The Hitch Hiker's Guide to VAX Document

Order Number. PSQ-TDK-014

by Theo de Klerk
SWAS/Project Services

This Guide gives an overview of the different components that comprise the VAX DOCUMENT FT1.2 type setting system. It is an unofficial document written by a normally reasonably happy DOCUMENT user and administrator, especially while wearing Digital watches.

Revision/Update Information: V1.2 (6th edition, 11-MAY-1989 00:57:42.87)

DON'T PANIC

digital™
Digital Equipment bv
Utrecht, the Netherlands

May 1989

All rights reserved
Processed and prepared in the Netherlands

First edition: 1987 by Megadodoo Publications, Ursa Minor
Second edition: April 1988 by Digital Equipment bv
Third edition: May 1988 by Digital Equipment bv
Fourth edition: August 1988 by Digital Equipment bv
Fifth edition: August 1988 by Digital Equipment bv
Sixth edition: May 1989 by Digital Equipment bv

It is not the most remarkable, and certainly not the most successful book ever to have come out of the small publishing corporations of Ursa Minor. But it still is more popular than the Celestical Home Care Omnibus, worser selling than the VAX DOCUMENT Cookbook by the same author.

Although this document contains much that is apocryphal, or at least wildly inaccurate, it scores over the other VAX DOCUMENT books in two important ways. First, it is slightly cheaper, and second, it has the words "Don't Panic" inscribed in small ugly letters on the front cover.
It also incorporates some suggestions from the VAX DOCUMENT engineers.

This document was prepared using VAX DOCUMENT, Version 1.2

Preface

The first edition of this document was written after about one year of using VAX DOCUMENT, scanning its associated notesfile CLOSET::DOCUMENT and helping out people maintaining the product in their production environment.

Although the documentation supplied with VAX DOCUMENT is already very extensive, the topics in this document can only be appreciated after having used VAX DOCUMENT for a while and being familiar with its "heart", the TeX system.

This publication shows the relationship among several files (programs as well as data) produced and referenced by VAX DOCUMENT FT1.2 in its process from SDML source file to final printable output. It is by no means complete or definite. But everything in this publication is true, except the bits that are lies.

The third edition includes some corrections (removal of some lies) suggested by the VAX DOCUMENT engineers after reading the second edition. Everything they suggested may be considered true, but in addition I added something more myself that will go under the usual disclaimer of the previous paragraph (I'm really approaching DOCUMENT in a black-box fashion)

The fourth edition is revised to take some aspects of V1.1 into account. The differences are highlighted by changebars, which were automatically produced by comparing the current text with the original one by the CHANGEBAR program—another brainchild of mine, brought to perfection by Monty Sagal of the Israeli Local Engineering group.

The fifth and sixth edition adds information on the improved behaviour for foreign language string and hyphenation support made availabe to internal users in August 1988.

Hopefully it will guide the administrator in locating possible errors or problems and solving them quickly. For comments, improvements and suggestions, I invite you to drop me a VAXmail at IJSAPL::KLERK.

Thanks are due to Patti Anklam, Bill Kohlbrenner, Dave Parmenter on the VAX DOCUMENT front.

Amsterdam, May 9th, 1989

Theo de Klerk

Contents

Chapter 4 The Device Converter

Chapter 1
The VAX DOCUMENT program

```
$ DOCUMENT HHG2VAXDOC  IJSAPL.MANUAL  LN03/CONTENTS/INDEX
%DOC-I-IDENT, VAX Document T1.2     6-MAR-1989 14:28:26.99
```

1.1 Overview

The DOCUMENT command line identifies the three major components that play a part in the processing of your source file:

```
DOCUMENT  filespec  doctype  destination
```

The two main activities of the DOCUMENT program are:

1. Parse the command line and set up logical names
2. Invoke the other DOCUMENT components

The DCL command "DOCUMENT" invokes the VAX DOCUMENT program `DOC$TOOLS:-DOCUMENT.EXE`. This program will scan the command line, check its validity and setup information it will need in the subsequent phases and it will invoke the three main components of which the DOCUMENT system is built:

1. The Tag Translator
2. The TEXt Processor
3. The Device Converter

Each of these components is the subject of one of the next chapters. This chapter will concern itself with the main program `DOCUMENT.EXE` itself.

As the three components do no talk to each other directly, much of the information that is required for them is passed by the creation of logical names. The Tag Translator and Device Converters are separate shareable images. The Text Processor is an integral part of the `DOCUMENT.EXE` program.

1.2 Command line parsing

1.2.1 Filespec

The `filespec` parameter is the source file that contains the text to be formatted along with the type setting commands that are recognized by VAX DOCUMENT. These type setting commands are known as *generic markup* and more colloquially as *tags*. Appended to this file specification may be the qualifier `/PROFILE=filespec` in which case the file is considered to be just an element of a book to be built whose profile is specified with the qualifier.

A tag in the SDML source file is recognized by the fact that it is a name (consisting of alphanumeric characters and the underscore "_" and without blanks or tabs) enclosed by two angle brackets ("<" and ">"). Optionally the tag may be followed by one or more arguments. These arguments are enclosed in a set of parentheses ("(...)") and separated from each other by so-called backslashes ("\"). There can be no space or end-of-line between the tag and the arguments.

Some valid tag constructs are:

```
<TABLE>
<LIST>(NUMBERED)
<CHAPTER>(Once Upon A Time \openingchapter)
```

All valid tags are defined in the two VAX DOCUMENT User Reference Manuals. The first manual contains all tags that can be used in any text. These are so-called "global" tags. The second manual contains additional tag definitions that can only be used in the context of a special document type. For instance, a tag like <FROM_ADDRESS> makes sense when we're composing a letter, but not when writing a novel. Alternatively, <SLIDE> is only useful when making overhead transparencies.

The tags are interpreted by the Tag Translator component of VAX DOCUMENT. It will look up the definitons of all tags available for the specified doctype as supplied with the DOCUMENT software, and the user-defined tags. Getting familiar with the available tags is critical to making good use of VAX DOCUMENT. The Tag Translator also takes care of all the cross-referencing within the document.

As we will see in Section 2.2.2 the user may define tags that are only available locally and are "home made". This of course limits the use of the source file when it is going to be sent to computer systems on which these tags are undefined.

1.2.2 Doctype

The doctype parameter indicates how the text to be printed should be formatted. The word "doctype" is often used interchangeable with the word "design." A number of designs are delivered with the VAX DOCUMENT installation. Some of them are individual designs, such as *overhead* or *report*, others are members of a family of related designs. For a design in a family, you indicate both the family name and the specific family member name. For example, *software.specification* is the *specification* member of the *software* family of designs.

The specified name for the doctype is used by the DOCUMENT program to start looking for its validity. The available DOCUMENT doctype names are located a file called DOC$$FORMATS:DOC$DESIGNS.DAT. The DOC$$FORMATS is a logical name defined by DOCUMENT itself during run-time. It is a searchlist that is currently looking at DOC$PERSONAL_FORMATS, DOC$LOCAL_FORMATS and finally DOC$STANDARD_FORMATS. Hence DOCUMENT will search for DOC$DESIGNS.DAT in the following order:

```
DOC$PERSONAL_FORMATS:DOC$DESIGNS.DAT
DOC$LOCAL_FORMATS:DOC$DESIGNS.DAT
DOC$STANDARD_FORMATS:DOC$DESIGNS.DAT
```

These files are normal ASCII text files and can be edited using an ordinary editor such as LSEDIT or EDT or even TECO.

The three logical names DOC$*_FORMATS should translate into three (sub)directories and are scanned by DOCUMENT.EXE in the order given. Only DOC$STANDARD_FORMATS is defined by the standard installation. The other two do not need to exist. If they do, it is compulsory that the file DOC$DESIGNS.DAT, however small, is present in that directory.

