end has no arguments, but the parentheses are required.

Example

This sample statement copies the accumulated information to a structured file when it finds the word Sincerely followed by a comma (,) at the end of a text line.

```
"Sincerely,$"      @end()
```

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Clears these buffers after writing them to the structured file
Variables	None
Location in text file	None
Structured file	Copies the contents of item, note, and category buffers to the structured file

@equate

@equate initializes a variable and then puts the specified string in the variable. If the variable already exists, the information it contained before @equate is lost.

@equate is a convenient way to initialize variables. You also can use it to associate the variable name with a pattern or text string. After you use @equate, you can specify the variable name in other commands instead of the pattern or text string.

Syntax

@equate("*variable-name*","*string*")

variable-name is the variable into which the string is placed. *variable-name* is a variable name string constant.

string is the pattern or string constant, enclosed in a pair of quotation marks (" "), or a previously defined variable that contains a string or pattern constant.

Example

In this example, @equate associates the variable with the string Company. Then, on any line that starts Company: or any other string that begins with the string Co and ends with a colon (:), TXT2STF puts everything after the colon (:) into the variable Category. @custom_category then uses both variables Category and Parent to create a category.

```
START           @start()
                @equate("Parent","\\Company")

"^Co.*\:"       @trim("^Co.*\:", "$", N, "Category")
                @custom_category(Category, Parent,)
                @end()
```

When the resulting structured file is imported into Agenda, the Company category is added as a child of MAIN.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	None
Variables	Creates new variables or replaces the contents of existing ones
Location in text file	None
Structured file	None

@grab

@grab temporarily divides portions of the current text line into values and puts one or more of the resulting values into variables. If a variable already exists, its previous contents are lost. If a variable does not exist, TXT2STF creates it.

@grab leaves the text line intact in memory. The next command in the statement can operate on the same text line that @grab originally processed. All commands that follow @grab can use values that @grab stores in variables.

To put text into variables, @grab first divides the line into values using the specified separator pattern. It then assigns a number to each value, assigning the number 1 (one) to the first (leftmost) value, the number 2 to the second, and so forth. Finally, @grab puts into variables the values you identify by number. For example, to put the second value from the text line in a variable, you tell @grab to put value number 2 in a specific variable.

Syntax

@grab(*separator-pattern,position-number,"variable-name"*
    [*,position-number,"variable-name"*,...])

separator-pattern describes the string that TXT2STF uses to divide the line into separate values. This pattern specifies where a given value ends and another begins. *separator-pattern* must be a pattern constant. You can specify the actual string, in a pair of quotation marks (" "), or a variable that contains the string value.

position-number identifies, by number, the value to put in a variable. *position-number* must be a numeric constant. For each position number, you must also identify a variable.

variable-name identifies the variable in which to put the value identified by *position-number*. *variable* is a variable name constant.

To identify more than one value in a single @grab, specify a *position-number* and *variable* for each value. The position number and variable pairs *must* all be on the same line as the rest of the @grab command. Put a comma (,) before each new position number. If you identify a value that does not exist, TXT2STF ignores the position number and its associated variable, and continues executing the command. For example, if you specify a variable for the fifth value, and only four values exist, TXT2STF does not change the specified variable.

Example

This example shows one way to convert the following text, by making make the first word on the line, such as To or From, into a parent category, and the name on the line into its child category.

```
To: Mark
From: Sonjaya
Company: ABC Corporation
```

That is, you import the following category hierarchy:

```
To
    Mark
From
    Sonjaya
Company
    ABC Corporation
```

The following sample definition file processes one text line at a time. The pattern "^.*\:" matches each line that starts with a string followed by a colon (:). After the line is matched, @grab puts the first string on the text line, such as To, in a variable called Parent and puts the second string after the colon (:) in a variable called Category. For each line, @custom_category creates a category using the text current in the Category variable, and whose parent is the text stored in the Parent variable.

```
START               @start()

"^.*\:"             @grab("\:",1,"Parent",2,"Category")
                    @custom_category(Category,Parent,)

END                 @end()
```

When the resulting structured file is imported, categories To, From, and Company are added as children of the category you specify when you use the Agenda **File Transfer Import** command.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	None
Variables	Creates new variables or replaces the contents of existing ones
Location in text file	None
Structured file	None

@item

@item creates an item using text from the text file. The item starts with the current text line and can include subsequent text lines from the text file, depending on the arguments you specify in the command.

Note @item is an alternative to @item2. If you already have definition files that use @item, you can continue to use them. However, to have more control over how TXT2STF constructs items, consider using @item2 and not @item in your definition files. (See "@item2" later in this chapter.)

The item created by @item includes the entire current text line. (If you want to remove characters from this text line, use trimming commands *before* executing @item.) @item adds additional text lines to the item buffer until it either finds the termination pattern for the item or the item buffer contains the maximum number of characters, as specified in the @item command. (This differs from @item2, which copies text only up to where the termination pattern or maximum length is encountered, even if this means stopping in the middle of the first text line.)

TXT2STF maintains a single item buffer. This means you can create only one item at a time. If more than one @item is executed between the @start (or @end) that initializes the buffer and the @end that copies the item buffer to the structured file, TXT2STF adds the text to the end of the item and/or note buffer, as appropriate.

The item buffer can contain up to 350 characters. If this buffer fills during execution of @item, TXT2STF puts the remaining characters in the current note buffer, unless you specify otherwise in @item. The TXT2STF note buffer can contain up to 10,000 characters. @item stops copying text to the note buffer when it encounters the termination pattern in a text line, fills the note buffer, or reaches the end of the text file.

For each text line that it processes, @item always uses the *entire* text line. It does not process portions of a text line. (This differs from @item2, which processes each text line character by character.)

@item always copies the entire first text line it processes to the item buffer. Then TXT2STF loads the next unprocessed text line into memory. At this time, @item starts searching for the termination pattern or testing to see if the maximum number of characters have been added to the item. TXT2STF goes through line after line of text, adding each one to the item (or note) buffer, until either the termination string or maximum number of characters is encountered. When either of these occurrences ends @item, TXT2STF loads the next unprocessed text line into memory, and continues processing the text file. (This is different from @item2, which can copy a portion of the current text line to the item buffer, which means that the next command in the current statement starts where @item2 stopped processing the current text line. See "@item2" later in this chapter.)

Syntax

@item(*termination-pattern*,*total-length*,[*N*])

termination-pattern describes the string that ends the item text. *termination-pattern* is a pattern constant. The pattern can consist of one or more characters, including the space character ( ). You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

Tip To use only the current text line as the item, specify caret (^), which is the start-of-line character, as the termination pattern. This causes TXT2STF to terminate the item when it reaches the start of the *next* text line.

@item starts searching for the termination pattern *after* copying the first text line to the item buffer. The line that contains the termination pattern is *not* included in the item. If @item does *not* specify a total length, @item stops executing when it the termination pattern is matched. Otherwise, @item continues until either it copies the number of characters specified by *total-length* into the item (or note) buffer or reaches the end of the text file.

total-length is a numeric constant that specifies the maximum number of characters @item can process. *total-length* is a numeric constant. You can specify up to 350 for this value.

@item adds the *entire* current text line to the buffer when any character in the line causes the item buffer to contain the maximum number of characters specified by *total-length,* even if this adds more than the specified number of characters. If @item allows text to be added to the note buffer (specified by *N,* below) *and* the termination pattern has not yet been encountered, @item adds text to the note buffer until it finds the termination pattern. The line that contains the termination pattern is *not* included in the note.

N (optional) prohibits TXT2STF from adding text to the note if the item buffer becomes full.

Example

This example shows one way to convert the following text into an item and item note:

```
Re:      Lunch

Can't make lunch appointment today, can we reschedule for next week?
```

The following sample definition file statement executes when TXT2STF finds the string Re: at the start of a line. @custom_category creates a category called Messages. @skip causes TXT2STF to skip over the string Re: on the current line, and @item makes Lunch an item, by including everything up to the start of the next line, as specified by the pattern "^". (The 80 supplements the "^" specification, telling @item that the item can be no longer than 80 characters in length.) @note creates a note from the remaining text in the file, up to the end of the text file, as specified by the predefined variable EOF, or it fills the note buffer.

```
"^Re\:"  @start()
         @custom_category("Messages",,)
         @skip(": +",1)
         @item("^",80,)
         @note(EOF)
         @end()
```

When the resulting structured file is imported into Agenda, Messages is added as a child of the category specified when you use the Agenda **File Transfer Import** command. The item created by @item is assigned to the Messages category. The note is added as an item note.

Note As an alternative to @item("^",80,) and @note(EOF), you could use the single command @item(EOF,80,), which makes an item from the current line (which internally is assumed to by 80 characters long) and then puts the remaining text lines into a note for the item.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	The text line that matched the termination string, or the text line *after* the text line that caused the maximum number of characters to be added to the item (or note) buffer.
Item, category, and note buffers	Appends text to the item buffer and may append text to the note buffer.
Variables	None
Location in text file	Advances to the line where the item ends, or to the *next* line if the item contains only one text line.
Structured file	When the next @end is executed, an item is defined in the structured file using the contents of the item buffer. Text appended to the note buffer by @item is added as an item note.

@item2

@item2 creates an item using text from the text file. The item starts at the beginning of the current text line and can include as many characters from the current text line as necessary. The item also can include subsequent text lines from the text file, depending on the arguments you specify in the command.

Note @item2 is an alternative to @item, and provides greater flexibility in how you can create an item.

The item created by @item2 starts at the beginning of the current text line, and continues until either it finds the termination pattern for the item or the item buffer contains the maximum number of characters, as specified in the @item command. (This differs from @item, which copies the entire first text line into the item buffer before it starts looking for the termination pattern or checking to see if the item buffer is full. @item2 differs also in that it copies text into the item buffer character by character, and not line by line in the way @item does.)

TXT2STF maintains a single item buffer. This means you can create only one item at a time. If more than one @item2 is executed between the @start (or @end) that initializes the buffer and the @end that copies the item buffer to the structured file, TXT2STF adds the text to the end of the item and/or note buffer, as appropriate.

The item buffer can contain up to 350 characters. If this buffer fills during execution of @item2, TXT2STF puts the remaining characters in the current note buffer, unless you specify otherwise in @item. The TXT2STF note buffer can contain up to 10,000 characters. @item2 stops copying text to the note buffer when it encounters the termination pattern in a text line, fills the note buffer, or reaches the end of the text file.

@item2 copies text to the item buffer character by character. The next command in the *current* statement starts where @item2 stopped processing the current text line. (This differs from @item, which always copies entire lines of text into the item buffer. See "@item" earlier in this chapter for more information.)

If @item2 is the last command in the statement, TXT2STF recopies the most recently processed text line back into memory, which means that the next text line to be processed is the same text line where the item ended.

Syntax

@item2(*termination-pattern*[,*total-length*][,*N*])
or
@item2(*item-length*[,*N*])

termination-pattern describes the string that ends the item text. *termination-pattern* is a pattern constant. The pattern can consist of one or more characters, including the space character ( ). You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

Tip To use only the current text line as the item, specify a dollar sign ($) as the termination pattern. This causes TXT2STF to terminate the item when it reaches the end of the current line.

@item2 includes in the item buffer characters up to, but not including, the matched characters. If @item2 does not specify a total length for an item, @item2 stops executing when it matches the termination pattern. Otherwise, @item2 continues until either it copies the maximum number of characters to the item (or note) buffer or reaches the end of the text file.

total-length (optional) is a numeric constant that specifies the maximum number of characters @item2 can process. *total-length* is a numeric constant. You can specify up to 350 for this value.

@item2 stops adding characters to the item buffer when the item buffer contains the maximum number of characters. If @item2 allows text to be added to the note buffer (specified by *N*, below) *and* the termination pattern has not yet been encountered, @item2 adds text to the note buffer until it finds the termination pattern. The termination pattern is not included in the note.

item-length is a numeric constant that specifies the maximum number of characters the item can contain. TXT2STF stops adding characters to the item buffer when it contains the maximum number of characters. *item-length* is a numeric constant. When the item buffer contains this number of characters, TXT2STF stops executing @item2. You can specify up to 350 for this value.

N (optional) prohibits TXT2STF from adding text to the note if the item buffer becomes full.

Example

This example shows one way to convert the following text into an item and item note:

```
Re:      Lunch

Can't make lunch appointment today, can we reschedule for next week?
```

The following sample definition file statement executes when TXT2STF finds the string Re: at the start of a line. @custom_category creates a category called Messages. @skip causes TXT2STF to skip over the string Re: on the current line, and @item2 makes an item from the current line (Lunch). Then @note copies the remaining characters in the text file to the note buffer, until it reaches the end of the text file, as specified by the predefined variable EOF, or it fills the note buffer.

```
"^Re\:"  @start()
         @custom_category("Messages",,)
         @skip(": +",1)
         @item2("$")
         @note(EOF)
         @end()
```

When the resulting structured file is imported into Agenda, Messages is added as a child of the category specified when you use the Agenda **File Transfer Import** command. The item created by @item2 is assigned to the Messages category. The note is added as an item note.

Note As an alternative to @item2("$") and @note(EOF), you could use the single command @item2(EOF,80), which makes an item from everything in the current line (up to a maximum of 80 characters) and then puts the remaining text up to the end of the file into a note.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	The current line becomes the line where the item text ends.
Item, category, and note buffers	Appends text to the item buffer and may append text to the note buffer.
Variables	None
Location in text file	Advances to the next character after the item text.
Structured file	When the next @end is executed, the item is added in the structured file using the contents of the item buffer. Text appended to the note buffer by @item2 is added as an item note.

@make_external_note

@make_external_note creates an external note file and copies text from the text file to the note file. The note file is associated with the item in the item buffer.

If the item buffer is empty when TXT2STF copies the external note file name to the structured file, the note file is not attached to an item when the structured file is imported to Agenda. This occurs when there are no item commands between the previous command that cleared TXT2STF buffers (@start or @end) and the @end that copies the note to the structured file. When you import the resulting structured file, Agenda creates an empty item for the note.

TXT2STF gives the note file a name of the form NOTE*n*.AGN, where *n* is a number between 1 and 9999. TXT2STF increments *n* by 1 each time it creates a note file. If a file already exists with this name, TXT2STF retains the existing file and increments *n* until it creates a unique file name. TXT2STF stores the file in the directory you specify, and copies the name of the note file to the structured file.

