DVItoVDU System Guide

Version 1.0 for VAX/UNIX
Andrew Trevorrow, July 1986

This document explains how to install or modify DVItoVDU, a TEX page previewer written in Modula-2. Details on how to use the program can be found in the *DVItoVDU User Guide*. It is assumed you are familiar with TEX under VAX/UNIX. Please convey any problems or suggestions to:

Andrew Trevorrow
Computing Services, University of Adelaide
GPO Box 498, Adelaide, South Australia, 5001
ACSnet address: `akt@uacomsci.ua.oz` Phone: (08) 228 5984

Contents

Important files.

We'll assume you've dumped the release tape into some directory and are now looking at its contents. The important files are:

Modula-2 source files

`dvitovdu.mod`	the main module
`screenio.def/mod`	low-level terminal i/o routines (see also `unixio.def/c`)
`sysinterface.def/mod`	command line interface routines
`dvireader.def/mod`	DVI file translation routines and data structures
`pxlreader.def/mod`	PXL file access routines
`vduinterface.def/mod`	generic VDU parameters and routines
`aed512vdu.def/mod`	routines for AED 512
`ansivdu.def/mod`	routines for ANSI compatible VDUs
`regisvdu.def/mod`	routines for ReGIS compatible VDUs
`tek4010vdu.def/mod`	routines for Tektronix 4010/4014 emulating VDUs
`vis500vdu.def/mod`	routines for VISUAL 500
`vis550vdu.def/mod`	routines for VISUAL 550
`vt220vdu.def/mod`	routines for VT220
`vt640vdu.def/mod`	routines for VT100 with Retrographics

Executable files

`dv`	DVItoVDU; built under VAX/UNIX 4.2 BSD
`build`	to build `dv` from scratch
`makefile`	to make `dv` automatically

Test files

`tripvdu.tex`	the source for `tripvdu.dvi`
`tripvdu.dvi`	a torture test for DVItoVDU

Documentation and help files

`dvitovdu.hlp`	the text file read by DVItoVDU's `?` command
`online.hlp`	source file for on-line help
`guidemacs.tex`	macros used by the following guides
`sysguide.tex`	the source for this document
`userguide.tex`	the source for the *DVItoVDU User Guide*

If you don't have a Modula-2 compiler (and don't intend getting one) then you might as well delete all the `.o` files. If you're not even interested in how DVItoVDU works then delete the `.def` and `.mod` files as well.

Installing DVItoVDU.

Since DVItoVDU is too much of a fingerful, the executable file is called `dv`. Before making `dv` publicly available, read the section on command options in the *DVItoVDU User Guide*. If you have a Modula-2 compiler, you may want to make a few changes to `sysinterface.mod` to suit your site:

- Change the default value of `-r` to match the resolution of your printer.
- Change the default values of `-x` and `-y` to define the dimensions of the default paper used by your printer.
- Change the default value of `-f` to tell DVItoVDU where to look for PXL files.
- Change the default value of `-d` to indicate which PXL file to use in case a requested one does not exist.
- Change the default value of `-h` to indicate the location of `dvitovdu.hlp`.

Individual users can of course override any of these defaults. (They may have to if you choose not to use our defaults but don't have a Modula-2 compiler to implement your own.) Any changes you make should be reflected in the user documentation in `userguide.tex` and `online.hlp` (the latter file provides a suitable basis for a `man` entry). System-dependent information in these files is flagged by the string "`SYSDEP`" so you can quickly locate where changes need to be made.

Overview.

As its name would imply, a Modula-2 program is typically built up from a number of separately compiled modules. Each module can import data and procedures from other modules. An imported module is further decomposed into a definition module and an implementation module. The definition part contains declarations for all exported objects and serves as the interface to client modules. The details of how exported objects are actually realized are contained in the implementation part (i.e., hidden from clients). Rapid system regeneration is possible because client modules only depend on the definition part; the implementation part may be modified (e.g., optimized) without the need to recompile client modules. The module map on page 5 shows the importation dependencies of all the modules making up DVItoVDU.