The DOC$PERSONAL_FORMATS is defined privately and can contain private modifications of standard designs or entirely new designs.

The DOC$LOCAL_FORMATS is local to the entire VAX system, and is defined if the Digital internal kit is installed. It contains modifications to standard designs or new designs that are accessible to all users on the system, but are not part of the standard set.

The DOC$STANDARD_FORMATS is the default one supplied with VAX DOCUMENT. As such, it should not be modified.

If there are no private or local doctypes, the first two logical names should either not exist or point to the null device (NL:).

Each line in the DOC$DESIGNS.DAT files has an identical format:

```
    doctype /definitions=tagfile/macros=designfile/fonts=fontfile
```

The first match of the doctype given on the command line with a definition in one of those DOC$DESIGNS.DAT files will cause VAX DOCUMENT to use that definition. This way it is possible to overrule a standard doctype by defining the same name in the local or private version of DOC$DESIGNS.DAT. If the specified doctype is not found in either of these three files, VAX DOCUMENT will abort with a message saying that the doctype specified was unknown.

Local doctypes within Digital

The local design definitions can accomodate site-specific doctypes, but can also be modifications of existing doctypes that are defined in the DOC$STANDARD_FORMATS directory and are supplied with VAX DOCUMENT. Within Digital it is used to allow for company-specific modifications to certain designs (e.g. the MEMO is modified to be styled to the corporate standard interoffice memo layout).

Valid doctype definitions specify which tag definitions are to be used by the Tag Translator (through the /DEFINITIONS=), which design file (through /MACROS=) and what fonts to use by the TEXt Processor and Device Converter (through /FONTS=).

The DOCUMENT.EXE program creates a process logical name (during its execution) DOC$$DOCTYPE_MACROS to identify the design file specification.

We will talk about tag definitions in Chapter 2 and about doctype design files in Chapter 3.

1.2.3 Destination

The destination parameter indicates to VAX DOCUMENT what device converter should be used to produce the final pages after procesing by the TEXt Processor.

Using a line printer leaves only the standard monospaced fonts and very few degrees of freedom to make a pleasing looking page. It is useful however for line printer output, or terminal and mail readable text.

If an LN03 file is to be produced, all characters must now be "translated" into a dot pattern of white and black dots that together make for a nice looking character on paper. Which areas of the page remain white or become black is all defined in the LN03 font files that were bought separately through Digital from CompuGraphic. Hence the LN03 output files are not readable by anything but an LN03, or a translator, such as that provided with the PrintServer 40 and ScriptPrinter.

Using PostScript output requires another approach. Here, most font information is hidden in the PostScript printer itself. Like a Basic interpreter, this printer digests a string of PostScript commands and formats a page accordingly. Hence the PostScript output is human readable on a normal printer or video terminal, but doesn't make much sense unless you're very fluent in the PostScript language.

A destination only for Digital internal use is the VOILA_ONLINE bookreader output for DECwindows' Bookreader. This allows you to read a book from your workstation screen, layed out as if it were printed from an LN03 or PostScript printer. It should be used in combination with the ONLINE doctype.

All valid destinations for your command line are located in a system-wide file, maintained by the DOCUMENT Administrator. There used to be only one place where VAX DOCUMENT can find information on valid destinations. From FT1.2 onwards, DOCUMENT searches for the file `DOC$$FORMATS:DOC$DESTINATIONS.DAT` to find this information. Since this logical name is in fact a searchlist logical name, the destinations can be defined in a similar manner to the way doctypes are defined: you can have three locations for this destination information. Those three places are:

```
DOC$PERSONAL_FORMATS:DOC$DESTINATIONS.DAT
DOC$LOCAL_FORMATS:DOC$DESTINATIONS.DAT
DOC$STANDARD_FORMATS:DOC$DESTINATIONS.DAT
```

Similar to its search for a match with the specified doctype, DOCUMENT will also search through these three files for a match with the specified destination. The search is done through those files in the order specified above and on a first-match basis. This way by using identical names in the different files, one can overrule such a destination's characteristics.

The `DOC$STANDARD_FORMATS:DOC$DESTINATIONS.DAT` file is originally created and filled during the installation, but can be modified later, either by part-installing VAX DOCUMENT or by editing it manually. Modifications should happen in the other two files - the standard one should remain untouched.

Care must be taken not to add any blanks, tabs etc.[1] It contains information on

— the valid *destination* names

— to which physical queue on the system is associated with the destination

— additional print qualifiers (such as form type, setup,priority etc).

— the standard TEX macro format definition (FMT) file to use during the TEXt processing phase.

— the files containing special character font definitions

If the specified destination is not found in the file, VAX DOCUMENT will abort, reporting an unknown destination.

The DOCUMENT.EXE program will set up a logical name `DOC$$DEVICE_MACROS` to identify the file that contains the standard format (FMT) file with TEX macros to use during the TEXt processing phase. Inside these FMT files are references are made to the hyphenation rules for a particular language. The FT1.2 version comes with a set of foreign language hyphenation rules[2].

[1] the DOC$TOOLS:AUTODEST utility used to be available for this in BL8. However this tool proved full of gremlins. It will be replaced in a next release by another automated tool for modifying DOC$DESTINATIONS.DAT. Currently manual modification or part-installation is the only (error prone) way.

[2] V1.1 only supports American hyphenation

1.3 Quota problems

From the rough outline described in the previous sections, it will be clear that there are many data files, programs and reference files used during the execution of VAX DOCUMENT. In fact, this may be the reason why sometimes DOCUMENT aborts abruptly due to an exceeded diskquota. Any totally illogical error message (like access violations) should make you wonder about your diskquota.

Additionally, you must have at least the specified process quotas for number of open files (FILLM > = 30) and page file quota (PGFLQUOTA > = 25,000) for DOCUMENT to proceed correctly[3].

1.4 Summary

The DOCUMENT program performs the following activities:

1. Parse the DCL command line parameters:
    1. Verify valid doctype in `DOC$*FORMATS:DOC$DESIGNS.DAT` and setup logical name `DOC$$DOCTYPE_MACROS` for the design file
    2. Verify valid destination in `DOC$*_FORMATS:DOC$DESTINATIONS.DAT` and setup print queue information as well as the TEXt macro .FMT format file through a logical name `DOC$$DEVICE_MACROS`.
    3. Setup flags for production of index file, contents file, etc depending on other valid DOCUMENT command qualifiers.
    4. Open list and map file if required through /LIST and /MAP
2. When step 1 is successful and /BATCH is specified, create small command procedure to re-issue the same command in batch mode and submit it in the specified queue and exit to DCL.
3. When step 1 is successful, pass command to the Tag Translator
4. When Tag Translator completes successfully, pass command to TEXt Processor
5. When TEXt Processor completes successfully, pass command to the Device Converter
6. When Device Converter completes successfully, submit final print file to the printer queue, using appropriate qualifiers.

[3] This was 15,000 for V1.1

Chapter 2 The Tag Translator

```
[ T a g   T r a n s l a t i o n ]...
%TAG-I-DEFSLOADD, End of Loading of Tag Definitions
%TAG-I-ENDPASS_1, End of first pass over the input
```

2.1 Overview

In this chapter we will outline the actions taken by the first phase of the DOCUMENT system: the Tag Translator. This can be split into four parts:

1. Pass 0: Tag loading
2. Pass 1: Source file tag validation
3. Pass 2: Writing the .INT_TEX file
4. Pass 3: Post-Processing: writing the .TEX file

The Tag Translator itself is a callable routine in DOC$TOOLS:TAG$TRANSLATESHR.EXE shareable image and it called from the main program of VAX DOCUMENT.