If an error occurs when TXT2STF tries to create the note file, for example the specified directory does not exist or TXT2STF cannot create the file for another reason, TXT2STF reports the error, ignores the command, and continues converting the text file.

Syntax

@make_external_note(*termination-pattern*,["*directory*"])

termination-pattern describes the string that ends the note text. *termination-pattern* is a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

TXT2STF starts the note with the current text line. After adding that text line to the external note file, TXT2STF starts searching for the specified termination pattern, and ends the note when it reaches a line that contains that pattern. The note does not include the line containing the termination pattern; that text line is the next line to be processed.

directory (optional) is the name of the directory in which you want the note file to be stored. *directory* is a file name string constant. This value is a string constant. You can specify the directory name itself, in a pair of quotation marks (" "), or a predefined variable that contains the directory name.

To store the note in the current directory, omit the *directory* argument.

Example

The following sample statement is executed when TXT2STF finds the string Subject: at the start of a text line. @skip removes Subject: from the text line. @item2 makes an item from the remaining text on the current line. @make_external_note copies all remaining text lines, up to the end of the text file (as specified by the predefined variable EOF) into a new note file called NOTE*n*.AGN (where *n* is the next available number) in the Agenda directory (\AGENDA).

```
"^Subject\:"        @skip(" +",1)
                    @item2("$")
                    @make_external_note(EOF,"\agenda")
```

When the resulting structured file is imported into Agenda, the note file is attached to the item created by @item2.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	The current line becomes the first line from the old current line that has a string which matches the *termination-pattern.*
Item, category, and note buffers	Causes TXT2STF to ignore the contents of the note buffer.
Variables	None
Location in text file	The next command begins executing on the beginning of the new current line.
Structured file	When the next @end is executed, instead of writing the contents of the note buffer to the structured file, TXT2STF copies the name of the file note created by @make-_external_note.

@note

@note creates a note and assigns it to the item in the item buffer. If TXT2STF finds more @note commands before @end copies the item and note to the structured file, TXT2STF appends the text to the exis-ting note.

To attach a note file, use @note_file instead of @note. To assign the note to a category, use @category_note or @category_note_file.

Caution @note can only create notes that contain up to 10,000 characters (about seven pages of double spaced text). If the text contains more than this number of characters, use @make_external_note, which puts any number of characters in a text file and attaches the text file as an external note.

If the item buffer is empty when TXT2STF copies the note to the structured file, the note will not be assigned to an item when the structured file is imported to Agenda. This occurs when there are no item commands between the previous command that cleared TXT2STF buffers (@start or @end) and the @end that copies the note to the structured file. When you import the resulting structured file, Agenda creates an empty item for the note.

Syntax

@note(*termination-pattern*)

termination-pattern describes the string that ends the note text. *termination-pattern* is a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

TXT2STF starts the note with the current text line. After adding that text line to the note buffer, TXT2STF starts searching for the specified termination pattern, and ends the note when it reaches a line that contains that pattern. The note does not include the line containing the termination pattern; that text line is the next line to be processed.

Tip To use only the current text line as the note, specify caret (^) as the termination pattern. This causes TXT2STF to terminate the note when it reaches the start of the *next* text line.

Example

The following sample definition file creates an item from text on a line starting with the string QUOTE, and a category from the text on a line starting with the string PUT. When TXT2STF finds the string DETAILS at the start of a text line, it starts a note for the current item. The note goes up to, but does not include, the next line that starts with the string QUOTE.

```
"^Start_trans\:"  @start()
.
.

"^QUOTE  "        @skip(" +",1)
                  @custom_category("Quotes","\\Buys",)
                  @item2("$")

"^PUT  "          @skip(" +",1)
                  @category("$","\\Buys\\Puts",)

"^DETAILS  "      @note("^QUOTE  ")
                  @end()
```

When the resulting structured file is imported into Agenda, Buys is added as a children of MAIN. Quotes and Puts are children of Buys. The category created by @category is added as a child of Puts. The item created by @item2 is assigned to the Quotes category and to the category created as a child of Puts.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	The current line becomes the first line after the old current line that has a string that matches the *termination-pattern*.
Item, category, and note buffers	Appends into the note buffer all the information in the text file staring from the current location in the text file to the end of the line before the new current line.
Variables	None
Location in text file	The next command begins executing on the beginning of the new current line.
Structured file	When the next @end is executed, a note is defined in the structured file with the contents of the note buffer.

@note_file

@note_file identifies the external note file for the item in the item buffer.

Notes If you use both @note_file and @note, TXT2STF ignores @note and copies only the name of the note file specified by @note_file to the structured file.

TXT2STF does not check to see if the file exists. If the file does not exist, Agenda asks whether you want to create it when you try look at the note in the Agenda file.

Syntax

@note_file(*file-name*)

file-name is the name of the external note file. *file-name* is a file name string constant. You can specify the note file name itself, in a pair of quotation marks (" "), or the name of a variable that contains the note file name.

Example

The following sample definition file statement is executed when TXT2STF finds a line that starts with NEW VENDOR. @item2 creates an item from the current text line. @note_file makes the current text file the note file for the item. (FILENAME is a predefined variable that represents the name of the current text file.)

```
"^NEW VENDOR"       @item2("$")
                    @note_file(FILENAME)
```

When the resulting structured file is imported into Agenda, the original text file is attached as a note file to the item created by @item2. Since TXT2STF does not change the text file when it converts text in it, the contents of the original file are available as note text.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Causes TXT2STF to ignore the contents of the note buffer.
Variables	None
Location in text file	None
Structured file	When the next @end is executed, instead of writing the contents of the note buffer to the structured file, TXT2STF defines a file note whose name is *file-name*.

@numeric

@numeric lets you create assignments to numeric categories. @numeric specifies the category name and optionally specifies a numeric value.

When @end copies the category specified by @numeric to the structured file, it also copies the item in the item buffer to the structured file. When the structured file is imported, Agenda uses the numeric value specified by @numeric to assign the item to the numeric category.

If the numeric category already exists, Agenda retains the existing category and does not create a duplicate category.

Caution If there is no item in the item buffer when the next @end is executed, then on import Agenda creates the category but does not import the number since there is no item to assign to the numeric category.

Syntax

@numeric(*category*[,*value*])

category is the name of the numeric category. This value must be a string constant. You can specify either the category name itself, in a pair of quotation marks (" "), or a previously defined variable that contains the category name.

value (optional) is the numeric value used in the assignment. This value must be a string constant. You can specify either the value itself, in a pair of quotation marks (" "), or a previously defined variable that contains the value.

If you omit *value*, TXT2STF uses the entire current text line in the assignment. Agenda imports the text line starting from the first character on the line and continuing until it finds a nonnumeric character. Agenda ignores the rest of the line. If the first character is nonnumeric, Agenda ignores the entire line.

Example

The following sample command assigns the current item to the Cost category with a numeric value of 350.23.

```
\                 @numeric("Cost","350.23")
```

When the resulting text file is imported into Agenda, the numeric category Cost is imported as a child of the current category, as specified when you use the Agenda **File Transfer Import** command.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Adds a numeric category to TXT2STF category buffers.
Variables	None
Location in text file	None
Structured file	When the next @end is executed, the numeric category is added to the structured file. If an item exists in the item buffer at that time, TXT2STF also adds the item to the structured file and assigns it to the numeric category using the value specified by @numeric. Otherwise, there is no item to assign to the category, and so the category is imported into Agenda but the associated numeric value is ignored.

@replace

@replace replaces the specified number of occurrences of a given string in the text line with another string.

Syntax

@replace(*search-pattern*,*replacement-string*,[*number*])

search-pattern describes the string for which TXT2STF searches. *search-pattern* is a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or a variable that contains the pattern.

replacement-string is the string that replaces the string identified by *search-pattern*. This value must be a string constant. You can specify the string itself, in a pair of quotation marks (" "), or a variable that contains the string.

number (optional) specifies the number of occurrences of *search-pattern* to be replaced by *replacement-string*. *number* must be a numeric constant. If you omit *number* in the text line, @replace substitutes *replacement-string* for all occurrences of *search-pattern* in the text line.

Example

The following sample statement replaces all dashes in a text line with slashes. With the text line "11-12-90", you will get a When date of "11/12/90", which is in the Agenda default date format.

```
":d:d-:d:d-:d:d"  @replace("-","/",)
                  @date(W)
```

See @date_format for other ways to transform dates into other formats.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	May alter the contents of the current line
Item, category, and note buffers	None
Variables	None
Location in text file	None
Structured file	None

@reset

@reset replaces the text line in memory with the contents of the specified variable, or with the text string you specify.

You can use @reset if you need to convert the same text line in more than one way or to use text from one text line when converting a later text line. To do this, save the complete or trimmed text line with the @set variable. Restore the saved line at a later time by using @reset, and use the variable named in the @set.

Syntax

@reset(*replacement-string*)

replacement-string is the string that replaces the current text line in memory. *replacement-string* must be a string constant. You can specify the string itself, in a pair of quotation marks (" "), or a variable that contains the replacement string.

Example

This example shows one way to convert information from the Lotus Metro® appointment book, extracting each appointment as an item. This example uses the following appointment book information, which has already been printed to an ASCII text file:

```
Monday, November 5, 1990
Book:  MAIN
   6:00a
   7:00
   8:00   R.Ball (Shelley)
   9:00
  10:00   Meeting with Phil and Anne re. sign
  11:00
  12:00p
   1:00   to 3 - Downtown, Bookseller mtg.
  .
  .

Tuesday, November 6, 1990
Book:  MAIN
   6:00a
   7:00
   8:00
   9:00
  10:00   At Large Wkly Wrap-up
  11:00
  12:00p
   1:00   send-off plan review mtg.
  .
  .
```

The following sample definition file uses variable Full Date to assign each item the correct When date and variable Book to assign each item to a category that names the book the item is from.

Notice in this example that @set specifies variable names, such as Date and Bookname, in a pair of quotation marks (" ") because it is simply naming the variables. @reset names the Full Date variable without quotation marks to instruct TXT2STF to use the *contents* of the variable.

```
#Start collecting information for a new item, delete day name
#(for example, Monday) from day header line, and put date
#in variable Date

"^:a+, :a+ :d+, :d:d:d:d"   @start()
                            @trim("^",",",)
                            @set("Date")

#Put name of the book into variable Bookname

"^Book\:"                   @skip(" ",)
                            @set("Bookname")
#Make contents of Bookname a child of category Book

":d+\::d:d[ap ].*:a+"       @custom_category(Bookname,"Book",)
#Trim text, construct date and time in variable Full Date

                            @trim("^",":d+\::d:d",,"Time")
                            @trim("^",":n",N)
                            @item2("$")

                            @equate("Full Date",Date)
                            @append("Full Date"," at ")
                            @append("Full Date",Time)
#Reset current line to contents of variable Full Date,
#create a When date using the "new" line, and copy
#current item/category/date info to structured file

                            @reset(Full Date)
                            @date(W)
                            @end()
```

When TXT2STF uses this definition file to convert the sample Metro file using the sample definition file, it creates five items for your Agenda file, one for each appointment listed in the file.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	Replaces the current line
Item, category, and note buffers	None
Variables	None
Location in text file	None
Structured file	None

@set

@set stores the current text line in the specified variable.

This lets you use the current text line in more than one way. You can save the text line by using @set, and then trim or convert the line however you want. Then, to use the original text line in another way, use @reset to restore the full text line. The @reset can be in a different statement than @set.

Tip @set stores the entire text line in a variable. To save a portion of the line, use one of the trimming commands to remove unwanted text before you use @set.

Syntax

@set("*variable-name*")

variable-name is the variable in which the text line is stored. *variable-name* must be a variable name string constant.

See @reset for an example of using @set.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	None
Variables	Copies the contents of the current text line to the specified variable
Location in text file	None
Structured file	None

@skip

@skip divides the current text line into separate values using the separator pattern you specify. It then deletes the specified value from the text line.

This lets you remove unwanted text from the text line before you convert the line into an item, category, or note. TXT2STF uses the entire text line minus the unwanted text, leaving no spaces where the text was deleted.

Tip If you want to use a text line in several ways, consider using @trim and not @skip (since @trim saves deleted text in variables for later use). If you use @skip, you can use @set to save the original line in a variable before executing @skip. To restore the original line at a later time, use @reset.

To delete text, @skip first divides the line into values using the specified separator pattern. It then assigns a number to each value, assigning the number 1 (one) to the first (leftmost) value, the number 2 to the second, and so forth. Finally, @skip deletes the value whose number you identify in @skip. For example, you delete the second value from the text line by telling @skip to delete value number 2.

Each @skip removes one value from the text line. To remove more than one value from a line, you must include more than one @skip in a statement. Each @skip divides the line and renumbers the resulting values based on the specified separator pattern.

Syntax

@skip(*separator-pattern,position-number*)

separator-pattern describes the string that TXT2STF uses to separate the text line into separate values. This pattern specifies where a given value ends and another begins. The separator pattern is included at the end of each string where it is found. *separator-pattern* must be a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or a variable that contains the pattern.

position-number identifies, by number, the value to delete. *position-number* must be a numeric constant.

Example

This example shows one way to convert the following text line to create a category swind:pkyn, with a parent of ID.

```
SYSID:swind:USERID:pkyn
```

The sample statement shown below is executed when TXT2STF finds the string SYSID: at the start of a text line. The first @skip uses the colon (:), as specified by the pattern "\:", to separate the text line into four values (SYSID:, swind:, USERID:, and pkyn). The same @skip then deletes the first value (SYSID:) from the text line.

The second @skip again uses the colon (:) to separate the text line, which results in *three* values (swind:, USERID:, and pkyn). The same @skip then deletes the second value (USERID:) from the text line. Then @category makes a category from the current text line, which now contains only swind:pkyn.

```
"^SYSID\:"          @skip("\:",1)
                    @skip("\:",2)
                    @category("$","ID",)
```

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	May delete text from the start of the current line
Item, category, and note buffers	None
Variables	None
Location in text file	None
Structured file	None

@skip_lines

@skip_lines lets you skip over unneeded lines of text. It copies lines into memory and discards them until it finds the termination point you specify.

Syntax

@skip_lines(*termination-point*)

termination-point describes the string or numeric constant that specifies where to stop skipping lines. *termination-point* can be a pattern constant, or a numeric constant.