Not only do modules allow a large program to be broken up into manageable chunks, they also provide a sensible basis for describing a program:

`DVItoVDU`

The main module is primarily concerned with the user interface. It handles all the interactive command processing and contains the high-level logic used to update the dialogue and window regions. Data and procedures are imported from the following modules to help the main module carry out its many tasks.

`SysInterface`

This module provides the interface to the UNIX shell. The command line used to invoke DVItoVDU is parsed and the DVI file name extracted. All the options are also initialized, either to explicit values given in the command line or to site-dependent default values.

`VDUInterface`

DVItoVDU can work efficiently on different types of VDUs by letting Modula-2 procedure variables act as generic VDU routines. These routines, along with the generic VDU parameters, are defined in `VDUInterface`.

The generic VDU routines are:

`StartText`	switch to "text mode"
`ClearTextLine`	erase given line
`MoveToTextLine`	move cursor to start of given line
`ClearScreen`	erase the entire screen
`StartGraphics`	switch to "graphics mode"
`LoadFont`	simulate the given font for future `ShowChar` calls
`ShowChar`	display a Terse mode character at given screen location
`ShowRectangle`	display a given rectangular region of screen pixels
`ResetVDU`	reset VDU to its normal state

Most of these are quite trivial to implement for a specific VDU. The main module looks after all the tricky graphic operations such as the clipping of characters and rules outside the current window region, and the way in which visible paper pixels are scaled to screen pixels.

From an efficiency point of view, the two most critical routines are `ShowChar` and `ShowRectangle`. `ShowChar` is used by the main module to display a character in Terse mode. The only information given is the TEX character (currently restricted to `\char0..\char127`) and its screen position. Since characters are displayed one font at a time, some VDUs can use scaling information sent by the most recent `LoadFont` routine to select an appropriate hardware "font".

`ShowRectangle` is used for all other window graphics. It is used to draw the paper edges, to draw all rules (regardless of display mode), to draw all glyph outlines in a Box display, and to draw the horizontal lines making up all glyphs in a Full display. The majority of rectangles are in fact horizontal or vertical lines just one screen pixel thick.

The generic VDU parameters are:

`DVIstatusl`	DVI status line, usually 1
`windowstatusl`	window status line, usually 2
`messagel`	message line, usually 3
`commandl`	command line, usually 4
`bottoml`	bottom text line in screen
`windowh`	window region's top left h coordinate
`windowv`	window region's top left v coordinate
`windowwd`	unscaled window width
`windowht`	unscaled window height

These "parameters" are actually integer variables. The main module treats them as constants.

There are two screen coordinate systems used by DVItoVDU:

(1) When updating the screen in text mode (i.e., when updating the dialogue region or during a `?` or `S` command), DVItoVDU assumes text lines start at 1 and increase downwards. The bottom text line on the screen is given by the parameter `bottoml`.

(2) When updating the screen in graphics mode, DVItoVDU assumes the top left screen pixel is positioned at (0,0). Horizontal coordinates increase to the right and vertical coordinates increase down the screen. The top left pixel in the window region is at (`windowh`,`windowv`). Specific VDU modules may have to do a translation to the actual coordinate scheme used by the VDU. The size of the window region in screen pixels is given by `windowwd` and `windowht`.

`AED512VDU`, `ANSIVDU`, `REGISVDU`, etc.

Each specific VDU module exports an initialization routine that will assign appropriate procedures to the generic VDU routines and specific integers to the generic VDU parameters. `VDUInterface` imports all these initialization routines and uses the `-v` value set in `SysInterface` to execute one of them.

`DVIReader`

This module exports the routines and data structures needed to move about randomly in a DVI file and interpret selected pages. Although the main module is currently the only client, it is anticipated that `DVIReader` could just as well form the basis of a more conventional DVI translator such as a non-interactive device driver.