2.2 Pass 0: Tag loading

Pass 0 of the Tag Translator is the loading of all required tag definitions and it opens the output file with file type .INT_TEX that will eventually contain all original source SDML text with all tags replaced by their definitions. Although during pass 0 this file is opened, it will normally not be used until pass 2. When we're in a book building process, a separate .INT_TEX file is created for each of the elements of the book and the profile itself.

The loading of tags consists of reading a minimum of two files:

1. The doctype specific definitions
2. The user-defined (or site-specific) tag definitions

More files are read if one of the above contains <INCLUDE> tags.

Doctype Specific and Standard Tag Definitions

First, the doctype specific tag definitions are read in. During the validation of the doctype command parameter, the the DOCUMENT program has found a valid entry in the DOC$DESIGNS.DAT file where the file with doctype specific tag definitions was defined through the /definitions= qualifier. The filename specified defaults to file type .STT and must be present in the DOC$*FORMATS directory where the matching DOC$DESIGNS.DAT file was also found.

Although not necessarily true, all doctype tag definition files currently start with an

```
<INCLUDE>(DOC$STANDARD_FORMATS:TAG$SDML_TAGS.STT)
```

tag that will force the Tag Translator to start to process that file. It contains the "global tags" of VAX DOCUMENT, that are available to all doctypes.

After the reading of this "standard tag" file, the remainder of the doctype tag definition file is processed. This contains additional tags, specific to the doctype, redefinitions of some of the standard ones or orders to "hide" several tags to make them unknown.

Tag definition files either have the file type .STT or .GDT[4]. Here .STT stands for *Saved Tag Table*. This file is a binary version of its source file: a .GTD file. The .GTD stands for *Generic Tag Definition*.

Private Tags

Finally, additional private tags will be loaded from the file that is referred to by the (process)[5] logical name TAG$LOCAL_TAGS. This file can also be specified with the /SYMBOLS= qualifier on the command line. Alternatively, using this qualifier allows the definition of a a number of symbols through the <DEFINE_SYMBOL>(name\text) tag. All the files with tag and symbol definition are read in only once. This is in contrast to the file specified as the SDML file to be processed and any files mentioned in that file through <INCLUDE> or <ELEMENT>. These will be scanned twice: once in each of the passes 1 and 2 of the Tag Translator.

2.2.1 Difference in STT and GTD files

Using the binary STT version of tag definitions allows faster loading of the tag definitions[6]. You can use either the .STT or .GTD file. The Tag Translator is insensitive to the file type, as both are processed by exactly the same code, which is also the same code that processes the normal SDML tags.

Normally there is no normal "free" text in the tag definition files: just a (large) set of <DEFINE> tags that define the tags. If ordinary text is written in the definition file, it is output into the .INT_TEX file in the same way as later during pass 2 the text of the SDML file is copied unchanged.

An STT file can be loaded much faster because all the main tags such as <DEFINE> and <STRING> are encoded in a special format. These definitions are moved straight into the internal tag table without further processing. This loading is triggered by the <LOAD_TAGS> that usually is written at the start of the STT file. If non-encoded text is encountered, the Tag Translator exits the fast loader and returns to the normal processing routines that "compile" the tags into the encoded format and then load them into the table. The rapid loading resumes when another <LOAD_TAGS> is encountered in the file. As such, it is not the filetype STT or GTD that triggers special processing, but rather the presence of the tag <LOAD_TAGS> inside the file. However, it is advised to adhere to the STT and GTD difference for clarity.

[4] Older files used GDX as file type

[5] Due to a mistake in the Tag Translator code, this logical name is used only when it is specified in the process logical name table in V1.0

[6] Compare TPU section files with TPU Command files

When there are only a few tags defined in a definition file, the time saved by reading the pre-compiled STT file instead of the source GTD file is minimal. With a large number of tag definitions, it is substantial. All tag definitions that come with the VAX DOCUMENT kit are only available in STT format. As advantage to the development group, this will discourage more people from modifying them to produce a maintenance nightmare.

2.2.2 User created and modified Tag Definitions

When customizing designs or creating private, house-style, designs, there may be a need for local tags. The process of defining tags is not available or documented for customers at this moment. For Digital internal users, there is a special manual called the *Tag Designer's Guide* available that explains in detail how the Tag Translator works and how a proper tag definition is made using the <DEFINE> command.

The source file containing tag definitions is normally a .GTD file: Generic Tag Definition. When there are only a few tags present, it is not worth while to make an STT file. Rather, the Tag Translator can read the GTD definitions and compile them each time they are read in.

Alternatively, with many tag definitions, the GTD file can be compiled into a binary version, the .STT Saved Tag Table file by processing the GTD file by VAX DOCUMENT by the command:

```
DOCUMENT my_doctype.GTD  DOCTYPE_STT  TAG_TABLE/KEEP=INT_TEX
```

or

```
DOCUMENT my_doctype.GTD  GLOBAL_STT  TAG_TABLE/KEEP=INT_TEX
```

The doctypes DOCTYPE_STT and GLOBAL_STT are recognized by VAX DOCUMENT if you have installed the Digital Internal kit (DOCINT012.A and DOCINT012.B).

These "doctypes" are not available to customers. In fact, the DOC$DESIGNS.DAT file in DOC$LOCAL_FORMATS contains the following two entries to allow these doctypes to be valid:

```
!
! Definition files for creating Saved Tag Table (STT) Files
!
name GLOBAL_STT/definitions=TAG$BUILD_GLOBAL.GDX
name DOCTYPE_STT/definitions=TAG$BUILD_DOCTYPE.GDX
```

where both GDX files (old name for GTD) contain only one line that forces DOCUMENT to save all tags in an STT file:

```
<SAVE_TAGS>
```

in the TAG$BUILD_DOCTYPE.GDX file or

```
<SAVE_TAGS>(ALL)
```

in the TAG$BUILD_GLOBAL.GDX file. This tag must be encountered during Pass 0 of the Tag Translator and will cause the Pass 1 to be bypassed, and go into Pass 2 in which the .INT_TEX file is written. During this pass 2, it is processed the same way as a normal SDML file. During pass 0 the <SAVE_TAGS> sets up a special flag that is checked by the Tag Translator in pass 2 and causes it to output an STT file rather than a TEX file for further processing.

The destination TAG_TABLE was added to the DOC$DESTINATIONS.DAT file during installation of the internal kit:

```
DESTINATION TAG_TABLE TAG -
/exclude_action=(text_formatter,device_converter,print) -
/output_filetype=.STT
```

There is a difference in the output of GLOBAL_STT or DOCTYPE_STT. When making a new doctype that is based on an existing one, you may wish to copy all the corresponding tags of that doctype and add your own. In this case, your tag source file will start with

```
<include>(standard_tag_filespec)
<define>(my_own_tag\  ...)
```

Specifying DOCTYPE_STT will ignore the <INCLUDE> tags during tag compilation and only use the <DEFINE> tag definitions to make binary output. The <INCLUDE> tags are copied literally into the final STT file. When the Tag Translator reads your STT file, it will see the <INCLUDE> tag and starts reading the specified file first. This way your own STT file can remain small and it can benefit from any addition or correction made to the STT files it refers to.

By specifying the GLOBAL_STT doctype, it will not ignore the <INCLUDE> tag during tag compilation. The final STT file will contain binary versions of all tags: those taken from the <INCLUDE> files and those <DEFINE>d yourself. Hence the final STT file will be much larger, but will not contain any <INCLUDE> tags: all tags are available in one single STT file. Due to fewer file lookups this is obviously faster but also means that corrections made to the <INCLUDE>d tag definition files are not automatically used by your STT version.