- A pattern constant (or a variable that contains the pattern) causes @skip_lines to skip to the beginning of the first character that matches the pattern. @skip_lines skips over all intervening text lines.

- A numeric constant causes @skip_lines to skip the specified number of lines from the current line. For example, specifying 1 (one) skips to the next line in the text file, which becomes the current line.

Specifying a pattern as the termination point for @skip_lines is particularly useful when you know the pattern for:

- The start of the text you want to convert

  In this case, the line for which you are searching becomes the current text line in memory. This means that you can include the commands (@item2, @note, and so forth) for the current line in the same statement.

- The end of the text you want to skip over

  In this case, the line you wish to convert is the next line after the line with the pattern. To skip to this next line, use @skip_lines(1).

Note After TXT2STF matches the @skip_lines termination pattern in a text line, the entire line including the matching string is available to the next command. If @skip_lines is the last command in the statement, the entire matching text line is available to match the pattern at the start of the next statement.

Example

This example shows one way to convert the following text file:

```
Message: 123456889766787
WEEKLY SALES UPDATE: Week of 11/16/90
Northeast/Mid-Atlantic:
October goal of 70,000 met due to success of CopyCat and Stratford copiers;
regional sales meeting Monday 12/3/90.
```

The following sample definition file statement is executed when TXT2STF finds the string Message: at the start of a text line. @skip_lines causes TXT2STF to skip down to the next line. @item2 creates an item from the current line, and @note creates a note from the remaining text lines, as specified by the predefined variable EOF.

```
"^Message\:"        @skip_lines(1)
                    @item2("$")
                    @note(EOF)
                    @end()
```

When the resulting structured file is imported, it adds the item WEEKLY SALES UPDATE: Week of 11/16/90. It adds the remaining text as an item note.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	May change the current line to be a later line in the text file.
Item, category, and note buffers	None
Variables	None
Location in text file	If @skip_lines specifies a number, TXT2STF points to the start of the new current line. If @skip_lines specifies a termination pattern, TXT2STF points to the first character that matches the pattern in the text line.
Structured file	None

@start

@start initializes the item, category, and note buffers.

Execute @start before any commands that convert information you want to copy to the structured file, so that TXT2STF has buffers in which to keep the converted information. @start is typically the first command in the first statement executed in a definition file.

You can include @start anywhere else in the definition file where you want to initialize buffers *without* copying their contents to the structured file.

Tip To copy the contents of the item, category, and note buffers to the structured file, use @end.

Syntax

start()

@start has no arguments, but the parentheses are required.

Example

The following sample statement starts converting information whenever it finds the string To: at the beginning of a line.

```
"^To\:"            @start()
```

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Initializes these buffers
Variables	None
Location in text file	None
Structured file	None

@strip

@strip deletes from the text line one or more occurrences of the string you identify.

Syntax

@strip(*strip-pattern*,[*number*])

strip-pattern describes the string to be removed from the text line. *strip-pattern* must be a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or a variable that contains the pattern.

number (optional) specifies the number of occurrences of the strip string to be deleted from the text line. *number* must be a numeric constant. If you omit the number, @strip deletes all the occurrences of the strip string.

Example

This example shows one way to convert the following text file:

```
Part-23-17-A
200/250 filter for Empress humidifier (model #230), with pre-filter attach-
ment for Gen Clean (hospital model #1230/1231)
```

The following sample definition file statement executes when TXT2STF finds a line starting with the string Part-, which is the word Part followed by a hyphen (-). @strip removes all the hyphens (-) from the current text line, @set puts the resulting text into the variable Catname, @custom_category makes the contents of Catname into a child of the category Part. @skip_lines advances TXT2STF to the next text line. @category_note makes this text line and the remaining text lines in the text file into a note for the category, as specified by the predefined variable EOF. @end copies the categories to the structured file.

```
"^Part-"            @strip("-",)
                    @set("Catname")
                    @custom_category(catname,"Component",)
                    @skip_lines(1)
                    @category_note(EOF)
                    @end()
```

When the resulting structured file is imported into Agenda, Component is added as a child of the current category, as specified when you use the Agenda **File Transfer Import** command. The category Part2317A is added as a child of Part. The category note is added to category Part2317A.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	Deletes from the current line one or more occurrences of the identified string
Item, category, and note buffers	None
Variables	None
Location in text file	None
Structured file	None

@table

@table converts information in a table into items and categories. For example, you can use @table to convert a table of information from a Lotus 1-2-3 worksheet that has been printed to a text file.

@table starts processing the table with the current line. TXT2STF executes @table once for each line in the table. To specify where the table ends, you include a termination pattern that describes a string on the line immediately *after* the table.

To convert tabular information, @table separates the information into columns, using the separator pattern you specify. It then processes each line in the table, converting the text in specific columns into items, categories, and category values. For example, @table can make text in the first column of each line into an item, text in the second column into a category, and so forth.

@table assumes that the first line in the table provides header or label information for the column, and converts the columns in that line into categories. When @table processes the remaining lines in the table, it assigns items to the category that heads the item column. When @table converts text in a category column in the table:

- For a standard category, @table makes the text a child of the category that heads the category column.
- For a date, numeric, or unindexed category, @table makes the text a value (for example a date) for the category that heads the category column.

Note @table adds an item and/or categories to the structured file after converting each text line. To do this, @table internally executes an @start before and an @end after converting each text line.

Syntax

@table(*separator-pattern,termination-pattern,col-number,type*
[,col-number,type,...])

separator-pattern describes the string that TXT2STF uses to divide the table. This pattern specifies where a given value ends and another begins. *separator-pattern* must be a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

termination-pattern describes the string that ends the table. *termination-pattern* must be a pattern constant. You can specify the pattern itself, in a pair of quotation marks (" "), or the name of a variable that contains the pattern.

The termination pattern must describe a string on the line that immediately follows the last line of the table. If the text file does not include the pattern, TXT2STF processes each line in the text file up through and including the last line in the file.

col-number is a numeric constant that identifies a column to process into an item, category, and so forth. To determine the number for a column, count the columns in the text file from left to right, using 1 for the first column, 2 for the second column, and so forth. When counting, include all columns, including columns you intend for TXT2STF to ignore.

TXT2STF only converts and copies to the structured file those columns that @table identifies by number. For each position number in @table, you must specify a type. The column number and type combinations *must* all be on the same line as the rest of the @table command.

type determines whether information in the column becomes an item or a category. For categories, this value also identifies the type of category to create. *type* must be a type constant. You can specify "date", "exclusive", "numeric", "standard" or "unindexed" for categories, or "item" for items.

Example

This example shows one way to convert the following tabular information:

```
Table 1:
Events              Start Date   End Date   COG    Revenues
Post-XMAS R         12/26/89     12/30/90   2800    3200
Spring Fling R      04/01/90     04/15/90   4500    8200
Met Sidewlk Days    05/02/90     05/09/90   4200   75001
-----------------------------------------------
```

The following sample statement finds the string Table at the start of a line. @skip_lines skips TXT2STF down a line, where @table starts processing the tabular text. @table divides columns by searching for two or more space characters (as specified by the pattern ":   "+) @table makes the first table column the item column, the third column a date, and the fifth column a numeric category. @table ignores the second and fourth column.

```
"^Table"    @skip_lines(1)
            @table(":  +","---+",1,"item",3,"date",5,"numeric")
```

When the resulting structured file is imported, Events, End Date, and Revenues are added as categories under the current category specified when you use the Agenda **File Transfer Import** command. Items created from the first column of the table, such as Post-XMAS R, are assigned to the Events category. Each item is also assigned to the End Date category by the date value on the same row as the item, and to the Revenue category by the numeric value on the same row.

See "Converting a Table of Information" in Chapter 3 for another example of using @table.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	When TXT2STF finishes processing the table, the current line is the line that matched the termination pattern.
Item, category, and note buffers	Adds items and categories to TXT2STF buffers.
Variables	None
Location in text file	When TXT2STF finishes processing the table, the location in the text file is the beginning of the line that matched the termination pattern.
Structured file	Adds to the structured file after processing each line in the table, adding the item and/or categories created from the line.

@trim

@trim deletes text from the text line and optionally puts the deleted text into a variable.

This lets you remove unwanted text from the text line before you convert the line to be an item, category, or note. TXT2STF then uses the entire text line minus the unwanted text (leaving no spaces where the text was deleted).

You can save the deleted portions of the line in a variable. This lets you use the deleted text in a later command or statement.

To delete text, @trim locates the specified start and end patterns in the text line. It then deletes the text between the patterns. It also deletes the pattern text if you instruct it to.

Each @trim removes one value from the current line. To remove more than one value from a line before you use it, you must include more than one @trim in a statement.

Syntax

@trim(*start-pattern,end-pattern,*[*N*][,"*variable-name*"])

start-pattern and *end-pattern* identify the text to be deleted from the current line by giving the starting and ending points for deletion. *start-pattern* describes the string where deletion should start and *end-pattern* the string where deletion should end. Both patterns must be pattern constants. You can specify the patterns in a pair of quotation marks (" "), or the name of the variables containing the strings.

N (optional) specifies that the deletion is *not* inclusive. *N* must be a logical constant. If you specify N, @trim deletes the text between the specified strings without deleting the strings. If you omit N, @trim deletes from *start-string* through *end-string*, including the strings.

variable-name (optional) identifies a variable into which TXT2STF puts the string trimmed from the current text line. *variable-name* must be a variable name string constant.

Example

This example shows one way to convert the following sample text file to get two categories, Robin McAfee and Anand Mohanty, with the address for each person a category note for the category created for the person.

```
Name:      Robin McAfee
Address:   1234 Main Street
           Anytown, IL
           61820

Name:      Anand Mohanty
Address:   85 Broadway
           South City, MA
           02042
```

The sample definition file shown below converts this text file. The first statement in the definition file is executed when TXT2STF finds the string Name: at the start of a text line. Then, @trim deletes from the line everything from the start of the line (specified by the pattern "^") up through the block of spaces that separates the string Name: from the actual name (for example, Robin McAfee). @category makes the remaining text line (for example, Robin McAfee) a child of the Student Name category.

The second statement is executed when TXT2STF finds the string Address: at the start of a text line. @category_note creates a note that starts with the current text line and continues up to (but not including) the next name in the file (which is on a line beginning with the string Name:). @end copies the current category and category note to the structured file.

```
"^Name\:"          @start()
                   @trim("^","  +",)
                   @category("$","Student Name",)

"^Address\:"       @category_note("^Name\:")
                   @end()
```

When the resulting structured file is imported, Student Name is added as a child of the current category specified when you use the Agenda **File Transfer Import** command.

For an example of how @trim puts a string into a variable, see the sample definition file shown at the end of "Writing the Definition File Patterns and Commands" in Chapter 1.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	Deletes text from the current line
Item, category, and note buffers	None
Variables	Can create new variables or replace the contents of existing ones
Location in text file	None
Structured file	None

@unindexed

@unindexed lets you create assignments to unindexed categories. @unindexed specifies the category name and optionally specifies a text value.

When @end copies the category specified by @unindexed to the structured file, it also copies the item in the item buffer to the structured file. When the structured file is imported, Agenda uses the text value specified by @unindexed to assign the item to the unindexed category.

If the unindexed category already exists, Agenda retains the existing category and does not create a duplicate category.

Caution If there is no item in the item buffer when the next @end is executed, then on import Agenda creates the category but does not import the text value specified by @unindexed since there is no item to assign to the unindexed category.

Syntax

@unindexed(*category*[,*value*])

category is the name of the unindexed category. This value must be a string constant. You can specify the category name, in a pair of quotation marks (" "), or a previously defined variable that contains the category name.

value (optional) is the text value used in the assignment. This value must be a string constant. You can specify either the value itself, in a pair of quotation marks (" "), or a previously defined variable that contains the value. If you omit *value*, TXT2STF uses the entire current text line in the assignment.

Example

The following sample definition file statement creates an assignment to an unindexed category, Label, with the text value A1. @end copies the category to the structured file.

```
START               @start()
                    @unindexed("Label","A1")
                    @end()
```

When the resulting text file is imported into Agenda, the unindexed category Label is imported as a child of the current category specified when you use the Agenda **File Transfer Import** command. If an item is added at the same time, the item is assigned to the Label category by means of the text value A1. If no item is added with the unindexed category, Agenda ignores the text value A1.

Changes made by this command

Elements that commands can change	*Changes made by this command*
Current line in memory	None
Item, category, and note buffers	Adds an unindexed category to TXT2STF category buffers.
Variables	None
Location in text file	None
Structured file	When the next @end is executed, the unindexed category is added to the structured file. If an item exists in the item buffer at that time, TXT2STF also adds the item to the structured file and assigns it to the unindexed category using the value specified by @unindexed. Otherwise, there is no item to assign to the category, and so the category is imported into Agenda but the associated text value is ignored.

Chapter 6
Converting and Importing Text

Before you import text into Agenda, you can convert the text so that Agenda knows which text to use as items, categories, and notes. To do this, you run the Agenda TXT2STF utility to convert the text file to a structured file. Then, you return to Agenda and import the structured file using the Agenda **File Transfer Import** command.

In this Chapter

This chapter describes how to convert and import text by:

- Explaining how to use the TXT2STF utility and its options to convert a text file to a structured file
- Describing the **File Transfer Import** command and additional options you can specify to control the importing of structured files

For information about structured files, see Appendix B. For general information about converting files to be imported into Agenda, see Chapter 23 in the *User's Guide.*

Note If you plan to convert a file from an external source (for example, a word processing document), make sure it is an ASCII text file. Refer to the manuals for the software you use for information on how to create an ASCII text file.

Running TXT2STF

You run the TXT2STF utility to convert a text file to a structured file. You run this utility outside of Agenda, as follows:

1. Make sure you are in the Agenda program directory.

2. At the operating system prompt, type the TXT2STF command, following the syntax described in "The TXT2STF Command," below.

   TXT2STF displays the TXT2STF copyright box and converts the text file to a structured file with the same name and the extension .STF. TXT2STF converts the *contents* of the text file without modifying or deleting the original text file.

The TXT2STF Command

The TXT2STF command has the following syntax:

Syntax
TXT2STF *textfile.ext* [*option(s)*]

textfile.ext is the name of the text file you want to convert. For example, if the file name is MEMO.DOC, specify MEMO.DOC for *textfile.ext*. The file must be an ASCII text file.

options provide additional information to TXT2STF. You can include the following types of options in the TXT2STF command:

- File conversion options give TXT2STF information about how to convert the text file. These options are discussed in "File Conversion Options" later in this chapter.
- Debugging options tell TXT2STF to return debugging information when it converts the text file. For more information about these options, see "Including Debugging Options" in Chapter 7.