Font, character and rule information is stored in dynamically allocated lists to avoid imposing any limit on their numbers. The length of the font list is determined soon after opening the DVI file by reading all the font definitions in the postamble. Each font node is a record made up of many fields; one of these fields is the head of a character list. The nodes in each character list will store the positions and TEX codes of all characters on a page. Besides the font list, there is also a rule list. The nodes in the rule list will store the positions and dimensions of all rules on a page. (Character and rule positions are stored as pairs of horizontal and vertical paper pixel coordinates. The manner in which `DVIReader` calculates such positions is based firmly on Donald Knuth's `DVItype`.)

Just before interpreting a selected DVI page, the rule list and all the character lists are deallocated if necessary. During interpretation, `DVIReader` adds a new rule or character node to the *tail* of an appropriate list. When the main module processes such lists, rules and characters will be displayed in somewhat the same sequence as seen in the DVI page; i.e., top-to-bottom and left-to-right. (Since there is a separate rule list, as well as a character list for each font, the precise sequence is not remembered.) After interpretation, the nodes in the font list are sorted so that fonts with the least number of characters (> 0) will be processed first.

The various lists contain most of the information needed to display the page; they are traversed by the main module whenever the window region is updated. If the page isn't empty then `DVIReader` will also determine the edges of the page rectangle. This is the smallest rectangle containing all black pixels in glyphs and rules, as well as all character reference points (needed for Terse displays). The main module uses the page rectangle to decide if the page is off the paper, and to restrict window movement.

`PXLReader`

DVItoVDU gets all its character information from standard PXL files. `PXLReader` exports routines for moving about in such files and grabbing various bytes and words. A PXL file contains the crucial TFM widths needed by `DVIReader` to correctly position characters when interpreting a DVI page. These widths, along with other details about each glyph, are kept in a PXL file's font directory. A directory is loaded just once for each font (the very first time the font is seen) and DVItoVDU will display '`Loading font data from ...`' in the message line.

A PXL file also contains glyph shape information (in the form of bitmaps) for all characters in a TEX font. DVItoVDU uses these bitmaps during a Full display to draw characters a font at a time. Each time a PXL file is opened, the message '`Drawing characters from ...`' will appear.

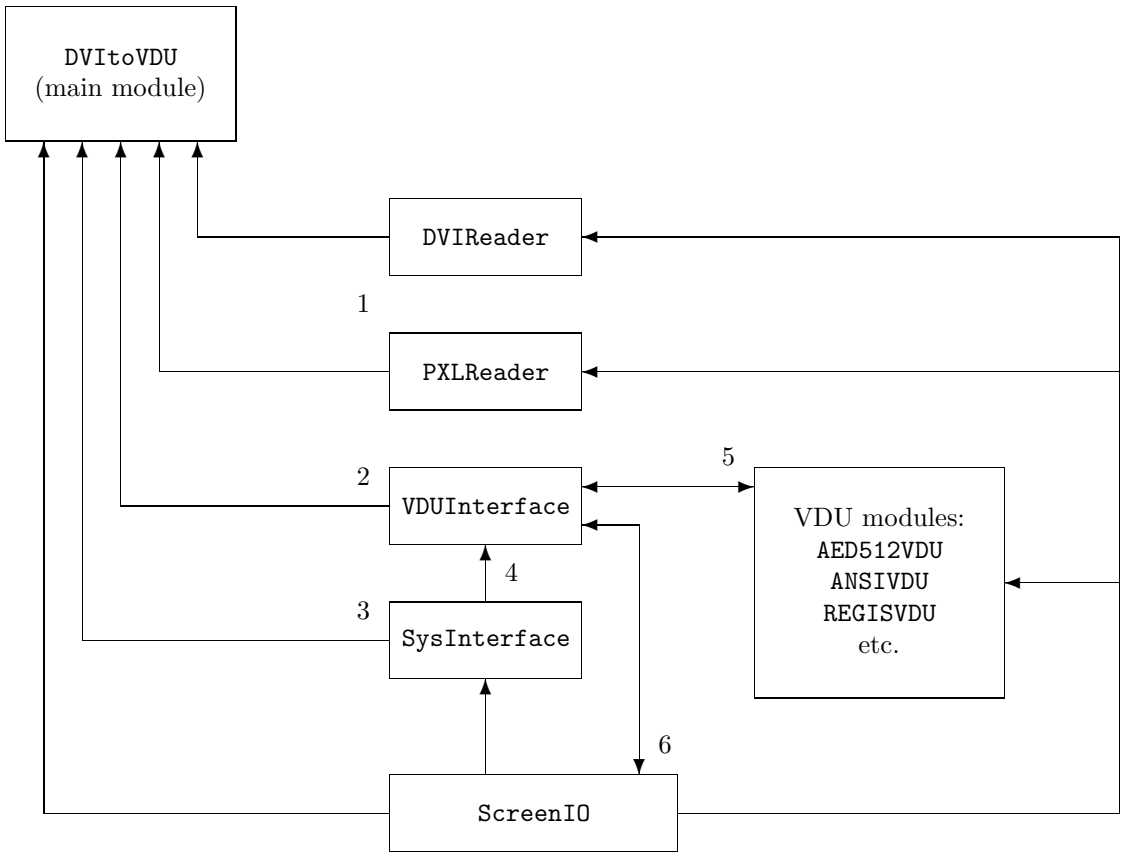
Because `DVIReader` creates a separate character list for each font used on a page, there never needs to be more than one PXL file open at any given time.