2.2.3 The Tag Designer's Guide

This book (for Digital internal use only) explains how to redefine an existing tag or to create a new tag that in effect is a combination of other, standard, tags and is compulsory reading for anyone who wants to define tags or create tags that are "short hand" for longer combinations of existing tags.

The true designer will also want to create tags that will translate into TEXt Processor macros to be further interpreted by the TEXt processor. Unfortunately, this is not described in the Tag Designer's Guide, but needed nonetheless. As TEXnician (term defined by Donald Knuth who "invented" the original TEX program), you need to be able to output TEX macros which are typically written as a name preceeded by a backslash ("\") and have their arguments enclosed in braces. For example:

```
\bottomline{For Internal Use Only}
```

is a straight translation of the original SDML source file code

```
<BOTTOM_LINE>(For Internal Use Only)
```

by the Tag Translator. In order to do this, a tag <BOTTOM_LINE> must be defined in a GTD or STT file. In this example that definition would look like:

```
<DEFINE>(BOTTOM_LINE\| ^Tbottomline^B$1^E &\1\1)
```

You can see that the backslash is represented by a CTRL/T character and the braces "{" by CTRL/B and "}" by CTRL/E respectively. For more specific information on the <DEFINE> tag, we direct you to the Tag Designer's Guide.

To supplement the manual[7], the characters used in the tag definitions to output valid TEX macro definitions or invocations are listed in Table 2-1 There is a special utility that makes the processing of the GTD files easier. It allows to process the original GTD (or GDX) file into a GDE (edit) file, which is the same file, but with all control characters

[7] Table 2-3: Built-in Tag Alias Codes

replaced by normal printable characters, as indicated in the "GDE file representation" column. This way the CTRL/B will be replaced by "{", while the CTRL/T is represented as a tilde.

Table 2-1: TEX built-in tag alias codes

TEX symbol	Control char	ASCII value	GDE file representation	Description
&	CTRL/A	1	^&	Table column separator
{	CTRL/B	2	{	TEX macro argument start delimiter
%	CTRL/C	3	%	TEX comment on rest of the line
$	CTRL/D	4	^$	TEX mathematical mode delimiter
}	CTRL/E	5	}	TEX macro argument end delimiter
=	CTRL/Q	17	^=	TEX macro assignment
\\	CTRL/R	18	^R	TEX new line when in horizontal mode
\	CTRL/T	20	~	TEX macro escape introducer

Hence, a TEX macro like \hbox will be written as ~hbox.

The conversion from GTD (or GDX) to GDE file is done through @DOC$LOCAL_TOOLS:-EDITGDX.COM.

When you've completed the editing, you can restore the proper tag definition file by converting the TEX characters into their control characters through @DOC$LOCAL_TOOLS:-FIXGDX.COM.

2.3 Pass 1: Source file tag validation

Once all tag definitions are loaded during Pass 0, Pass 1 starts. It performs the following actions:

- Check tag definitions of SDML source file and perform the Pass 1 definitions of the tags
- Write label information in cross-reference file
- Open table of contents and index files when requested
- Write map file and close at end
- Write diagnostic file when required (through LSE or /DIAGNOSTIC)
- Signal message to user when the pass is completed.

Pass 1 opens the user source file (.SDML) and performs all Pass 1 specific actions specified for each tag it encounters. When invalid tags occur, this is reported and the Tag Translator will abort after Pass 1.

When specified, also the index file and table of contents file are created. At the end of pass 1, the map file is closed when it was required through /MAP and opened by the main program.

The first pass also opens an .XREF file to build a cross reference table for all labels encountered while processing the SDML file, such as in <CHAPTER>(chapter title\chapterlabel). These labels can be read in the cross reference table by the second pass when output is written to the .INT_TEX file. During Phase 1 no output is normally written to this file.

This first pass does not produce any other output file and is used only for syntax checking and setting some internal counters or flags, that will be used in Pass 2. Tag definitions may specify specific actions to perform only in Pass 1 or only in Pass 2. Usually, they are the same.

It is important to realize that the resolving of any references is done in the Tag Translator and *not* by the TEXt Processor, as most other TEX-based systems do. It is also the Tag Translator that determines the numbering of the chapters, tables etc[8].

2.4 Pass 2: Writing the .INT_TEX file

This pass is similar to Pass 1. However, in this pass, the .INT_TEX file, opened in Pass 0, is filled. All tags are replaced by their Pass-2 definitions as specified in the Saved Tag Table files. The <REFERENCE> tags are resolved from the .XREF file created during the first pass.

2.5 Pass 3: Post-Processing: writing the .TEX file

The Post Processor is called and opens the created .INT_TEX file and produces the final .TEX file. During book building, several .TEX files are created: one for each element, and one for the profile file. The Post Processor doesn't read any files other than the .INT_TEX files. Amongst others, it replaces the CTRL/T character by the true TEX backslash character and replaces MCS characters by special TEX macros to support them.

After the TEX files are created, the TEXt Processor is called.

2.6 Summary

The Tag Translator uses the following files:

- Pass 0: Read tag definitions
  1. Create the .INT_TEX file
  2. The doctype specific STT file as indicated in the DOC$DESIGNS.DAT file with the /definitions=tagfile. This normally also includes reading the DOC$STANDARD_FORMATS:TAG$SDML_TAGS.STT that contains all doctype-independent tag definitions[9]
  3. Read the file pointed to by logical name TAG$LOCAL_TAGS and/or /SYMBOLS=file for private tag defs[9]
- Pass 1: Validate SDML source file on tags specified. Create .XREF file and close MAP file when it exists.

[8] This explains also why it will be difficult to implement a reference to a particular page by VAX DOCUMENT, since the Tag Translator hasn't got a clue where the TEXt Processor will actually position the text

[9] Also additional <INCLUDE>(tagfiles) are processed

- Pass 2: Fill *source_file*.INT_TEX from source file and resolve references from .XREF file
- Pass 3: Post Processor creates *source_file*.TEX and inserts special macros to support MCS characters.

Chapter 3
The TEXt Processor

```
[ T e x t   F o r m a t t i n g ]...
%TEX-I-LINETOOLONG, Line too long by 1.51584 points
-TEX-I-ONPAGE, on page [3-6]

%TEX-I-LINETOOLONG, Line too long by 1.51584 points
-TEX-I-ONPAGE, on page [3-6]

%TEX-I-PAGESOUT, 28 pages written.
-TEX-I-OUTFILENAME, 'PSQ:[PRIVATE.THEO]HHG2VAXDOC.DVI_LN03'
[ C o n t e n t s   G e n e r a t i o n ]...
[ T e x t   F o r m a t t i n g   C o n t e n t s ]...
%TEX-I-PAGESOUT, 2 pages written.
-TEX-I-OUTFILENAME, 'PSQ:[PRIVATE.THEO]HHG2VAXDOC_CONTENTS.DVI_LN03'
[ I n d e x   G e n e r a t i o n ]...
%INX-I-ENDPASS_1, End of first pass over input file:
'PSQ:[PRIVATE.THEO]HHG2VAXDOC.INX'
%INX-S-ENDPASS_2, End of second pass over input file.
%INX-S-CREATED, 'PSQ:[PRIVATE.THEO]HHG2VAXDOC_INDEX.TEX;1' created
[ T e x t   F o r m a t t i n g   I n d e x ]...
%TEX-I-PAGETOOSHORT, Page too short - on page [INDEX-2]
%TEX-I-PAGESOUT, 2 pages written.
-TEX-I-OUTFILENAME, 'PSQ:[PRIVATE.THEO]HHG2VAXDOC_INDEX.DVI_LN03'
```

3.1 Overview

Once the SDML files have been converted into TEX files by translating the specified tags into TEXt Processor commands, the TEXt Processor is called to read the TEX file and corresponding design files to format the text into the so-called *device independent* DVI files.