File Conversion Options

The following TXT2STF options provide additional information about how to convert a text file:

Option	*Description*
/C	Tells TXT2STF to be case sensitive when matching literal alphabetic characters specified in patterns.
/D	Specifies a definition file. A definition file tells TXT2STF how to interpret a text file. For more information, see "Specifying a Definition File" later in this chapter.
/O	Specifies another name for the structured file. For more information, see "Specifying a Different Name for the Structured File" later in this chapter.
/S	Specifies an alternate separator character for paragraphs in the text file. For more information, see "Specifying an Alternate Separator Character in the Text File" later in this chapter.

Note TXT2STF ignores the /C option when matching a character specified as a range or group in brackets ([ ]) and matches those characters exactly as specified in the range or group. (See Chapter 4.)

Follow these guidelines when you include options in the TXT2STF command:

- You must use either a slash (/) or a hyphen (-) before each option.

  Correct:

  ```
  C:\AGENDA> txt2stf events.txt /o=calendar
  ```

  Incorrect:

  ```
  C:\AGENDA> txt2stf events.txt  o=calendar
  ```

- You can type options in uppercase or lowercase.
- You can include options in any order.

How TXT2STF Converts Text Files

If you run the TXT2STF utility without any file conversion options, TXT2STF converts each paragraph into an item, putting any text that does not fit in the item into a note for the item. The conversion works as follows:

- TXT2STF begins at the beginning of the text file, taking text for the first item. When TXT2STF reaches a double carriage return (a blank line), it ends the item and starts a new item.
- If there is no double carriage return before the 350th character, Agenda ends the item at the 350th character (the maximum length for an item) and puts the remaining text into a note.

For more information about how TXT2STF processes a text file, see Chapter 2.

Specifying a Definition File to Use with a Text File

The /D option of the TXT2STF command specifies a definition file that tells TXT2STF how to convert the text file. You must specify a definition file that already exists. For information about how to create a definition file, see Chapter 2.

The /D option has the following syntax:

Syntax
/D=*definition-file*

definition-file is the name of the definition file to use. You do not need to specify an extension: TXT2STF automatically uses the extension .STF.

Example

```
C:\AGENDA> txt2stf memo.all /d=memconv2
```

This example formats a text file called MEMO.ALL using the definition file MEMCONV1.DEF (TXT2STF automatically uses the extension .DEF.

Example

```
C:\AGENDA> txt2stf news.txt /d=\def\wirenews.def
```

In this example, you specify a path since the definition file is not in the current directory. The definition file that TXT2STF uses in the above example is WIRENEWS.DEF in the DEF subdirectory.

Specifying a Different Name for the Structured File

The /O option of the TXT2STF command specifies another name for the output (structured) file.

By default, TXT2STF converts the text file to a structured file with the *same* name as the text file and the extension .STF; for example, MEMO.STF. If this file already contains information, TXT2STF overwrites the original information in that file when it puts the newly formatted text into the structured file.

Use the /O option if you want the structured file to have a different name. You might do this, for example, to save each structured file TXT2STF creates for a given text file.

This is especially useful during debugging, when you run TXT2STF each time you change the definition file. If you specify a different structured file each time you run TXT2STF, you can compare the results of each change you make to the definition file. For more information about debugging a definition file, see Chapter 7.

The /O option has the following syntax:

Syntax

/O=*structured-file*

structured-file is the name of the structured file in which TXT2STF puts formatted text. Do not specify an extension; TXT2STF automatically gives the output file the extension .STF.

Example

```
C:\AGENDA> txt2stf memo.all /o=run1
```

In this example, TXT2STF puts formatted text in the structured file RUN1.STF, even though the text file is named MEMO.ALL.

You can also specify a path with the /O option.

Example

```
C:\AGENDA> txt2stf events.txt /o=\stf\calendar
```

Specifying an Alternate Separator Character for a Text File

The /S option of the TXT2STF command specifies an alternate separator character for paragraphs in the text file. When TXT2STF encounters *two* consecutive paragraph separator characters in a text file, TXT2STF begins a new paragraph.

The default separator character is the carriage return (ASCII decimal 13). You can use the /S option to specify a different separator character if the paragraphs in your text file are separated with a different character.

The /S option has the following syntax:

Syntax
/S=*separator-character*

separator-character specifies the character TXT2STF uses as the paragraph separator character. You can use the following as separator characters with the /S option:

- Any keyboard character, such as a comma (,) or period (.)
- A three-digit decimal ASCII code preceded by a backslash (\)

Note To use a backslash (\) as a separator character, you must enter two backslash characters (s=\\).

Examples

```
C:\AGENDA> txt2stf news.txt /s=,
```

The preceding example specifies the comma (,) as the separator character. Each time TXT2STF encounters *two* commas in a row, TXT2STF starts a new paragraph.

```
C:\AGENDA> txt2stf news.txt /s=\010
```

The preceding example specifies the three-digit ASCII decimal code for a linefeed as the separator character.

Converting More Than One Text File

You might want to convert more than one text file into a single structured file. For example, you might want to format a group of files that each contains information about clients.

You can convert all the text files at one time if the files have similar names. In this case, you can use a wild card character to specify a group of files.

If you want to put the converted files into a single structured file, use the /O option when you run TXT2STF. If you omit the /O, TXT2STF creates a separate structured file for each text file it converts.

Example

```
C:\AGENDA> txt2stf *.inf /o=allinf
```

In this example, TXT2STF converts all files that have an extension of .INF, and puts the results in a single structured file named ALLINF.STF.

For more information about wildcard characters, see your operating system manual.

Importing the Structured File

After you create a structured file, you can import it into an Agenda file. Agenda uses the information in the structured file to create items, categories, and notes.

Note Before importing a structured file, make sure you have thoroughly debugged the definition file that produced the structured file, as described in Chapter 7.

To import a structured file:

1. Start Agenda and open a file.
2. Press F10 (MENU) and select **File Transfer Import**.

   Agenda displays the Import Structured File box, which lets you specify the type of information you want Agenda to import from the structured file.
3. Complete the Import Structured File settings and press ENTER.

Agenda imports items and categories from the structured file, and assigns the items to categories, according to the choices you specify in the Import Structured File settings. You can specify whether you want to import everything in the structured file, or only information that is new in the structured file since the last time you imported the same file. You also use the Import Structured File box to specify the types of information to import.

For details about the **File Transfer Import** command and the Import Structured File box, see Chapter 23 in the *User's Guide.*

You can import any or all of the following types of information:

- Items along with notes
- Category assignments (assignment of items to existing categories)
- New categories referenced in assignments
- New categories that are explicitly created in the structured file (created as independent categories and assigned to blank items on import)

You also can use Import Structured File settings to specify

- A category to which all imported items are assigned
- The criteria Agenda should use to determine whether categories in the structured file match categories in the Agenda file
- Whether Agenda should eliminate single carriage returns when it imports notes

Note After you import a structured file, Agenda tests items against all conditions in the file. If you have a large structured file and a complex Agenda file, this can take a significant amount of time. You may want to disable conditions before you import the structured file. (See Chapter 19 in the *User's Guide.*)

Chapter 7
Debugging a Definition File

Writing a definition file is like writing a computer program; you may need to make some changes before it works the way you want. For example, the first time you use the definition file, you might discover that TXT2STF converts your text file in a way different from what you want. You might find out that it ignores certain items, categories, or notes that you want converted.

The process of using a definition file, evaluating the results, and changing the definition file to work the way you want is called debugging.

In this Chapter

This chapter describes procedures and strategies for debugging a definition file, and provides additional information to help you write and debug definition files.

This chapter describes

- Procedures for debugging a definition file
- Testing a definition file
- Examining the converted information in the structured file
- Typical problems encountered during debugging and possible solutions
- Tips and techniques for writing better definition files

Procedures for Debugging a Definition File

To debug a definition file, you typically perform the following general procedures:

1. Test the definition file by running TXT2STF to convert a sample text file.
2. Look at the resulting structured file to examine the converted information that TXT2STF creates. Also note error messages if TXT2STF displays any. If the results are not what you want, perform Step 3.
3. Modify the definition file. When you think you've corrected the problems in the definition file, return to Step 1.

This chapter describes how to test the definition file and look at the resulting structured file. To modify a definition file, see "Writing the Definition File Patterns and Commands" in Chapter 2.

You perform the above series of steps, modifying the definition file and checking the results, until the definition file consistently returns satisfactory results. Since TXT2STF does not modify your text file, you can run TXT2STF to convert your text file as many times as necessary.

You may find it helpful to start by running TXT2STF on a small sample version of your text file. Then, when the definition file correctly converts information in that sample, you can test a more complicated text file and evaluate the results.

You are finished debugging your definition file when TXT2STF runs without displaying error messages and the structured file contains the information you want. At this point, you can import the structured file into Agenda. If you are developing the definition file for other users, you can distribute the definition file to them when you finish debugging it.

Testing a Definition File

To determine whether a definition file works the way you want, you need to try using it with TXT2STF. To name your definition file, include the /D option in the TXT2STF command. For more information about running TXT2STF, see Chapter 6.

Naming a Structured File

TXT2STF puts the text it converts into a structured file. By default, TXT2STF creates a structured file with the same name as the text file and the extension .STF. For example, TXT2STF creates a structured file named MEMO.STF for a text file named MEMO.DOC.

When you debug a definition file, you may want to name a new structured file each time you change the definition file and run TXT2STF. This way, you can compare the results of each change you make to the definition file.

To specify a structured file, include the /O option in the TXT2STF command. For more information, see "Specifying a Different Name for the Structured File" in Chapter 6.

Including Debugging Options

TXT2STF provides several debugging options to help you see how TXT2STF uses your definition file to convert the text file. These options tell TXT2STF to display information and return messages about its activities while it converts your text file.

You can specify any or all of the following debugging options when you run TXT2STF:

Option	*With this option, TXT2STF*
/A	Uses all debugging options (/L, /M, /T, /U, and /V)
/L	Displays the current text line when TXT2STF loads or modifies it
/M	Displays the text strings that match patterns that start definition file statements
/T	Displays each definition file command when it's executed
/U	Displays each category that TXT2STF creates
/V	Displays the contents of each variable every time it's modified

For information about other TXT2STF command-line options, see Chapter 6.

Redirecting Error Messages to a File

By default, TXT2STF displays error messages on your screen. So that you can examine these messages later, you can redirect them to a file by using the redirection symbol defined by your operating system (for example, > *filename*) when you run TXT2STF.

Example

```
C:\AGENDA> txt2stf xfer.txt /d=xfer /a > msgs
```

In this example, TXT2STF converts text file XFER.TXT using definition file XFER.DEF. TXT2STF puts debugging information in the file MSGS.

Examining the Converted Information

To see how TXT2STF uses the definition file to convert the text file, you can examine the results. To do this you can

- Examine the structured file that TXT2STF creates
- Examine the debugging information returned by TXT2STF
- Import the converted information in the structured file into Agenda and examine it

Examining the Structured File

You examine the contents of the structured file to determine if the definition file made the correct text into items, categories, and notes.

When you examine a structured file, you see that the structured file contains these text elements embedded in special tags that Agenda uses to import the text correctly. For example, the {C} tag indicates text to be imported as a category. For information about how to read the tags in structured files, see Appendix B.

You can examine a structured file by using any text processor that displays text files. If you redirect TXT2STF debugging information to a file, you also can examine the file using a text processor that displays ASCII files.

If you expect either file to be relatively short, you can use your operating system type/display command to quickly view the contents of the file on your screen. This command typically, however, does not let you page back and forth through the contents of a file in the way that a text processor does.

Example

```
C:\AGENDA> type outfile.stf | more
```

The above sample command displays the contents of structured file OUTFILE.STF.

If text in the structured file looks different from what you want, you need to modify the definition file based on the problems you identify in the structured file. Consult "Typical Problems and Solutions" later in this chapter for some ideas.

Importing the Converted Information

When you are satisfied with the contents of the structured file, you can import it into your Agenda file.

Caution To keep your Agenda file from containing incorrectly converted information, you should wait to import information until after the contents of the structured file indicate that you have thoroughly debugged the definition file.

To import the converted information into Agenda, see "Importing a Structured File" in Chapter 6.

Typical Problems and Solutions

There are three major types of errors that can occur when you run TXT2STF with definition files:

- Syntax errors in statements
- Pattern matching problems
- Converting information incorrectly from a text line

If you have a syntax error in any statements, TXT2STF displays an error message. For explanations of TXT2STF error messages, see Appendix D.

Other problems in your definition file, such as pattern matching problems, typically cause the structured file to contain incorrect information. For example, lines from your text file might be omitted from your structured file, or might be converted incorrectly.

Testing for Pattern Matching

Some reasons why lines from your text file might not be copied to the structured file include:

- The line is not converted, and thus not copied, because it does not match the intended pattern.
- The line matches the pattern, but is never tested against that pattern.

To find out if a text line matches the intended pattern, create a definition file that includes only the statement intended to match the line. Run TXT2STF using this single-statement definition file, and include the /L and /M debugging options. (See "Including Debugging Options" earlier in this chapter.)

- /M shows you exactly where each text line matches a pattern in a statement.
- /L displays the text line if TXT2STF matches it, and every time TXT2STF changes it.

If the text line does *not* match the intended statement, you need to modify the pattern in the statement.

If the text line *does* match the statement, consider whether:

- An earlier statement in the original definition file matches the text line.

  In this case, TXT2STF uses the earlier statement to convert the text line and never uses the intended statement. Remember that once TXT2STF finds a pattern that matches the text line, it does not search any further.

  Try refining the patterns or changing the order of statements in the definition file. Make sure that statements that include the *general* patterns occur after statements with *specific* patterns in the definition file. For more information about statement order in definition files, see "Tips and Techniques" later in this chapter.

  Also check @skip_lines commands in previous definition file statements to see whether they cause TXT2STF to skip over the line you want to convert. This occurs if @skip_lines matches a pattern on the line you want to convert or on a later line in the text file. To fix this problem, you can generally put the converting commands in the same statement as the @skip_lines command. The /L option can tell you where the @skip_lines command skipped to.