`ScreenIO`

All low-level terminal i/o is handled by the routines defined in this module. It uses some C routines written by Alex Dickinson (see `unixio.c`).

Module map.

An arrow from module `A` to module `B` indicates the latter imports data and/or procedures from the former. Module `B` is said to be a *client* of module `A`.



Points of interest:

1 `DVIReader` does not depend on `PXLReader`. It does know that PXL files contain crucial typesetting data, but leaves it up to the main module to specify how and where to get such information.

2 `VDUInterface` exports the generic VDU routines and parameters for use in the main module.

3 `SysInterface` extracts the DVI file name and options from the `dv` command line.

4 `VDUInterface` needs the `-v` value to select an appropriate VDU initialization routine.

5 Specific VDU modules import the generic VDU routines and parameters from `VDUInterface`. In return, each VDU module exports an initialization routine.

6 `ScreenIO` needs to use some `VDUInterface` routines to be able to leave the VDU screen in a decent state after abnormal termination.

Modifying DVItoVDU.

DVItoVDU was originally developed under VAX/VMS using the Modula-2 system from Hamburg University. The UNIX version of DVItoVDU was developed using the Modula-2 compiler from DEC WRL and can run under ULTRIX or UNIX 4.2 BSD. The main module and all the supporting modules are written in Modula-2, except for the C routines in `unixio.c`.

If you wish to modify DVItoVDU you'll obviously need Modula-2. Unfortunately, the DEC compiler is not in the public domain, so I can't send you a copy. Conversion to another Modula-2 system should, however, be quite easy. All the source files avoid non-standard compiler features, and system-dependent code is flagged by the string "`SYSDEP`". Terminal i/o is isolated in the `ScreenIO` module and most file operations are restricted to `DVIReader` and `PXLReader`. (The VMS-to-UNIX conversion took me about two weeks. A working version was up in less than a week; it then took another week to fix some terminal i/o problems and produce a new *User Guide*. I was a bit surprised at just how easy the conversion was.)

What about conversion to another language? An obvious candidate is Pascal. The similarities in syntax would allow any decent text editor to carry out much of the conversion automatically. The major difficulty in translation to standard Pascal would be the lack of procedure variables; it might be easier to implement just one type of VDU before attempting anything more sophisticated.

IMPLEMENTING A NEW VDU.

Assuming you have Modula-2, the most likely change you'll want to make is to add a new type of VDU to the program's capabilities. I've taken advantage of Modula-2's separate compilation facilities so that such a task can be undertaken without having to make any changes to the main module.