Just as the Tag Translator recognizes a collection of "global" and "doctype specific" tags, the TEXt Processor knows a standard set of commands (known as *macro definitions*) and doctype specific commands. Unlike the Tag Translator, the Text Processor is an integral part of the DOCUMENT.EXE image.

For those familiar with the TEX system of Donald Knuth or the LaTEX variety by Leslie Lamport, the TEXt Processor will look very familiar, yet it contains some alterations from the "standard" TEX program. In fact, DOCUMENT's TEX version no longer passes the "TEX test suite" and may hence not be called TEX. As such, plain TEX or LaTEX files cannot be processed by VAX DOCUMENT. One of the main differences is that the TEXt Processor is not used for cross reference resolving. This has already been done by the Tag Translator. Also the assignment of chapter numbers, table numbers and such has been accomplished by the Tag Translator.

The TEXt Processor scans the source .TEX file. It

1. Reads all required hyphenation patterns and TEXt macro definitions from various sources
2. Processes the source TEX file in one pass and writes information into a table of contents file or index file if the /CONTENTS or /INDEX were specified.
3. Processes the table of contents file and sorts the index file. Also outputs a separate contents and index file if the /CONTENTS and /INDEX qualifiers were specified.

3.2 Loading TEXt macro definitions

The TEXt Processor reads all required macro definitions from several sources. A "standard" set of files is read, but others are read only when triggered by an \input command in such a TEX file. Whenever only a filename and type is specified, the TEXt Processor uses the logical name DOC$$FORMATS: as a search list logical name. This logical name is defined during the execution of DOCUMENT and currently defaults to a search list of the three DOC$*_FORMATS: logical names.

The files normally read are

1. Standard VAX DOCUMENT macros and hyphenation pattern
2. Additional locally (re)defined macros
3. Document-specific macros

The last two are read from DOC$STANDARD_FORMATS:TEX$STARTUP.TEX which in turn invokes a set of other files. The first one "comes naturally" with the invokation of the TEXt Processor.

3.2.1 Standard VAX DOCUMENT TEX commands

The TEXt Processor starts by reading a file that contains all standard available macro definitions. These files are also known as "format file" and have extentions .FMT_MACRO. Which file to use, is defined in the file DOC$DESTINATIONS.DAT for each destination specified. The main program DOCUMENT.EXE has set up a logical name DOC$$DEVICE_MACROS to point to the one needed during processing.

When this does not specify a full file specification, DOCUMENT prefixes it with the logical name DOC$$FORMATS: which it has defined itself as being a searchlist of the three DOC$*_FORMATS: logical names.

The destination-dependent format file allows you to use different "standard" definitions, depending on the destination chosen.

A typical entry (for PostScript) in the destination file is:

```
DESTINATION POSTSCRIPT PS -
 /dvi_filetype=.DVI_PS -
 /exclude_action=(print) -
 /fonts_filetype=.POST_FONTS -
 /format_file=SDMLPS -
 /output_filetype=.PS -
 /print=(queue=PS_LPSPG,parameter=data_type=postscript) -
 /special_characters=TEX$PSCHARS
 parameter number HORIZONTAL_OFFSET
 parameter number VERTICAL_OFFSET
 parameter string STARTING_PAGE
 parameter string ENDING_PAGE
 parameter number NUMBER_OF_PAGES
```

The /format_file= entry specifies the macro format file to use. The standard set of format files provided by the VAX DOCUMENT kit are found in DOC$STANDARD_FORMATS: directory:

```
DOC$STANDARD_FORMATS:SDMLLPR.FMT_MACRO
DOC$STANDARD_FORMATS:SDMLLPS.FMT_MACRO
DOC$STANDARD_FORMATS:SDMLHEBREW_PS.FMT_MACRO
DOC$STANDARD_FORMATS:SDMLLN03.FMT_MACRO
DOC$STANDARD_FORMATS:SDMLVOILA.FMT_MACRO
```

depending on the fact whether it is a line printer, PostScript or LN03 type destination. The Hebrew version is special, since the Hebrew language runs from right to left. It is made only in PostScript destination.

Other format files are also read by the TEXt Processor, depending on the value of the logical name DOC$LANGUAGE. Given a valid translation, it will also read the format file containing the corresponding hyphenation rules, specified in *language*.FMT_HYPH[10].

Whenever hyphenation errors occur (since the rules only catch about 95% of all word hyphenations) these can be rectified within the file DOC$STANDARD_FORMATS:TEX$EXCEPTIONS.TEX which is read as one of the last files.

These FMT format files are a binary version of their original source files, DOC$ELEMENTS.TEX and DOC$SDML.TEX that were compiled by a special version of the TEXt Processor, known as INITEX. This SDML*.FMT_MACRO file contains definitions for all primitive ("simple") TEXt Processor macros such as those that specify "new paragraph", "indent left margin".

Many macro definitions will resemble the "plain TEX" definitions but many new ones are defined as well. Unfortunately, the VAX DOCUMENT group has sofar refused to publicize their macro definitions, so you must find out by guessing and trial and error what their specific impact is. In fact they discourage you to fiddle around with macros and a special *doctype design tool* (appropriately called DDT) is in preparation for a next release, so that the TEX layer will be invisible for almost all users other than the born die-hard hacker type[11].

3.2.2 TEX$STARTUP.TEX: additional definition files read

After reading the standard definitions from *.FMT_MACRO and *.FMT_HYPH the TEXt Processor reads the file DOC$STANDARD_FORMATS:TEX$STARTUP.TEX next. This file only contains some additional \input commands to read other files. It contains:

```
\input DOC$LOCAL_ELEMENTS:
\input DOC$$DOCTYPE_MACROS:
\input DOC$$DEVICE_MACROS:
\input DOC$$FORMATS:TEX$EXCEPTIONS.TEX
```

The files specified (either explicitly or by their logical names which were defined by DOCUMENT at run time) may in turn call other files to be read and included too. The first one loads the additional site- or company specific definitions, the next loads the definitions that go with the selected document design and the third reads special definitions to match each character typed with a particular character from the fonts used for the selected destination. The last one reads a list of words that the hyphenation rules don't catch correctly and need "manual overrule". This list can be empty or very long, depending on the quality of the hyphenation rules supplied.

[10] In V1.1 hyphenation was supported through a separate destination rather than by logical name. Eventually it will all depend on <LANGUAGE>(language)

[11] Some of TEX's processing can be seen when you define the global symbol TEX$$HACKER= ="TRUE" before you start processing. At least the name is obvious.

3.2.2.1 DOC$LOCAL_ELEMENTS: Site-specific modifications

This logical name either points to the NL: null device or to some .TEX file that defines additional standard TEX macros or redefines them. This is currently the case if you have the Internal Kit installed. The logical name refers to a file (CUP$LOCAL_ELEMENTS.TEX) that contains additional macro definitions such as the Digital logo. If you have the European language support installed, the logical name points to CUP$EUROPEAN_ELEMENTS.TEX which starts off by including the "general Digital" file CUP$LOCAL_ELEMENTS.TEX and then goes on by including yet another file, TEX$LANGUAGE_STRINGS.TEX which contains redefinitions of standard heading words written by DOCUMENT such as "chapter" and "table" into the language specified by the <LANGUAGE>(language) at the begin of the source SDML file. ("Hoofdstuk" instead of the default "Chapter" header text by redefining the \Chaptername macro's string value).