- The commands in the intended statement convert information differently from what you want.

  In this case, you can examine the information returned by the /L debugging option to see how the text line is modified by commands. For additional solutions, see "Testing How Information Is Converted."

To determine if either or both of these situations apply, you can examine the original definition file. Also, consider running TXT2STF again, using the original definition file and the /L and /M options.

Testing How Information Is Converted

Even though a text line matches the intended statement, the commands in the statement may not convert the line in the way you want. There are several reasons why this might occur.

To find out why information is converted incorrectly, you need to analyze how each command in the statement converts the text. For each text line that is not converted how you want, run TXT2STF using a definition file that includes only the statement intended to convert the text line. Also include debugging options in the TXT2STF command. (See "Including Debugging Options" earlier in this chapter.)

- To focus on how the text line is formatted, include the /L /T, and /V options in the TXT2STF command.

  /T displays each command when TXT2STF executes it. /L displays the contents of the text line each time TXT2STF changes it. /V displays the contents of each variable whenever TXT2STF modifies the variable. This is important if you use variable commands, such as @grab, to convert text lines.

- To see how TXT2STF creates categories, consider turning on all debugging options by including the /A option in the TXT2STF command.

  /A provides the most complete debugging information. In early stages of debugging, however, this option may provide more information than you need.

Typical reasons why a text line is converted differently from what you want:

- The line might be altered incorrectly by a trimming command.

  For example, the command might trim text that you want to convert. The /L option helps you find this problem by displaying the current line after each trimming command changes it.

- Commands such as @grab and @trim, which create variables, might incorrectly put values into variables.

  The /V option helps you find this problem by displaying variables each time they are created or modified. If this occurs, it is likely that you need to modify an argument in the command, such as the separator pattern in @grab or the start and end patterns in trim.

- An @skip_lines command might make the text line unavailable.

  A @skip_lines in the current definition file statement might skip over the text line you want to convert. In this case, the line is unavailable for conversion. Use the /L option to see which line @skip_lines skips to.

- Patterns included as parameters within commands might require adjustment.

  For example, you can specify a pattern to tell TXT2STF where an item or a note ends. If these patterns are incorrect, TXT2STF converts different portions of text than the portions you want.

Tips and Techniques

This section lists suggestions to help you write better definition files.

Writing Patterns

In general, it is a good idea to write patterns that explicitly describe the text to be matched. Include in the pattern as many actual characters from the string as you can. Specifying explicit patterns increases the likelihood that the correct text lines will match the pattern.

Even when using a generic pattern, try to be as exact as possible. For example, consider the generic pattern used in the sample definition file constructed in Chapter 1. That pattern, "^.*\:", matches lines starting with either To: and From: and thereby enables you to write one statement to handle two different text lines.

However, this pattern really matches an entire text line, as specified by the combination caret (^), period (.), and asterisk (*). Then, TXT2STF searches the line backward, character by character, to find a colon (:). (The backslash (\) tells TXT2STF to match the colon (:) as a text character and to ignore its special meaning in match-control characters.) The matching string consists of the entire line up to the *last* colon (:) in the line, which is acceptable for the example in Chapter 1, but could cause problems for text lines that include more than one colon (:).

A different pattern that also matches To: and From: is "^[^:]*\:", which matches an entire text line up through the *first* colon (:). This pattern starts at the beginning of a text line, and matches each character that is *not* a colon (:). If it finds a colon (:) in the line, the text from the start of the line up to the colon (:) matches the pattern. If the line does not contain a colon (:), the line does not match the pattern. (See "How TXT2STF Compares Patterns with Strings" in Chapter 4.)

Other tips for writing patterns:

- Analyze negative patterns carefully.

  A negative pattern uses the tilde (~) to match text that *is not* the pattern. Typically, many text lines can match a negative pattern. For example, the pattern "^~Tuesday" matches any line that does not start with the word Tuesday.

  When you need to use a negative pattern, anchor the pattern to a fixed characteristic of the string, such as the start or end of the line. This makes the pattern more likely to match only those text strings for which it is intended.

  When you use a statement starting with a negative pattern, make sure the statement is located toward the end of the definition file. For more information about negative patterns, see the discussion of the tilde character in Chapter 4.

- If you need to use a complicated pattern more than once as an argument in definition file commands, consider using @equate to assign the pattern to a variable. This way you need to debug the pattern only one time. You can then use the variable that contains the pattern in every command where you need the pattern. Furthermore, if you need that pattern in another definition file, you can copy the @equate command into that definition file and use the variable there.

Ordering Statements

The order of statements can significantly affects how the definition file converts text lines in the file.

This is because the definition file does *not* operate like a procedural program, but more like a list of different actions to take when TXT2STF finds specific text strings in a text file. Each definition file statement describes an action to take. When TXT2STF matches a text line with a definition file statement, it executes the commands in the matching statement.

Each definition file statement begins with a pattern that describes the line to be converted by the statement. TXT2STF converts a text file line by line, one text line after the other. For each new text line to be converted, TXT2STF compares the line to statements in the definition file, starting from the beginning of the definition file, searching for a pattern that the text line matches.

For this reason, it is important to organize the statements so that the first pattern matched by a text line is the correct pattern. In general, it is a good idea to put specific patterns, which exactly match particular lines, early in the file. This way, text lines for which specific patterns exist match their patterns right away.

Sometimes it is useful to have a statement match more than one line. For example, you might want to match both the To and From lines in an electronic mail file and use the same commands to convert both lines into categories.

Statements that are generic enough to match more than one lines should be put toward the end of the definition file. Statements with specific patterns should be put toward the start of the definition file. This way TXT2STF compares a text line with all specific patterns, which are early in the definition file, before comparing it with generic patterns.

Skipping Over Text

When converting text in a text file, you might need to skip over text on a specific text line. You also might need to skip over entire lines of text.

If you need to selectively convert portions of a single text line, use trimming commands such as @trim, @skip, @strip, and @replace. @trim is especially useful, since it saves the text it trims in variables so you can use the trimmed text later in the file conversion process.

Put the trimming commands for a text line before any commands that convert the line into item text, a category, or a note.

If you need to construct a single item, category, or note from text that is separated by intervening material in the text file, construct your statement as described below.

- Use the pattern to locate the start of the item, category, or note, and use the appropriate command to convert it.

- Next use the @skip_lines command to skip over the irrelevant material. You can delete several lines of unwanted text in the middle of an item or note.
- Finally, use another @item2 or @note command to append the rest of the text to an item or note.

For example, Figure 7-1 shows a sample text file.

```
Name:                Lynne George
Req:                 Job Posting FA-1290
Address:             10 Holstream Rd.
                     Jamaica Plain, MA 02130
D.O.B:               4-7-58
Social Security:     123-45-6789
Degree Program:      Fashion Design
Work Experience:     A.R.T. costumes
Comments:            Lynne has taken two semesters
                     of Pattern Making
```

Figure 7-1 *Sample text file*

Figure 7-2 shows a definition file that makes the name into an item, starts a note at the Req line, and then skips the information until D.O.B. It puts information until Degree into the note, then makes the degree program a category. The text following the degree is appended to the note that was started above.

```
#Convert student information to .STF format

#Skip over "Name:" and make item from name on the line

"^Name\:"         @start()
                  @skip("\: +",1)
                  @item2("$")

#Start note after skipping "Req:", skip over address,
#and add DOB to Degree into note

"^Req\:"          @skip("\: +",1)
                  @note("$")
                  @skip_lines("^D.O.B")
                  @note("^Degree")

#Skip over "Degree Program:" and make degree into
#category under category Program

"^Degree"         @skip("\: +",1)
                  @category("$","Program",)

#Add all of Work experience and Comments lines to end
#of note

"^Work"           @note("Name")
                  @end()
```

Figure 7-2 *Definition file for converting student information text file*

When the resulting structured file is imported into Agenda, the name Lynne George is added as an item and is assigned to the category Fashion Design. The item has a note that contains various information about the candidate Lynne George. The category Fashion Design is added as a child of Program, which is itself a child of the current category specified when you use the **File Transfer Import** command.

Handling More Than One Text Layout

You may need to write a definition file that can handle more than one text layout. For example, it might need to convert electronic mail messages whose topics are labeled either by Subject: or Re:, or that end in several different ways depending on the electronic mail source.

When you add statements to the definition file, make sure you add statements whose patterns match each differing layout. For example, add a statement that matches and converts a topic line that starts with Subject:, and another statement for a topic line that starts with Re:.

If the text can end in more than one way, include a statement of the form:

END @end()

The above statement executes @end, and copies the current item to the structured file when TXT2STF reaches the end of the text file. Without an @end, TXT2STF stops processing the text file without copying the item it created to the structured file.

Creating More Than One Item

You may want to create more than one item from the information in a text file. For example, a text file can contain several electronic mail messages, in which case you might want to create a new item for each message.

To create an item, you add statements to the definition file, each of which converts appropriate lines in the text file. Commands in each statement create the item, an item note, and one or more categories from the text file, as appropriate.

After the definition file creates each item, it should execute an @end to copy the item to the structured file. Then, TXT2STF can create another item by converting the next text lines in the text file.

If the text can end in more than one way, it might be useful to end each item by searching for the start of the *next* occurrence of text in the file (for example, the next electronic mail message). Then, make @end the first command in the statement followed by the commands that convert the matched text line. To write the very last item to the structured file, include a statement of the form:

END @end()

The following strategies describe typical situations when you may want to create more than one item from a text file:

- The text file contains multiple occurrences of the same type of information (for example, several electronic mail messages).

  In this case, each message has basically the same layout and can be converted by the same group of statements. At the end of each item (for example, the end of each message), execute @end. End the definition file with the statement END @end, as described above.

- The text file contains different types of information (for example, both electronic mail and telex messages).

In this case, you must add definition file statements that can handle each type of information in the file.

If the types of information differ significantly in layout, you may need to organize the statements for each layout into separate groups, each with its own pair of @start and @end commands. Even if differences in the file formats do not require separate statement groupings, you might prefer to group them separately to improves the readability of the definition file.

Appendix A
What's New in TXT2STF for Agenda 2.0

New features have been added to TXT2STF to support Agenda 2.0 features and also to enhance TXT2STF performance and give you greater control over how TXT2STF converts text files into structured files.

In this Appendix

This appendix describes changes in

- Running TXT2STF
- Definition files, including changes in patterns and commands (previously called actions)
- Structured file tags

Running TXT2STF

You run TXT2STF to convert the contents of a text file into items, categories, and notes. TXT2STF puts the items, categories, and notes in a structured file, which you can then import into Agenda.

The following differences apply whenever you run TXT2STF:

- TXT2STF runs faster than in Agenda 1.0. Comments in definition files do not slow performance.
- New error and warning messages describe situations that arise when TXT2STF converts a text file to a structured file. (See Appendix D.)

- New debugging options for use with definition files replace the TXT2STF debugging options used in Agenda 1.0 (see below).

TXT2STF still supports the /C, /D, and /O command-line options that you can use whenever you run TXT2STF (even when you do not use a definition file). For more information about running TXT2STF and using these options, see Chapter 6.

You also specify TXT2STF debugging options in the TXT2STF command line. The new TXT2STF debugging options are

Option	*With this option, TXT2STF*
/A	Uses all debugging options (/L, /M, /T, /U, and /V)
/L	Displays the current text line when TXT2STF loads or modifies it
/M	Displays the text strings that match patterns that start definition file statements
/T	Displays each definition file command when it's executed
/U	Displays each category that TXT2STF creates
/V	Displays the contents of each variable every time it's modified

For more information about TXT2STF debugging options, see Chapter 7.

Definition Files

You can create definition files to give TXT2STF instructions about how to convert text files into structured files.

When you continue a definition file statement to another line, you no longer need to start the continued line with a backslash (\). (You still must, however, make sure that you split the statement *between* commands and that you don't split a command to two lines.) TXT2STF still accepts backslashes (\) at the start of a line, so you do not need to remove the backslashes (\) from existing definition files.

Patterns

Patterns describe a text string that you want the definition file to match. For details about patterns see Chapter 4.

Changes have been made to the match-control characters you can specify in a pattern and to the number of character classes you can include in a pattern.

Match-control characters

The following changes have been made to match-control characters:

Match-control character	*Changes*
Caret (^)	Must be the first character in the entire pattern; for example "^Hardware ¦ ^Software" must now be written as "^(Hardware ¦ Software)" and "Re\: ¦ ^Subject\:" must be "^Subject\: ¦ Re\:"
Dollar sign ($)	Must be the last character in the entire pattern; for example "follows$ ¦ below$" must now be written as "(follows ¦ below)$" and "emblem$ ¦ sign" must be "sign ¦ emblem$"
Exclamation point (!)	No longer supported as the NOT character (see Tilde, below)
Parentheses ( )	Can be a maximum of 10 nesting levels defined by nested parentheses in a pattern
Square brackets ([ ])	Is always case-sensitive, regardless of whether you specify the /C command line option in the TXT2STF command
Tilde (~)	(New) Matches any line that contains a string that is *not* the pattern; creates a negative pattern, but works differently than the exclamation point (!) did in Agenda 1.0

Character classes

A character class is defined by colon a (:a), colon d (:d), colon n (:n), colon space (:), and by each range/group specification made in brackets ([ ]). TXT2STF now restricts you to a maximum of eight character classes per command.

Definition File Commands

Definition file commands, called actions in Agenda 1.0, tell TXT2STF how to convert the current line into item, category, or note text. For details about definition file commands, see Chapter 5.

With Agenda 2.0, you have more flexibility in using variables in commands. Several new commands have been added and existing commands have been changed to let you take advantage of new Agenda and TXT2STF features.

Variables

You can now take much more advantage of variables in TXT2STF commands. For example, you can now use @trim to put text from a text line in a variable, which lets you use the trimmed text in later commands or statements. New commands @append and @grab put text in variables, which makes it easier to use variables as arguments in definition file commands and gives you more flexibility in writing definition files.

