Originally, the source for the package was held in a number of text files, subsets of which were compiled by the manual preparation of compiler directives, naming each file to be included in a compilation. There were a number of drawbacks to this approach.

First, duplicate copies of source files were generated. A programmer working on one portion of a package level would typically copy the entire level to a private area of the file system and modify it in isolation; sharing only the global constants and types. Having completed the enhancements an attempt would be made to re-integrate the source. This would only be successful if no parallel and conflicting modification was applied to that level. A good example of how this approach can disrupt progress is that a special version of one level using Watkin's hidden line algorithm co-existed with the regular version for more than a year because the two could not be merged. The severe penalty this imposes is that common modifications must be performed on each version. The likelihood of these modifications being performed inconsistently or incorrectly increases dramatically as the number and age of the versions increases.

Second, the package was not sufficiently flexible in the way new devices could be added or devices, unnecessary to an application, excluded. Switching between devices is controlled at run-time throughout the device-sensitive source by means of a variable of a global type DEVICETYPE and the use of **case** statements which select appropriate actions as illustrated in Fig. 1.

The difficulty with this approach is that the set of devices supported by a segment of source text must always be the full set of devices for which its operation might be used, or the above problem of having multiple versions will still exist. If that source is compiled, then the generated system will contain code for all such devices, whether they are needed or not. In general this causes an instance of the system to be larger than necessary. The size of the generated system is an issue because it affects response time in mainframe timesharing and may even determine the feasibility of fitting into limited address space on mini- and microcomputers.

```
procedure Showl1;
{
Display the picture plotted so far.
The existing picture is NOT destroyed.
Programmed by: JCBeatty
Last Modified: Sat 2 February 1980 10:50pm
Copyright © 1980, University of Waterloo
}

begin
case device of
  virtual:      writeln( virtual_file, 's' );
  tektronix:    Bflushl0;
  volker_craig: Bflushl0;
  tek_4027:     Bflushl0;
  printer:      Dumpl1;
  zl_buffer:    Dumpl1;
  pdi:          Bflushl0;
  hp_2648:      Bflushl0;
end;
end {Showl1};
```

Fig. 1 Switching between devices

Adapting Thoth's Solutions

Experience with similar problems in managing the source text of a portable operating system, Thoth [3], suggested that techniques which had proven successful there might be applied to the graphics package.

Thoth's source is structured using its own hierarchical file system to form a tree of the nested abstractions which comprise the system. The source realizing a given abstraction is organized such that each module is held in a separate file whose name is the same as that of the module it contains. *Versions* of modules for specific processors, peripherals and so forth are then introduced in files which are subordinate to the base version of the module and named by the processor or other attribute which characterizes the version. Versions of versions are treated quite naturally in the same way. The compilation and assembly (assembler realization for a given processor is merely a version class) is then driven by an Inclusion Builder which "traverses" a tree within the file system and, by means of preset directives and dialogue with the user, determines which files should be processed. A more complete discussion appears in [4].

Unfortunately, it is not possible to apply the Thoth technique to the graphics package without modification and compromise. The reasons for this are twofold.

First, the GCOS file system is less well suited to this approach than is Thoth's. GCOS distinguishes catalogs (directories) from files and only catalogs may have substructure. Under Thoth there is a single node type in the hierarchy, allowing version files to be subordinates of other files. Moreover, GCOS imposes a number of limitations which restrict the extent to which the file system can be used to express the structure of its contents. Names of files and catalogs may not exceed 12 characters and the maximum catalog depth is 10 levels.

Second, Thoth's implementation language is a BCPL derivative called Zed. Zed's design expressly supported the construction of sophisticated portable systems programs, but Pascal's was intended primarily to foster disciplined program construction by novices and to allow for efficient implementation. The impact of Pascal is discussed below.

Working in Pascal

Rather than dwell on the well-known problems of programming-in-the-small in Pascal[6], we concentrate on the issues of programming-in-the-large[5]. The essence of the proposed approach to source management is that the source text be fragmented and structured in such a way that a family of systems is portrayed to programmers.

The *abstract* syntax of Pascal helps in this regard, for it clearly delineates the components which fill specific roles in the realization of a computational abstraction: constants, types, variables and procedures. This may be exploited.

Pascal provides nested declarations with an inherited scope rule, which one might think could be used for information hiding. However, the nature of the graphics package calls for each level to make some of its procedure identifiers available to the next higher level in the package. While this could be done by nesting the higher levels of the package within the lower ones and the application code itself at the deepest level, this would call for a return to monolithic compilation. With few exceptions, procedure declarations are not nested and lie at a single level, each with a unique name.

The *concrete* syntax is the more troublesome. The language calls for definition before use of all identifiers in order to facilitate one-pass compilation. Procedure declarations must therefore be ordered appropriately and *forward* declarations introduced when there is mutual recursion, as there is in the graphics package.