Unlike the standard *.FMT_MACRO file, the remaining files that the TEXt Processor uses are all in readable TEXt format. Each of these can contain additional \input macros to read additional files. This may partly explain the relatively long time VAX DOCUMENT spends in the text formatting phase as all macros must be internally compiled first[12]. The algoritms used to make a "pleasing looking page" also consume quite a bit of CPU time as MicroVAX users will no doubt have discovered.

3.2.2.2 DOC$$DOCTYPE: Standard Doctype-specific macros setup

From the command line, the DOCUMENT main program has determined the doctype specified.

From the DOC$DESIGNS.DAT file where it found the doctype definition, it also found the file that the TEXt Processor must read to obtain all doctype specific information. A process logical name was set up to indicate the file in question: DOC$$DOCTYPE_MACROS. This doctype specific macro file was indicated by the /macros=*filespec* qualifier on the doctype definition line, that specifies the filename of the doctype design file.

The design file's full specification is *filename*.DESIGN. It is found in the directory of the DOC$DESIGNS.DAT file that contains the doctype definition.

It contains general formatting information about page size, left and right margin, footers, headers, chapter formatting and such. It also contains information on what sort of fonts to use for headers, plain text, code examples etc.

More information on this is found in the *VAX DOCUMENT Doctype Designer's Guide*. This book is also available to customers. In addition to this, the book *The TEXbook* by Donald Knuth is indispensable for a good understanding of the TEX program (even if DOCUMENT uses a modified version).

Again, when the doctype design file is based on another design file, you will see that the current doctype .DESIGN file will start with a \input macro and then continues to add or redefine several macros that make the difference between the standard and "custom-made" design file.

[12] This is a similar situation as with the Tag Translator that must process GDT files instead of just loading the STT

3.2.2.3 DOC$$DEVICE_MACROS: output device specific definitions

The third macro definitions referenced by TEX$STARTUP.TEX is one that contains device specific macros (hence always the same ones regardless of doctype but dependent on destination). The DOCUMENT main program has set up a logical name DOC$$DEVICE_MACROS and it translates to a .TEX file, as found in the destination definition in DOC$DESTINATIONS.DAT with the /special_characters=. With the current FT1.2 kit, these qualifiers all point to either one of the following filespecs:

```
DOC$LPR_FONTS:TEX$LPCHARS.TEX
DOC$POST_FONTS:TEX$PSCHARS.TEX
DOC$LN03_FONTS:TEX$LN03CHARS.TEX
DOC$VOILA_FONTS:TEX$VOILACHARS.TEX
```

These files contain macro definitions that are introduced by the Post Processor during the Tag Translation Phase and define specific characters and their position (0—255) within the font (e.g. all <MCS> characters). The LPR file contains additional page sizing setups.

3.2.2.4 TEX$EXCEPTIONS.TEX: hyphenation exceptions

The final file referenced by TEX$STARTUP.TEX is one that contains "corrected" words that were not hyphenated correctly by the used hyphenation rules. Mostly, the contents of this file depends on experience and discovery, as about 95% of the words will be hyphenated correctly by DOCUMENT. Thereby DOCUMENT has the best hyphenation rules available of all DTP typesetting programs. Most come only near the 70%.

Depending on the <LANGUAGE>(language) definition, the Tag Translator has generated a TEX macro *LANG*textstrings where "LANG" are the first four characters of the language specified (e.g. DUTC for dutch, GERM for german). This will invoke the redefinition of all standard headings in TEX$LANGUAGE_STRINGS.TEX but also in this TEX$EXCEPTIONS.TEX file it will invoke the macro \LANGexceptions. For dutch[13] e.g.

```
\gdef\DUTCexceptions{
\hyphenation{
    voor-deur-de-lers-re-ge-ling
    be-drijfs-klaar
}
}
```

Additional words can be added. The hyphen indicates where DOCUMENT should hyphenate the word. No other locations than those specified are used[14].

3.3 Processing source TEX file

After having loaded the standard macro definitions, local elements, and all doctype specific design macros, the TEXt Processor finally starts to read the user's source file that was produced by the Tag Translator. In principle no errors should be found in this process, if we assume that the tag definitions had correct TEXt macro translations. If the end-user does discover errors, he should refer to the doctype designer, as there is little he can do about the error himself.

[13] This language is "forgotten" in FT1.2 - well, it shows no one is perfect

[14] The exceptions used to be included in TEX$LANGUAGE_STRINGS.TEX in V1.1 and in fact are still present there in FT1.2

Note

In this context it is useful to know that any line number that is mentioned in the error messages by the TEXt Processor refer to line numbers of the TEX file, not those of the original SDML file. By looking into the TEX file the neighboring text will probably quickly point you to the corresponding lines in the SDML file and limit the number of invalid or incorrectly coded <TAG> definitions.

The TEXt Processor reads the input TEX file paragraph by paragraph. Using a special bonus/malus point system[15], it computes the best looking setting of each paragraph, using lines that are as full as possible with as few hyphenations as possible. The afore mentioned *TEXbook* contains full details on this.

When the TEXt Processor encounters an `\input` command in the user's source file (which is the translation of the undocumented but highly popular <INCLUDE_TEX_FILE>(filespec)) it temporarily suspends processing the current TEX file and starts reading in and processing the specified `\input` TEX file.

Note

Using the <INCLUDE_TEX_FILE> tag is yet another way to redefine on a personal basis any definition made in any of the loaded TEX macro definition (design) files.

To determine how much will fit on a line, the TEXt Processor needs to know the width and height of each individual character that will appear in the final print. This information is stored in so-called TEXt Font Metric files. Judged on this size information, the TEXt Processor can fill out lines and pages and can complain about "underfull" and "overfull" lines when it cannot find suitable points for hyphenation. It does this by putting each character in a box of its own. A word is a horizontal box consisting of several character boxes glued together. A line is several word-boxes together and a page is a number of line boxes stacked on top of each other. The whole "page building" is kept together by "glue". And TEX keeps stretching and shrinking the glue to make everything look nice on the page. Once that is done, the page is "frozen" and the glue becomes "rigid". Then the page is written into the DVI file.

Depending on the destination chosen, the TEXT Font Metric files are found in one of the following four directories:

```
DOC$LN03_FONTS:*.TFM
DOC$LPR_FONTS:*.TFM
DOC$PS_FONTS:*.TFM
DOC$VOILA_FONTS:*.TFM
```

These directories also contain the actual font files, with filetype `.NFT` (Normal FonT — TEX could also handle enlarged or magnified fonts) or `.*PXL` pixel fonts.

The required fonts are specified in the doctype's `.DESIGN` file, and for the standard doctypes they are summarized in Appendix B of the Doctype Designer's Guide. There you will find symbolic names like `\normaltextfontspecs` to refer to a particular font macro, e.g. `\tenpoint`. This `\tenpoint` is associated with a true physical file in `DOC$*_FONTS` by combination information from the `DOC$DESTINATIONS.DAT` and `DOC$DESIGNS.DAT` files. The filename is found in the definition of the doctype in `DOC$DESIGNS.DAT` through the `/fonts=`. The associated file type is found in the destination definition in `/fonts_filetype=`. The splitting of name and type allows to use the same file name but different types for different destinations. By using the same doctype but a different destination, different font definition files can be referenced.

[15] Referred to by me as the "scrabble routine". Since it looks only for letter combinations, there is no huge dictionary needed of words. This makes hyphenation by TEX a lot faster than dictionary look-ups. See the TEXbook Appendix H for more details on hyphenation.