New commands

TXT2STF provides the following new commands:

New command	*Description*
@append	Adds a string to the end of the current contents of the variable
@append_item	Appends a string to the end of the item currently being constructed from the text file
@append_note	Appends a string to the end of the current note
@grab	Divides the current text line into several values, and puts each value into a variable
@item2	Creates an item using text from the text file; is an alternative to @item that provides greater control over how TXT2STF constructs an item
@numeric	Creates an assignment to a numeric category
@unindexed	Creates an assignment to an unindexed category

Changed commands

Changes and enhancements have been made to the following commands:

Existing command	*Changes made to the command*
@category and @custom_category	Can specify the ancestors of a parent category (enhancement to the second argument)
	Can now create "date", "exclusive", "numeric", "standard", and "unindexed" categories (enhancement to the third command argument)
@date	Can now name any date category and optionally identify a date value (such as 11/01/90) for the date category (enhancement to the first command argument; new optional second argument)
@make_external_note	Can now create note files that contain more than 10,000 characters
@table	Gives you greater flexibility in converting tabular information and lets you create any type of category (new argument list)
@trim	Can put trimmed text in a variable (new optional fourth argument)

Structured File Tags

A structured file contains text for items, categories, and notes. This text is embedded in structured file tags, which tell Agenda how to import the text.

Changes have been made to the following structured file tags:

Tag	*Change*
{C}	{C} can now create date, exclusive, numeric, standard, and unindexed categories. For date, numeric, and unindexed categories, the final term in the category specification is the value associated with the category, such as the date. Special characters specify the type of category to create. The escape character (%) lets you include any of these special category characters as normal text characters if necessary.
{STF}	{STF} specifies header information for the structured file using a new format. This new header tells Agenda that the structured file contains structured file tags for Agenda 2.0. Agenda still accepts structured files created with the Agenda 1.0 header; however, if the structured file contains Agenda 2.0 structured file tags, the information may be imported differently than how you want. For example, date, numeric, and unindexed categories would be imported as unindexed categories.

Appendix B
Structured Files

TXT2STF converts text from a text file, placing the converted text in a specially formatted file called a structured file. A structured file contains text that is structured in a way that Agenda can import.

After you create a structured file, you use the Agenda **File Transfer Import** command to import the structured file. For more information, see "Importing the Structured File" in Chapter 6.

You can also use your own software products or tools to create a structured file. In this case, you must create a structured file that conforms to the layout described in this chapter.

This appendix provides information to help you examine a structured file when you debug a definition file. It also provides information you need to know if you are creating your own structured file.

In this Appendix

This appendix describes the layout of structured files by providing

- An overview of structured files
- A description of structured file tags
- Sample structured files that you can use to see how text in a structured file typically looks

About Structured Files

A structured file includes information that is ready to be imported into Agenda. TXT2STF creates a structured file whenever you run TXT2STF to convert a text file or when you export from Agenda.

You need to be familiar with structured files and what they contain when you debug a definition file. By examining the definition file, you can quickly see how TXT2STF interprets the definition file statements. For example, you can see whether TXT2STF converted the proper text into items and assigned items to categories in the way you want.

You should also be familiar with structured files if you create your own structured files using a programming language, such as BASIC or PASCAL. The program you write to create the structured file must structure text so that Agenda can import the text in the way you want.

For more information about importing a structured file, see "Importing the Structured File" in Chapter 6. For additional information about importing files, see Chapter 23 of the *User's Guide*.

Note If users are sharing a structured file on a Local Area Network (LAN), they all can see the contents of any structured files, as long as the structured files are unprotected. However, only the first user to open a structured file can modify it. Thus, if a file is being modified, other users can see the file and use its contents but cannot make changes to it. Changes made to the structured file cannot be seen by other users on the LAN until the user making the changes exits from the structured file. For information on file reservation, see Appendix F of the *User's Guide*.

What a Structured File Contains

A structured file contains text to be imported into an Agenda file, with tags that tell Agenda how to import the text. For example, tags specify which text to use as items, categories, and notes, and which date format to use when importing dates.

Structured file tags are special codes that provide instructions to Agenda. Each structured file tag is enclosed in a pair of braces ({ }); for example, {I}. Structured file tags can create

- One or more items, each item having related categories and a note
- One or more independent categories (not assigned to items)
- Independent notes (attached to blank items on import)

A structured file can contain tags to create any or all of these.

You can examine a structured file using any text processor that can read ASCII text files. It is very helpful to look at a structured file when you are debugging the definition file that created it.

Creating Your Own Structured File

You can create your own structured file. This may be useful, for example, if you have a text file that cannot easily be formatted using TXT2STF (for example, you need to perform branching or conditional operations), or if you are already familiar with a programming language such as BASIC or PASCAL, and would prefer to use that language.

If you create your own structured file, keep in mind that the file must be a text file, must include structured file tags in the proper format, and must have an extension of .STF.

Structured File Tags

Structured file tags tell Agenda how to import the text in the structured file. For example, the {C} tag tells Agenda that the immediately following text should be imported as a category. The {N} tag indicates where note text begins.

Agenda provides structured file tags that

- Begin a structured file
- Create comment text (a block of text that contains information that Agenda ignores on import)
- Specify the format of dates in the structured file, so Agenda can correctly import subsequent dates
- Create notes
- Create categories (category and family, with any associated notes)
- Create items (item and associated categories, notes, and dates)

TXT2STF includes structured file tags in text when it copies the contents of the current item, category, and note buffers created by converting a text file to the structured file. If you construct your own structured file, you must make sure to include the correct tags.

The following table lists the tags that can be included in structured files. Tags must be in the same case (lowercase or uppercase) as shown in this table.

Tag	*Meaning*
{d}	Specifies a date format, such as MM/DD/YY
{C}	Beginning of a category specification (the category and family, with any associated notes)
{D}	Done date
{F}	Beginning of a category note
{E}	Entry date
{G}	Name of the note file for the category
{I}	Beginning of an item specification (the item and associated categories, and notes)
{N}	Beginning of an item note
{O}	Name of the note file for an item
{S}	Beginning of comment text to be ignored when imported
{STF}	Header that begins a structured file
{T}	Beginning of the text of an item
{W}	When date
{.}	End of a category specification
{!}	End of an item specification

Figure B-1 shows a sample structured file containing structured file tags. This example contains information converted from a sample MCI Mail® file.

```
#Processing electronic mail messages
{STF}11/19/90;11:54:49;002
{I}
{T} Weekly Sales Update
{C}\cc\    Alan Stewart %/ MCI ID: 234-5678{.}
{C}\To\    Pam Crawford %/ MCI ID: 567-8901{.}
{C}\To\   * Paul Kyn %/ MCI ID: 890-1234{.}
{C}\From\  Lynne George %/ MCI ID: 123-4567{.}
{C}\Entry@ ¦   Mon November 19, 1990  9:24am  GMT{.}
{C}\Mail\E-Mail{.}
{N}            WEEKLY SALES UPDATE: Week of 11/16/90
Northeast/Mid-Atlantic:
October goal of 70,000 met due to success of CopyCat and Stratford copiers;
hiring of two new personnel for fall school promotions (Henry Clarkman --
Hartford, Ct., Jennine Powell, New York City schools); regional sales meeting
Monday 12/3/90.
{!}
```

Figure B-1 *Sample structured file*

The above sample creates a single item, whose text is Weekly Sales Update. It adds the item to the cc, To, From, and Mail categories, and assigns it an Entry date of Mon November 19, 1990. All text after the {N} is a note for the item. The entire item specification starts with {I} and ends with {!}.

The remainder of this chapter describes the types of tags that can be included in a structured file, and gives additional information about how Agenda interprets tags and the structured information they identify.

Structured File Header Tag

{STF} starts a structured file. This tag must be the tag in the structured file. The following information occurs on the same line, immediately after this tag:

- The date and time when the structured file was created; the date is in MM/DD/YY format and the time uses a 24-hour clock
- A revision number that tells Agenda which revision of structured file tags are in the file; for Agenda release 2.0, this revision number value *must* be 002 (with leading zeros)

A semicolon (;) separates the date from the time and the time from the revision number.

Note If the header is incorrect, structures new to Agenda Release 2.0 will not be imported as expected. For example, all date and numeric categories are imported as unindexed categories.

Figure B-2 shows a sample structured file header line.

```
{STF}11/02/90;15:20:11;002
```

Figure B-2 *Sample header line*

If you create your own structured file, make sure you use the format shown in Figure B-2. Notice that the date must be in the default Agenda format of MM/DD/YY, and that the header must end with the number 002.

Comment Tag

{S} begins comment text in the body of the structured file. Comments let you describe what a structured file or its tags do. For example, you can include comments to describe how a group of tags organizes information to be imported.

Agenda ignores comment lines when it imports information from the structured file. Agenda ignores any text between {S} and either the next tag in the structured file or the end of the structured file.

Date Format Tags

Date format tags specify the format of dates and times in the structured file. On import, Agenda uses the date format specification to interpret dates in the structured file.

By default, Agenda assumes that dates in the structured file are in the format MM/DD/YY and use a 24-hour clock. If dates in the text file have a different format, the structured file must specify the format *before* specifying any date values. You can omit date format tags if all dates are specified either in the default format (MM/DD/YY) or in a combination of words and numbers, as in "Thursday November 8, 1990".

Note The date format tag does not affect the date and time specified in the {STF} tag, which *must* specify a date in MM/DD/YY format and time using a 24-hour clock.

TXT2STF adds a date format tag to the structured file when it executes @date_format command in a definition file.

The following table lists the valid date format tags and the date formats they specify.

Note These are only a subset of the date formats supported by Agenda. (See Chapter 7 in the *User's Guide*.)

Format number	*Associated date format*	*Associated time format*
1	MM/DD/YY	24-hour clock
2	DD/MM/YY	24-hour clock
3	DD.MM.YY	24-hour clock
4	YY-MM-DD	24-hour clock
5	DD-MMM	24-hour clock
6	DD-MMM-YY	24-hour clock
7	MM/DD/YY	12-hour clock
8	DD/MM/YY	12-hour clock
9	DD.MM.YY	12-hour clock
10	YY-MM-DD	12-hour clock
11	DD-MMM	12-hour clock
12	DD-MMM-YY	12-hour clock

Note Each date format tag must contain a *lowercase* d, and must be immediately followed by the format number as shown in the above table.

Figure B-3 shows an example of using date format tags.

```
{d}2
{C}\Activity Date@¦  15/11/90 {.}
{C}\Start Date@¦   Thu November 15, 1990  3:44pm  GMT.{.}

{d}1
{C}\Start Date@¦      11/19/90{.}
```

Figure B-3 *Sample date format tags*

The example in Figure B-3 begins with a {d}2 tag, which specifies that the structured file contains dates in the format DD/MM/YY. The next two {C} tags create date categories; each category specification starts with {C} and ends with {.}. The {d}2 tag applies to the first {C} in the pair, which adds a date in the DD/MM/YY format. The {d}2 tag does not apply to the second {C} specification, since the date value in that specification contains both words and numbers.

The {d}1 tag then changes the date format to MM/DD/YY format. This tells Agenda that subsequent date values in the structured file are in MM/DD/YY format. This {d}1 enables Agenda to import the date in the immediately following {C} specification, which is in MM/DD/YY format.

For more information about the {C} tag for specifying categories and dates see "Category Tags" later in this Appendix.

Note Tags

Note tags identify note text to be imported, or the name of a note file to be used by an item or category. The following table lists the note tags that can be included in a structured file.

Tag	*Meaning*
{F}	Beginning of a category note
{G}	Name of the note file for the category
{N}	Beginning of an item note
{O}	Name of the note file for an item

Caution A note can be a maximum of 10,000 characters. When Agenda imports a note that is longer than this maximum, it truncates the remainder of the imported note text when the note reaches 10,000 characters in length. Item text can contribute to the size of a note, as described in "Item Tags" later in this chapter.

Category notes

{F} and {G} identify category notes and must occur between tags that create a category. (See "Category Tags" later in this appendix.) Agenda adds the note to the specified category. If more than one category note tag occurs in the category specification, Agenda uses only the last one and discards the preceding note tags.

{F} starts a regular note. The note text to be imported follows immediately after, on the same line. Agenda assumes that all remaining text is note text until it encounters another structured file tag or reaches the end of the structured file.

{G} identifies an external file to be attached as a note file. If {G} identifies a note file that does not exist, Agenda asks you whether you want to create it when you attempt to look at the note in the Agenda file.

Figure B-4 shows sample category note tags in a category specification.

```
{C}\Client\
{G}c:\Agenda\clients.lst
{.}
```

Figure B-4 *Sample category note tags*

In Figure B-4, {C} and {.} begin and end the category specification, as described in "Category Tags" later in this appendix. Within the category specification, {G} instructs Agenda to make external file AGENDA\CLIENTS.LST the note file for the Client category.

Item notes

{N} and {O} identify notes. The location of the note tag determines what Agenda does with the imported note.

- If the tag occurs between item tags, as described in "Item Tags," Agenda adds the note to the item.
- If the tag occurs outside of an item specification, Agenda creates an empty item and adds the note to that item. This lets you import a note without assigning it to an item or category.

If more than one note tag appears in an item specification, Agenda uses only the last one and discards the preceding note tags.

{N} starts a regular note. The note text to be imported follows immediately after, on the same line. Agenda assumes that all remaining text is note text until it encounters another structured file tag or reaches the end of the structured file.

{O} identifies an external file to be attached as a note file. If {O} identifies a note file that does not exist, Agenda asks you whether you want to create it when you try to look at the note in the Agenda file.

Figure B-5 shows sample item note tags in an item specification.

```
{I}
{T} Sales Reward Dinner
{N}Allgrands Hearthside Inn
1234 Highway 1
West Linton, WA
  (Just past the Lighthouse Mall on the left)
{!}
```

Figure B-5 *Sample item note tags*

Figure B-5 defines an item, Sales Reward Dinner, and defines a note describing the location of the restaurant where the dinner will be held. On import, Agenda adds the note to the item because {N} is included between the {I} and {!} item tags that create the item. For more information see "Item Tags" later in this chapter.

Category Tags

Category tags create a category and its family, along with any associated category notes. The location of the category tags determines what Agenda does with the imported category.

- If the category tags occur between tags that create an item, as described in "Item Tags," Agenda assigns the item to the category.
- If the tags occur outside an item specification, Agenda adds the category as an independent category to the category hierarchy, but does not assign the categories to an item.

For more information see "Item Tags" later in this chapter.

The following tags create a category:

Tag	*Meaning*
{C}	Beginning of a category specification (the category and family, with any associated notes)
{.}	End of category specification

Each {C} begins a new category specification. The name of the category to be imported follows immediately after, on the same line. Each category ends with a {.} tag. If category note tags {F} or {G} occurs between {C} and {.}, Agenda adds a note to the category on import.