The steps required to implement a new VDU (which we'll call brand XXX) are:

- Read `vduinterface.def` to see if the task is even possible. DVItoVDU requires XXX to be able to carry out a number of primitive graphic operations such as clearing the screen, addressing individual pixels, etc. As little as possible is assumed about the capabilities of a VDU. In particular, the availability of a graphic input device is ignored. DVItoVDU should be able to work on any terminal that can:
  - — Mix text and graphics on the screen (some VDUs make no distinction).
  - — Erase all of the screen, or individual text lines.
  - — Move the cursor to any given screen pixel.
  - — Display a rectangular region of screen pixels (possibly just one).
- Look at some of the `*vdu.def` files and use them to help you create `xxxvdu.def`. Our Modula-2 system does not require `.def` files to be compiled.
- Look at some of the `*vdu.mod` files and use them to help you create `xxxvdu.mod`. Type '`mod -c xxxvdu.mod`' to create `xxxvdu.o`.
- Edit `vduinterface.mod` and add appropriate code in the recommended locations (search for occurrences of "`XXX`"). Type '`mod -c vduinterface.mod`' to create a new `.o` file.
- Relink `dv` by typing '`mod -o dv *.o`'.
- Test the new version of `dv` (see the next section).

It should be pointed out that DVItoVDU is currently targeted towards relatively primitive terminals. Significant changes to both the display logic and page data structures would probably be needed to take full advantage of the latest graphic workstations. For instance, a large amount of bit-mapped memory could store an entire rasterized page and allow much more sophisticated updating of the window region. Pan and zoom operations would also be much easier using a mouse or thumbwheel cursor controls.

Testing DVItoVDU.

The file `tripvdu.dvi` was created to test DVItoVDU using much the same philosophy as Donald Knuth's torture test for TEX. Most pages exercise rarely-used parts of the program:

page	contents
1	empty page (but with `\special` stuff that will be ignored)
2	one black pixel at (0,0)
3	entirely blackened A4 sheet of paper
4	like page 3 but one pixel wider
5	like page 3 but one pixel longer
6	has a rulelist with one full ruletable node (300 rules)
7	like page 6 but has one more rule
8	has a charlist with one full chartable node (3000 characters)
9	like page 8 but has one more character
10	multiple TEX page counters
11	negative TEX page counter
12	characters from many fonts
13	characters from many fonts, some of which have no PXL file
14	paragraph using most characters from standard roman font
15	page in bottom half of A4 paper
16	page entirely above and to the left of A4 paper
17	page entirely below and to the right of A4 paper
18	page extending beyond all edges of A4 paper

Type '`dv tripvdu`' and go through all the pages to verify that the program is working correctly. You may need to specify a `-v` value to tell `dv` what type of terminal you're using. See the *DVItoVDU User Guide* for details on what `-v` values are available and the special requirements of some VDUs.

If a catastrophe occurs and DVItoVDU actually crashes, please report the problem back to me. If the reason for the crash is not obvious, and you have a Modula-2 compiler, you might try to rebuild `dv` after editing the `.mod` files that contain debugging code. Change all comments of the form `(* DEBUG ... GUBED *)` to `(* DEBUG *) ... (* GUBED *)`. Only a few modules have such code; use '`grep DEBUG *.mod`' to list them. The extra code might provide additional clues as to what is going wrong.

Revision history.

Version 1.0 of DVItoVDU for VAX/UNIX is based on version 1.5 for VAX/VMS (TUGboat vol. 7 no. 1 has an article describing the VMS version). The UNIX version is virtually identical as far as the user is concerned. The command option syntax is a little different of course, and there are a couple of new VDU types, but the interactive commands are exactly the same.

A few improvements have been made to the UNIX version:

- Whenever a `\special` command is ignored, the first few bytes are displayed in case some users need to see them.
- While updating the window region, the main module uses `BusyRead` from `ScreenIO` to check for user input after drawing each visible rule or character. (The VMS version checked after every 8th rule or character; the overheads for `BusyRead` seem to be much less under UNIX.)
- `ScreenIO` required significant changes to be able to handle some of the more useful UNIX interrupts: `^C` can be used to return quickly to the '`Command:`' level, and `^Z` can be used to suspend `dv`. (Thanks to Alex Dickinson who wrote the necessary `unixio.c` routines.) These interrupts are only detected during `Read`, `ReadString` and `BusyRead`.

⋆ ⋆ ⋆