In the current FT1.2 set, the names are always one of the following four:

```
DOC$STANDARD_FORMATS:TEX$STANDARD_FONTS.LINE_FONTS
DOC$STANDARD_FORMATS:TEX$STANDARD_FONTS.POST_FONTS
DOC$STANDARD_FORMATS:TEX$STANDARD_FONTS.LN03_FONTS
      DOC$STANDARD_FORMATS:CUP$VOILA.VOILA_FONTS
                              |                |
                    from doc$designs.dat   doc$destinations.dat
```

In the TEX$STANDARD_FONTS.LN03_FONTS file you will find that e.g.

```
\readfont\tenex{amx10}
```

meaning that wherever the font \tenex is used in fact the characters are to be output are stored in the file AMX10.NFT while their metrics are stored in AMX10.TFM. When the PostScript output is used, the fonts refer to its own resident fonts, e.g. the Times font.

Note

As a slightly more complicated example on the standard font: the \normaltextfontspecs translated to \tenpoint. This in turn translates into \let\tenpoint=\paltenpoint. This means that this definition is identical to \paltenpoint. And that translates to (some parts were omitted)

```
   %
   %  Process \paltenpoint
   %
1  \readfont\paltenrm{palac10}        % text
   \readfont\paltenit{palaci10}       % italic
   \readfont\paltenmi{ammi10}         % math italic
   \readfont\paltenbf{palacb10}       % bold
   \readfont\paltenbi{palacbi10}      % bold italic
   \readfont\paltensy{amsy10}         % math symbols
2  \def\paltenpoint{%
3    \normalbaselineskip=11pt%  leaded one point
                :
4    \def\rm{\fam0\paltenrm\let\superfam=\rm}%
     \def\it{\fam\itfam\paltenit\def\bf{\bi}\let\superfam=\rm}%
     \def\bf{\fam\bffam\paltenbf\def\it{\bi}\let\superfam=\bf}%
     \def\bi{\fam\bifam\paltenbi\let\superfam=\bf}%
                :
   }%
```

with the following annotation:

1 Font associated with \paltenrm

2 start of definition of the \paltenpoint macro

3 defines the standard line width. It is taken 11 points: one point more than the font height, leaving a little space between lines.

4 defines the "roman" appearance of the tenpoint font when the \rm is postfixed to \tenpoint or \paltenpoint . Then the \paltenrm font is used. This is already defined on the line marked with callout 1. Subsequent lines define the *italic*, **bold**, bold-italic, (as opposed to bold, italic or small capitals or sans serif), which refer to fonts defined at the top of the example.

It is possible to define your own set of fonts by modifying the above font definition files, and to create new fonts (or adapt existing ones from the Public Domain) using a special program called MetaFont. You must however make sure that all font characters are positioned exactly on the same locations as they are with the supplied fonts of VAX DOCUMENT. Needless (?) to say that, especially in symbols and MCS characters, the default MetaFont output does not put characters at the same font position as where VAX DOCUMENT expects them...

There are two books by Donald Knuth that describes this program: *MetaFont book* and *MetaFont: The Program*. The program produces both the required TFM font metric file and the required printer font file. Designing fonts is a complicated business and requires more than computer skills.

Wherever figures are to be included through the <FIGURE_FILE>, <ICON_FILE> and similar tags to include figures that are not to be touched during text processing, the TEXt Processor reserves the indicated amount of blank space and inserts a so-called "special" command into the DVI file to be produced. This "special" command is recognized by the device converter and it will include the specified graphics file in the final output without further modification. This "blind spot" or ignorance on the TEXt Processor's part also explains why text and graphics can run into each other or overlap if insufficient space is reserved for inclusion.

Finally, the following files are produced by the TEXt Processor:

- A *source*.DVI_device file
- A *source*_CONTENTS.DVI_device index file
- A *source*_INDEX.DVI_device index file

The file extentions are again defined in the `DOC$DESTINATIONS.DAT` file through the `/dvi_filetype=` qualifier.

Depending on the presence of the <INDEX_FILE> or <CONTENTS_FILE> tags, the index and contents files will be included in the final printer file by the Device Converter or they will be processed as separate printable files.

3.4 Summary

The following files are read or written by the TEXt Processor:

1. Read `DOC$STANDARD_FORMATS:*.FMT_MACRO`: standard TEX macros for the printer as indicated in the destination definition
2. Read the `DOC$STANDARD_FORMATS:*.HYPH` file that contains the hyphenation pattern indicated by logical name `DOC$LANGUAGE`
3. Read `DOC$STANDARD_FORMATS:DOC$STARTUP.TEX` that in turn invokes:
    a. Read file by logical name `DOC$LOCAL_ELEMENTS`: Add or modify macros on local system (the file refered to may do several `\input` ***file***).
    b. Read `DOC$*FORMATS:`*doctype*`.DESIGN`: Doctype specific macros
    c. Read the device font definition file belonging to the destination selected
    d. Read the file `DOC$*FORMATS:TEX$EXCEPTIONS.TEX` containing corrections on incorrectly hyphenated words
4. Read any file specified by <INCLUDE_TEX_FILE> in SDML source
5. Write *source*`.DVI_destination` file
6. Write *source*`_CONTENTS.DVI_destination` file
7. Write *source*`_INDEX.DVI_destination` file

Chapter 4
The Device Converter

```
[ D e v i c e   C o n v e r s i o n ]...
%DVC-I-INCLUDING, including input file:
PSQ:[PRIVATE.THEO]HHG2VAXDOC_CONTENTS.DVI_LN03;
%DVC-I-INCLUDING, including input file:
PSQ:[PRIVATE.THEO]HHG2VAXDOC_INDEX.DVI_LN03;
%DVC-I-PAGESOUT, 27 pages written to file:
PSQ:[PRIVATE.THEO]HHG2VAXDOC.LN03
```

4.1 Overview

The device converter is by far the simplest component from the user's point of view: there is nothing we can do to influence it other than the /DEVICE qualifier on the command line. It will pick up the DVI file produced by the TEXt Processor and produce the final output file. It may include table of contents, index and graphics files along with the main text. The final file type is again defined in `DOC$DESTINATIONS.DAT` with `/output_filetype=` qualifier.

4.2 Device Independent Files

From the command line it was clear for what type of device the output had to be generated. It may look funny that the TEXt Processor outputs a so-called Device Independent file with filetype `.DVI_device` that looks as if it is not at all independent.

In fact, most of the contents of the DVI files *is* device independent, and conforms to the file structure described in Donald Knuth's book *TEX The Program*. It is printer type dependent with respect to special commands only useful for specific printers, often using fonts that are available on only one type of device. For instance, to include sixel files for LN03 printers, line art for line printers and PostScript drawings for PostScript printers. The places where these files are to be inserted into the final output file is indicated by the "special" commands written into the DVI files by the TEXt Processor.

The destination however, can also be output for another piece of software. FT1.2 supports VOILA output, which is not a printer at all, but a software program. Also MAIL as destination is not really a printer device.

4.3 Font files

Currently, the Device Converter basically comes in four separate flavours:

1. A DVI to ASCII converter for Line Printer and MAIL and TERMINAL output
2. An DVI to LN03 converter to read LN03 font files to produce LN03 files
3. A PostScript generator to convert the DVI file into a PostScript page description file.

4. A DVI to VOILA output for Bookreader accessible files

Each of these "flavours" in fact is an entirely different callable routine, that is called from the DOCUMENT main program. They are implemented as a set of three shareable images and a command procedure:

```
DOC$ROOT:[TEX.LPR.TOOLS]DVC$LPRSHR.EXE
DOC$ROOT:[TEX.POSTSCRIPT.TOOLS]DVC$PSSHR.EXE
DOC$ROOT:[TEX.LN03.TOOLS]DVC$LN03SHR.EXE
DOC$ROOT:[TEX.VOILA.TOOLS]DVC$DRIVEVOILA.COM
```

Depending on the destination chosen, one of these images is called. Note that for VOILA output a command procedure is invoked. Only the LN03 converter reads the required (and licensed) font files from directory DOC$LN03_FONTS, the PostScript driver only some bitmapped fonts not available inside the PostScript printer from DOC$PS_FONTS , while the lineprinter doesn't read any fonts: they are either permanently available on the printer belt or very obviously missing.