When you use the **File Transfer Import** command, you can specify whether Agenda should import either or both of the following types of categories:

- New categories created in item specifications
- New independent categories

Agenda imports the categories you specify and adds them to the category hierarchy. For more information, see Chapter 23 in the *User's Guide.*

If the category is created in an item specification, Agenda assigns the item to the category on import. If the category is a date, numeric, or unindexed category, Agenda uses the associated value, such as the date, to assign the item to the category.

Caution If a date, numeric, or unindexed category is specified outside of an item specification, Agenda creates the category if appropriate, but does not import the associated numeric, date, or unindexed value.

If the structured file identifies a category that already exists, Agenda retains the original category on import, and does not import the duplicate category.

Category characters

Special category characters in the category name tell Agenda the type of category to create.

Category Character	*Meaning*
\	Standard category
/	Exclusive
¦	Unindexed
#¦	Numeric
@¦	Date

The category character immediately *follows* the category to which it applies. For example, the following text after a {C} tag adds Date Delivered as a date category:

Date Delivered@¦

To indicate parent and child relationships in category information, the {C} tag separates each parent name from its child by using the appropriate category character. In date, numeric, and unindexed categories, the category character separates the category name from the associated value. For example, the following text after a {C} tag adds the Date Delivered category with a value of 11/26/90:

Date Delivered@¦ 11/26/90

If the structured file identifies a new parent category, Agenda creates the parent category when it imports the structured file.

If the category specification *begins* with the standard category character, the backslash (\), Agenda imports the category as a child of MAIN. For example:

{C}\From\ Terry Smith {.}

In the above example, the backslash (\) in front of From tells Agenda to add From as a child of MAIN.

A category specification that starts without a category character is added as a child of the current category specified by the **File Transfer Import** command. (See Chapter 23 in the *User's Guide.*)

Date categories

By default, Agenda assumes all dates are in the format MM/DD/YY. Dates in a different format must be preceded with a date format tag that identifies the format. (See "Date Format Tags" earlier in this chapter.) On import, Agenda discards any information that cannot be interpreted as date information in the current format.

Date categories can be created using {C} and {.} tags. However, Agenda also supports an older set of category commands for creating Agenda-defined date categories. These tags are:

Tag	*Meaning*
{D}	Done date
{E}	Entry date
{W}	When date

When any of these tags is used to create a date category, the date value immediately follows and concludes the date specification. A date specified by any of these tags does *not* end with the category-end tag {.}.

For example, the following line adds a When date of 11/12/90:

{W}11/12/90

The following example shows how the same date category specification looks using {C} and {.} tags:

{C}\When@¦ 11/12/90

Escape character

An escape character is a special character that tells Agenda to interpret the immediately following character as a text character. This is necessary whenever a category includes one of the category characters as a text character. The escape character in structured files is the percent sign (%).

For example, if a category includes the number sign (#) in its text, the number sign must be preceded with the escape character (%#) so that Agenda imports the number sign as text. Without the immediately preceding escape character (%), Agenda interprets the number sign as the first character in the numeric category character.

The following guidelines apply to the use of escape characters in category specifications:

- The escape character must immediately precede the special character to which it applies. There cannot be a space between the escape character and the character.
- Each category character to be imported as text requires its own escape character. For example, if two slashes in a row (//) are to be imported as text, an escape character must precede *each* slash character, for example:

  %/%/

- If the category includes the percent sign (%) as a text character, the category specification must include two percent signs in a row:

  %%

Figure B-6 shows how category tags define categories.

```
{C}\From\   Lynne George %/ MCI ID: 123-4567{.}
{C}\When@ ¦   Mon November 19, 1990  9:24am  GMT{.}
```

Figure B-6 *Sample category tags*

Figure B-6 defines two categories. Because each {C} specification begins with a backslash (\), the new categories are added as children of MAIN when they are imported into Agenda. The first {C} defines a From category, and makes the category Lynne George (along with her MCI ID) a child of From.

The second {C} defines a When date of Mon November 19, 1990. The date category characters (@ ¦) apply to the When category, and indicate that When is the parent category. The next value in the category specification is the date. If this When category is imported with an item, Agenda uses the date to assign the item to the When category. If the When category is not imported with an item, Agenda ignores the date value.

Item Tags

Item tags create an item and its associated categories and note. The following tags create an item:

Tag	*Meaning*
{I}	Beginning of an item specification (the item and associated categories, notes, and dates)
{T}	Beginning of the item text
{!}	End of an item specification

Each {I} begins a new item. The item ends with a
{!} tag. Tags between {I} and {!} add other elements to the item:

- The {T} tag begins the item text
- Category tags assign the item to the specified categories
- Note tags {N} or {O} add a note to the item

Category and note tags are discussed earlier in this chapter.

{T} specifies the item text. On import, Agenda ends the item text when it encounters another structured file tag. In Agenda, the item text can be a maximum of 350 characters in length.

Caution If {T} identifies item text that is longer than this maximum, Agenda makes the first 350 characters into the item and ignores the remaining characters.

When you use the **File Transfer Import** command, you can specify whether Agenda should import any of the following information from item specifications in the structured file:

- Items and notes
- Category assignments (assignment of items to existing categories)
- New categories created in item specifications

You also can specify the category to which imported items are assigned. For more information, see Chapter 23 in the *User's Guide.*

Figure B-7 shows how sample item tags define an item.

```
{STF}11/02/90;12:21:08;002
{I}
{T}Friday Lunch Canceled
{C}\From\LeeX {.}
{C}\Entry@ ¦  Fri Nov 2, 1990  9:23am  GMT{.}
{C}\Mail\E-Mail{.}
{N}Rescheduled meeting Mon Nov 5, at 9am.
{!}
{I}
{T}Attention Book Fans
{C}\From\MartyV {.}
{C}\Entry@ ¦  Fri Nov 2, 1990  11:17am  GMT{.}
{C}\Mail\E-Mail{.}
{N}Gala Re-Opening
Au Claire Books announced it's reopening its downtown store on the week-
end of Nov 10 (you may remember, they were burned out along with Stacy's
a few months ago).

Their flyer says they'll have poetry readings Sat and Sun from 1 to 3 in the
afternoon, and balloons for the kids. New books upstairs, and an expanded
used section downstairs. And, an expanded comfy chair section!! (And, as
always, 5% of sales to the usual charitable causes, including the Homeless
Coalition, the Literacy Campaign, etc.).
{!}
```

Figure B-7 *Sample item tags*

Figure B-7 contains information extracted from electronic mail. This sample constructs two items. The text for the first item is Friday Lunch Canceled. The second item is Attention Book Fans. Each item is assigned to the From and Mail categories, and is assigned an Entry date. All text after the {N} is a note for the item. Each item specification starts with {I} and ends with {!}.

What Structured Files Look Like

This section provides examples of structured files. The first structured file is based on a simple example in Chapter 4. The remaining structured files in this section are generated by the definition files presented in Chapter 3.

This section presents

- A simple electronic mail example with text file, definition file, and resulting structured file
- The sample structured files created for the examples in Chapter 3

A Simple Electronic Mail Example

This example shows one way to convert text in a sample electronic mail memo into category and item text. Figure B-8 shows a text file that contains an electronic mail message to be converted.

```
To:  Jill
From: Linda

Subject:  Your proposal

                Jill, I liked your proposal
                but I had a few questions,
                could we get together next Wednesday
                to talk about it?
```

Figure B-8 *Text file containing an electronic mail message to be converted*

Figure B-9 shows the definition file that converts the sample text file. This definition file can convert a text file containing one or *more* electronic mail messages. The definition file assumes that each new electronic mail message begins with the string To: at the start of a line. When it finds this string, the definition file executes @end to save the current contents of item, note, and category buffers to the structured file. Then it then begins converting the electronic mail message by executing @trim. It continues converting text lines until it finds another To: at the start of a line, at which point it executes @end, and then converts the new electronic mail message. When it reaches the end of the text file (END), the definition file executes @end and then terminates.

```
#Converting memo text using @append_item

START               @equate("SPACE"," ")
                    @equate("PERSON","")
                    @start()

#Create To category ("To:" starts each memo in the file)

"^To\:"             @end()
                    @trim("^To\:", "$", N, "PERSON")
                    @custom_category(PERSON, "To",)

#Assign a person to the From category

"^From\:"           @trim("^From\:", "$", N, "PERSON")
                    @custom_category(PERSON, "From",)

#Add subject line to item buffer

"^Subject\:"        @trim("^","Subject\:",,)
                    @item2("$")
                    @append_item(SPACE)

#Discard blank lines so "~To:"(NOT To:) won't match them

"^[ ]*$"            @equate("blanks","")

#Add each remaining line to item buffer

"^~To\:"            @trim("^[ ]*","$", N, "LINE")
                    @append_item(LINE)
                    @append_item(SPACE)

END                 @end()
```

Figure B-9 *Definition file to convert text files containing electronic mail messages*

The definition file shown in Figure B-9 converts the names Jill and Linda into categories and converts the remaining memo text into an item. When it converts the memo text, the structured file removes the leading spaces from the text. Figure B-10 shows the structured file created by this definition file.

```
#Converting memo text using @append_item
{STF}11/26/90;19:43:34;002
{I}
{T}Your proposal
Jill, I liked your proposal but I had a few questions, could we get together
next Wednesday to talk about it?
{C}To\ Jill{.}
{C}From\ Linda{.}
{!}
```

Figure B-10 *Resulting structured file*

When the structured file is imported into Agenda, To and From are added as categories under the category specified in the **File Transfer Import** command. The name Jill is added as a child of To and Linda is added as a child of From. The item, which consists of the text from the Subject line plus the four-line memo, is assigned to the categories Jill and Linda.

Sample Structured Files

This section shows the sample structured files developed for the examples in Chapter 3. That chapter presented the text file and definition file to:

- Convert electronic mail messages
- Convert a table of information

The resulting structured files for each example is shown below.

Converting an electronic mail message

Figure B-11 shows the structured file created when TXT2STF used the sample definition file to convert the sample electronic mail file presented in Chapter 3.

```
#definition file for electronic mail
{STF}05/24/90;13:59:54;002
{I}
{T}Charting Seminar!

{C}From\Susan Anthony\
{F}Public Relations
{.}
{C}\Entry@ ¦    11/08/90 15:12:04{.}
{O}EMAIL5.EML
{!}
```

Figure B-11 *Sample structured file for converted electronic mail*

Converting a table of information

Figure B-12 shows the structured file created when TXT2STF used the sample definition file to convert the sample CD/Corporate Industry Participants report file presented in Chapter 3. Since the original report, and therefore the resulting structured file, is long, Figure B-12 shows only the structured information for the first five companies in the report.

```
#Table-reading DEF file sample for CD/Corporate participant table
{STF}05/24/90;09:59:34;002
{C}Industry\Food Processing\{.}
{I}
{T}Sara Lee Corp
{C}Income # ¦ 410,492{.}
{C}Sales# ¦ 11,717,678{.}
{C}FYE@ ¦ 7%/01%/89{.}
{C}Company\{.}
{C}Rank ¦ 1{.}
{!}
{I}
{T}Conagra Inc
{C}Income # ¦ 197,878{.}
{C}Sales# ¦ 11,340,414{.}
{C}FYE@ ¦ 5%/28%/89{.}
{C}Company\{.}
{C}Rank ¦ 2{.}
{!}
{I}
{T}IBP Inc
{C}Income # ¦ 62,328{.}
{C}Sales# ¦ 9,066,101{.}
{C}FYE@ ¦ 12%/31%/88{.}
{C}Company\{.}
{C}Rank ¦ 3{.}
{!}
{I}
{T}Archer Daniels Midland Co
{C}Income # ¦ 424,673{.}
{C}Sales# ¦ 7,928,836{.}
{C}FYE@ ¦ 6%/30%/89{.}
{C}Company\{.}
{C}Rank ¦ 4{.}
{!}
{I}
{T}Borden Inc
{C}Income # ¦ 311,882{.}
{C}Sales# ¦ 7,243,526{.}
{C}FYE@ ¦ 12%/31%/88{.}
{C}Company\{.}
{C}Rank ¦ 5{.}
{!}
```

Figure B-12 *Sample structured file for converted CD/Corporate Industry Participants report*

Appendix C
Quick Reference

This Quick Reference provides information about the patterns and commands you can include in definition files.

Patterns

To specify a pattern, you can

- Type the text string itself
- Include match-control characters in the pattern
- Use special patterns START and END

Match-Control Characters

Use the following match-control characters to write patterns that can match a varied set of text strings in the input line.

Match-Control Character	*Description*
Asterisk (*)	Matches zero or more occurrences of the preceding pattern character
Backslash (\)	Specifies that the immediately following character in a pattern is a text character and not a match-control character
Brackets ([ ])	Specifies a range or group of characters, and matches any single text character specified in the range or group

continued

Match-Control Character	*Description*
Caret (^)	As the first character in a pattern, causes the pattern to match the corresponding text only when it occurs at the beginning of a text line As the first character in a pair of brackets ([ ]), caret (^) negates the range or group specified in the brackets ([ ])
Colon a (:a)	Matches any single uppercase or lowercase alphabetic text character
Colon d (:d)	Matches any single numeric text character from 0 through 9
Colon n (:n)	Matches any single alphabetic or numeric text character
Colon space (:)	Matches a single space or control character in the text file
Dollar sign ($)	As the last character of a pattern, causes the pattern to match the corresponding text string only when it occurs at the end of a text line
Parentheses ( )	Groups pattern characters into a string so you can apply a match-control character to the string
Period (.)	Matches any single character in the text file (except a carriage return or linefeed character)
Plus (+)	Matches one or more occurrences of the immediately preceding pattern character
Tilde (~)	Matches any text string that is *not* the pattern
Vertical bar (¦)	Matches one pattern *or* another (performs a logical OR)

Special Patterns

You can use the following two special patterns in definition file statements.

Pattern	*Used for*
START	Statements you want to execute only when TXT2STF starts processing the text file
END	Statements you want to execute only when TXT2STF finishes processing the text file

Definition File Commands

The following pages describe

- Syntax for definition file commands
- Predefined variables you can use as command arguments

Command Syntax

This section provides the syntax for definition file commands.

Note You can include spaces after the commas that separate arguments. If you omit an optional argument, you must *include* the comma that precedes the argument only if the comma *precedes* the brackets ([ ]) in command syntax.

Command	*Description*
@append("*variable-name*",*string*)	Adds a string to the end of the current contents of the variable
@append_item(*string*[,*N*])	Adds a string to the end of the current item
@append_note(*string*)	Adds a string to the end of the current note
@category(*separator-pattern*,[*parent*],["*type*"])	Creates one or more categories from the text line and can make them children of a specified parent category
@category_note(*termination-pattern*)	Creates a category note and assigns it to the category created by the most recently executed @category or @custom_category
@category_note_file(*file-name*)	Identifies the external note file for the current category
@custom_category(*category*,[*parent*],["*type*"])	Creates a category from text that you specify and can make the category a child of a specified parent category
@date(*category*[,*value*])	Creates assignments to date categories; assumes date format MM/DD/YY

continued

Command	*Description*
@date format(*format-number*)	Indicates the format of dates and times in the structured file by specifying a format number (formats 1-6 use 24-hour time; 7-12 use 12-hour time): 1 and 7 = MM/DD/YY 2 and 8 = DD/MM/YY 3 and 9 = DD.MM.YY 4 and 10 = YY-MM-DD 5 and 11 = DD-MMM 6 and 12 = DD-MMM-YY
@end()	Copies the current contents of item, note, and category buffers to the structured file, and then initializes the buffers
@equate("*variable-name*","*pattern*")	Stores a pattern or a text string in a variable
@grab(*separator-pattern,field-number,"variable-name"* [*,field-number,"variable-name"*,...])	Divides the current text line into several values and puts one or more of the resulting value into variables; leaves the text line intact in memory
@item(*termination-pattern,total-length,*[*N*])	Creates an item using text in the text file; is the Release 1.0 alternative to @item2
@item2(*termination-pattern*[*,max-length*][*,N*]) or @item2(*max-length*[*,N*])	Creates an item using text in the text file; is the Release 2.0 alternative to @item
@make_external_note(*termination-pattern,*[*"directory"*])	Creates an external note file and copies text from the text file into the note file; on import the note file is added as an item note
@note(*termination-pattern*)	Creates a item note and assigns it to the current item
@note_file(*file-name*)	Identifies the external note file for the current item
@numeric(*category*[*,value*])	Creates assignments to numeric categories
@replace(*search-pattern,replacement-string,*[*number*])	Replaces the specified number of occurrences of the string identified by *search-pattern* with the replacement string
@reset(*replacement-string*)	Replaces the current text line in memory with the specified string
@set("*variable-name*")	Stores the current text line in the specified variable

continued

Command	*Description*
@skip(*separator-pattern,field-number*)	Divides the text line into values and then deletes one of the resulting values from the text line
@skip_lines(*termination-point*)	Skips over the number of text lines you specify
@start()	Initializes the item, note, and category buffers TXT2STF uses to hold converted information
@strip(*strip-pattern,number*)	Deletes one or more occurrences of a specified string from the text line
@table(*separator-pattern,termination-pattern,col-number,"type"* [,*col-number,"type"*,...])	Makes items and categories from a table in the text file, using *separator-pattern* to divide the text into table columns
@trim(*start-pattern,end-pattern*,[N][,*"variable-name"*])	Deletes all the text between one string and another, and can place the deleted text in a variable
@unindexed(*category,value*)	Creates assignments to unindexed categories

Predefined Variables

You can use these predefined variables as arguments in definition file commands.

Variable	*Represents*
FILENAME	The name of the current text file
EOF	End-of-file; used for specifying a termination point in commands
CR	Carriage return; used for adding the combination of carriage return (ASCII decimal 13) and linefeed (ACII decimal 10) to text
D, E, and W	Done date, Entry date, and When date, respectively

Appendix D
Error Messages

Error messages provide information about problems that TXT2STF encounters while converting a text file. This appendix describes the error messages that you may see when working with TXT2STF.

In this Appendix

This appendix provides

- Information about TXT2STF error messages
- A description of each error message, with possible ways to correct the current problem

About Error Messages

Error messages notify you when one or more problems occur in converting a text file and give you a starting place for locating the problem. Each TXT2STF error message starts with an identifying number, followed by a phrase that describes the situation that caused TXT2STF to display an error message.

TXT2STF error messages can identify:

- Errors in how a definition file statement is composed

  For example, the statement may begin with an invalid pattern or might include commands that do not comply with command syntax.

- Errors that arise when TXT2STF uses the definition file

For example, a statement may need to use the contents of a variable before any text is copied into the variable.

- Errors that relate to your computer or its resources

  For example, your computer may run out of memory while TXT2STF is still converting a text file.

If errors occur when you use a definition file, TXT2STF stops executing either the command or the statement where the error occurs and displays the error message with as much information as possible about the current error. TXT2STF then resumes converting the file, after first skipping over the command or statement where the error occurred.

If it skips over a command, TXT2STF lists the command that caused the problem when it displays the error message. If it skips over an entire statement, TXT2STF lists the current command, current text line, and the commands already executed in the statement when it displays the error message.

To see more information about your errors, use the debugging option /A when you run TXT2STF. Also, redirect TXT2STF messages to a file so you can view them later. (See Chapter 7.)

Error Messages and Descriptions

The remainder of this appendix lists TXT2STF error messages and provides possible ways to correct the problem identified by each error message. Error messages are listed by error number.

0 **The end of the definition file was reached before the end of a command**

The definition file ends before expected. The current command is probably missing the closing (righthand) parenthesis.

1 **The current statement has an invalid format**

Make sure the current statement begins with a pattern and includes at least one command to convert text. Make sure that commands begin with the at (@) character and end with an argument list enclosed in parentheses. Also make sure that each pattern is enclosed in a pair of double quotation marks (" ").

2 An unrecognized command is in the definition file

Make sure that all commands in the definition file begin with an at (@) character and are spelled correctly.

3 An argument for the command is not recognizable

Make sure that all arguments in the command are in the correct position and that all *required* commas are included in the command when optional arguments are omitted. Make sure that each argument is the right type. For example, if the syntax specifies that an argument can only be a numeric constant, make sure you have specified a numeric constant. Make sure that each string constant (including file name string and variable name string constants), pattern constant, and type constant is enclosed in a pair of double quotation marks (" "). Also make sure that each numeric constant, logical constant, and variable is *not* enclosed in double quotation marks (" ").

4 There is not enough memory to execute this sequence of commands

Your computer ran out of memory while TXT2STF was converting a text file. It is possible that you are currently running a software product that uses memory. Try to free this memory so that TXT2STF can use it, as follows:

- You may have temporarily exited to the operating system from a software product.

  To free memory, return to the product (typically you type `exit` and press ENTER) and then exit completely from the product.

- You may have run a product that stays resident in memory after executing, such as Lotus Express™.

  To free memory, use the key combination defined for this purpose (for Express, press the key combination ALT-SHIFT-END) or restart your computer without running any of products that remain in memory after use.

If none of the above strategies free enough memory, see if you can shorten the statement so that it will need less memory to execute. It is possible that your computer does not have sufficient memory installed to run the definition file; contact computer support personnel at your site if necessary.

5 There is not enough memory to define another pattern

There are more than 20 patterns currently defined in memory. The pattern that begins a statement remains defined in memory throughout the execution of the statement. Patterns that are used as arguments in commands are defined in memory only while the command is being executed. When the next command in a statement starts executing, the patterns for the previous command are removed from memory and the patterns for the current command are defined in memory.

6 Bad pattern in the command

Make sure that each pattern in the command is enclosed in a pair of double quotation marks (" "). Make sure that you used the correct match-control characters. Make sure the command does not include special patterns START or END; these commands can only begin statements. Make sure that all arguments are in the correct position. Make sure the command includes the correct number of arguments.

7 Not enough arguments were supplied for the command

Make sure that all arguments in the command are specified in the correct position and that all *required* commas are included for optional arguments that are omitted. Make sure the argument list does not include a closing parenthesis in the wrong location.

8 Too many arguments were supplied for the command

Make sure that the command includes only allowed arguments and that the command does not include non-required commas for optional arguments that are omitted.

9 Command or data line too long

The current text line can be a maximum of 512 characters in length.

10 The variable specified in the command line does not exist

The command specifies a variable that does not yet contain text, so it does not exist. Reorganize the statements in the definition file so that the statement that places information into the variable executes before the current statement. To review the order in which TXT2STF executes definition file statements, see "How TXT2STF Processes Each Text Line" in Chapter 2.

11 The variable name specified in the command line is too long

A variable name can be up to 32 characters in length. Make sure that the variable name is spelled correctly. If the current argument must be a variable name string constant, make sure the variable is enclosed in a pair of double quotation marks (" "). If the argument is simply a variable, make sure the variable name is *not* enclosed in quotation marks. Make sure that all arguments in the command are specified in the correct position and that all *required* commas are included for optional arguments that are omitted. Make sure that non-required commas are *not* included for omitted optional arguments.

12 A variable was given for an argument that cannot be a variable

The command includes a variable for an argument that cannot be a variable. Consult the command syntax for valid arguments. Make sure that all arguments in the command are specified in the correct position and that all *required* commas are included for optional arguments that are omitted. Make sure that non-required commas are *not* included for omitted optional arguments.

13 A number was given for an argument which cannot be a number

The command includes a numeric constant for an argument that cannot be a numeric constant. Consult the command syntax for valid arguments. If the number should be a string constant, enclose it in a pair of double quotation marks (" "). Make sure that all arguments in the command are specified in the correct position and that all *required* commas are included for optional arguments that are omitted. Make sure that non-required commas are *not* included for omitted optional arguments.

14 Argument too long

An argument in the command contains too many characters. Consult the command syntax for the correct length for arguments. Make sure that all arguments in the command are specified in the correct position and that all *required* commas are included for optional arguments that are omitted. Make sure that non-required commas are *not* included for omitted optional arguments.

15 Missing , or) in command

The command either is missing a required comma (,) or does not end in a closing (righthand) parenthesis.

16 Variable is full

The current variable contains 512 characters, which is the maximum number of characters a variable can contain.

17 Wrong type supplied for an argument

An argument in the current command is specified incorrectly. For example, the argument is supposed to be a numeric constant, but is currently a string constant.

18 Unbalanced ()'s

A pattern includes a different number of open (lefthand) parentheses than closing (righthand) parentheses. Make sure each open parenthesis has a corresponding closing parenthesis. Also make sure each closing parenthesis closes an expression that starts with an open parenthesis. Inspect nested parentheses carefully for mismatched open and closing parentheses.

19 Too many tildes (~) per subpattern

A subpattern contains too many tilde characters (~). There can be a maximum of one tilde (~) in each subpattern defined

- In a nesting level defined by a pair of parentheses
- At the top level of the pattern, before an open parenthesis begins a nesting level

For example, the pattern "~a~b" is incorrect but ~(ab) and ~a(~b) are correct.

Each pattern on either side of a vertical bar (¦) character defines a new subpattern at the current level. This means that the pattern "~Meeting ¦ ~Seminar" is correct because each subpattern at the current level contains only one tilde (~).

20 Pattern too big

The pattern is too complicated. When TXT2STF tries to compile the pattern into an internal form, the pattern exceeds the capacity of TXT2STF to store it. Try to simplify the pattern.

21 Too many classes

An entire pattern can include up to eight different character class specifications.

22 No closing square bracket ( ])

An open bracket ([ ) in a pattern indicates the start of a range or group specification, but the specification does not end in a closed bracket ( ]).

23 Bad colon(:) class

There are only four match-control characters that begin with colons, colon a (:a), colon d (:d), colon n (:n), and colon space (:). Make sure that each colon space (:) combination includes a space after the colon.

24 Too many ()'s

The pattern includes too many levels of nested parentheses. A pattern can include up to 10 nesting levels. If you end one set of parentheses before starting a second set, both parentheses are at the *same* level of nesting, and do not cause this error to occur.

25 Carat (^) can only be the first character

The carat (^) *must* be the first character in a pattern to match the pattern with a string at the start of a text line. For example, "^Hardware ¦ ^Software" is invalid; the correct way to write this pattern is "^(Hardware ¦ Software)".

26 Dollar sign ($) can only be the last character

The dollar sign ($) *must* be the last character in a pattern to match the pattern with a string at the end of a text line. For example, "follows$ ¦ below$" is invalid; the correct way to write this pattern is "(follows ¦ below)$".

27 Bad pattern

TXT2STF cannot use the pattern because it contains invalid characters or some other error that invalidates it.

28 One of the arguments for the command was empty

It is possible that an argument in the current command either specifies an empty string, which consists of two double quotation marks ("") *without* any intervening space. In this case, correct the argument. Another possibility is that a variable does not contain text when TXT2STF attempts to use it. In this case, see the description for error message 10 for ways to correct the problem.

29 There is not enough memory to define another pattern

Your computer ran out of memory while TXT2STF was converting a text file. See the possible solutions listed for error message 4. Also, see if you can eliminate any patterns from commands in the current statement. Look for commands that can be rewritten to use an argument other than a pattern. For example, see if @item2 can specify where the item ends by using a total length parameter and not a pattern.

30 Unexpected end of file

The text file ends before expected. The current statement assumes that more text remains to be converted. Look at the command and make sure it specifies the correct arguments. Make sure the text file is complete, and has not been shortened (truncated).

31 File has an invalid format

The text file is not an ASCII text file. It is possible that it contains hidden characters added by the product originally used to create the file. Use that product to create a copy of the file that contains only ASCII characters. Then, convert that copy of the file.

32 Out of memory

Your computer ran out of memory while TXT2STF was loading a statement. It is possible that the statement is too big. It also is possible that there is not enough memory to execute this sequence of commands; see the possible solutions listed for error message 4.

33 Current text line is split because it's too long

The current line contains more than 512 characters and so TXT2STF must split the line to process it. TXT2STF processes the first 512 characters as one text line, and the remaining characters as another text line. If the current statement ends when TXT2STF is only partway through this second text line, TXT2STF discards the text line and continues to the next unprocessed text line in the text file. Unprocessed characters on the discarded second text line are never processed.

34 The exclamation point (!) is not a valid pattern character

The pattern contains an exclamation point (!), which is not a valid match-control character in a pattern. To create a negative pattern, use the tilde (~) and not the exclamation point (!). To search for the exclamation point (!) as a regular text character, place a backslash (\) immediately before the exclamation point (!).

Index