4.3.1 Producing LINEPRINTER files

The Device Converter will use the text written in the DVI_LINE file to produce an ASCII file. As the fonts are fixed to the monospaced fonts available on the printer belts or the built-in laser printer typewriter fonts, the final information in the output file consists of the ASCII characters that make up the words, and additional form feeds for new pages (as well as single carriage returns for bold overstrike). It doesn't read any <FIGURE_FILE>(LINE\filespec\size) as this is already done by the TEXt Processor (it's not different than a normal literal piece of text, except that it is read from another file).

4.3.2 Producing POSTSCRIPT files

The Device Converter will output the text of the .DVI_PS [16] file, but also insert additional PostScript commands. In this respect the DVI representation of the TEXt language is now translated into the Page-Independent Encapsulated PostScript file format of Adobe Systems, Inc.

The PostScript device driver does not have to load any fonts since most of them are resident inside the printer itself. Since the DVI file was processed with the TEXt Processor, and TEX normally knows nothing about PostScript, the font specifications (like \tenpoint) that were translated into the DVI file make no sense to PostScript. Hence, the PostScript device converter uses the file DOC$PS_FONTS:DVC$FONT_CONFIG.DAT to "translate" the TEX font names to the appropriate PostScript fonts, and also indicates which fonts need to be downloaded as they are not part of a normal PostScript printer font set. The end of this font configuration file contains sufficient information for further reading.

The standard PostScript fonts families are different from the LN03 font families and as such also produce a differently looking text page. The device driver will download bitmapped fonts from DOC$PS_FONTS (mostly the *.*PXL fonts).

Any indicated PostScript figure file will be included too. The device converter will read the bounding box comment of an encapsulated PostScript file and places the graphic so that the lower left of the bounding box is placed at the lower left of the space reserved for the graphic[17]. It may take quite a while for the PostScript printer to print files containing graphics.

[16] .DVI_POST in V1.0

[17] The V1.0 behaviour is obtained under V1.2 by editing the PostScript graphic file and placing a carriage return, space or any PostScript statement except a comment at the top of the file. This will disable automatic positioning of the graphic

Some fonts not resident on the PostScript printer can still be used, if they are downloaded to the printer in the document. This is the case with a number of AM* and *.PS fonts available in DOC$PS_FONTS directory.

4.3.3 Producing LN03 files

The Device Converter will read the text from the DVI file and examine which fonts are required to print them. By scanning the DVI file for the first time, a list of required fonts and characters is setup. Then, at the second pass, the final LN03 file is output. The file starts with specific LN03 commands to load the required fonts into the cartridge memory modules of the LN03. This must be done, as VAX DOCUMENT does not use any printer-resident fonts, but always downloads its own required set. The fonts thus defined are temporary, consisting only of those characters and fonts actually used in the document.

As some documents are rather large or contain many different font types, it may not be possible to load all fonts up front in the final LN03 file. The LN03 printers—even with their memory cartridges—may not have sufficient font memory to contain it all. Hence for large documents, the Device Converter will cycle its two passes repeatedly on subsequent sections of the file until the entire document is processed. Hence, the font loading may be repeated during printing. This explains why sometimes the printer temporarily stops printing and seems to enter a state of Deep Thought before continuing with refreshed memory and a different (font) state of memory.

After the font load, the LN03 file contains instructions to activate any particular font just loaded and to use this for printing the text that was also written in the DVI. Whenever figure files for destination LN03 are found, they are opened, checked for the "sixel mode start" escape characters and if found to be correct, included into the LN03 file. Whenever the printer will get to the sixel picture processing, it will temporarily halt, load the sixel picture in internal memory (*not* the font memory cartridges) and when completed, print it. An LN03+ has a much larger internal memory and can produce much more complicated pictures than a simple LN03.

When printing an LN03 file, it will require a fair amount of time to download these fonts before the printer starts printing. Issuing a /COPIES=10 of a particular LN03 file will cause this downloading to happen ten times. Currently there is no way to address the fonts already loaded from a previous print run. A walk to a photocopier may be a much quicker way.

4.3.4 Bookreader

The internal kits also support the VOILA_ONLINE bookreader format. When this destination is chosen (with corresponding doctype ONLINE), a command procedure DOC$VOILA_TOOLS:DVC$DRIVEVOILA.COM is executed. The bookreader format uses its own special (screen) font set and format and produces a .VOILA output. The output is different from ordinary paper-to-screen mapping, since it makes connections between references in the text (in the contents, index, references) and the text referenced to. Simply clicking on the reference will open another window on the screen and display the text. This makes reading a book very easy as all relevant pages can be displayed next to each other. A separate notesfile deals with the ONLINE output: the Online Documentation Systems on MOSAIC::ONLINE_INFORMATION.

4.4 Knitting files together

When a <CONTENTS_FILE> or <INDEX_FILE> tag was requested in the SDML source file and the DOCUMENT command line contains the /CONTENTS and /INDEX qualifiers, the TEXt Processor produced the corresponding `source_CONTENTS.DVI_device` or `source_INDEX.DVI_device` files and left special insertion commands in the main `source.DVI_device` file to instruct the Device Converter to include those contents/index files at the appropriate places in the final output file.

In fact, one can include any DVI file that is suitable for a particular device type at any place in a document by using the <CONTENTS_FILE> or <INDEX_FILE> tags with a single argument. When omitted this argument defaults to the standard contents or index DVI_ device file[18]. Along the same line, the internal tag <MEMO_FILE>(filespec) works similarly.

However, when you would like to include a special document inside another document (e.g. a slide inside a report) this can not be done on SDML level since the Tag Translator and the TEXt Processor will protest about undefined, illegal or conflicting tags and macros or will, at least, format all pages according to a single doctype. When we process both files separately upto the DVI stage, then only the Device Converter has to do its work and this one doesn't care about any formatting. It only wants to knit files together and load the corresponding fonts. This can now be done easily. It is obvious that page numbering will be out of order as well as the index since this information is written by the TEXt Processor for each individual source file and was unaware of our intention to combine the two.

Remember, too that a file included by the <CONTENTS_FILE> or <INDEX_FILE> tags cannot, in turn, include another file by the same mechanism.

4.5 Summary

The following files are read or written by the Device Converter:

1. Read `source.DVI_device` to collect a list of required fonts
2. Read all `DOC$*_FONTS:` font files referred to by the DVI file and use possible translation tables to translate from TEX fonts to PostScript
3. Read all DVI and graphics files indicated by the TEXt Processor through "special" Device Converter commands in the DVI file. This includes figures but also contents and index files.
4. Write final device file, containing all font information, text and graphics

[18] In V1.0 you should only specify file names, V1.1 accepts file types too

Appendix A
Bibliography

The TEXbook
Donald Knuth, Addison & Wesley, 1986
Describes how to program in TEX and make your own designs

The MetaFont book
Donald Knuth, Addison & Wesley, 1986
Describes how fonts for the TEX program can be generated

TEX: The Program
Donald Knuth, Addison & Wesley, 1986
Describes the TEX program and the DVI file layout

The VAX DOCUMENT Doctype Designer's Guide
part of the VAX DOCUMENT documentation set
Describes how to modify designs for VAX DOCUMENT

The Tag Designer's Guide
Describes how to create <TAG> definitions.
Available only for Digital internal use by DEC employees

Index