# The Project Automated Librarian

by J. M. Prager

***The Project Automated Librarian (PAL) is a tool that has been created to manage the logistical problems inherent in a medium-sized software development project. The main goals of PAL are to eliminate the problems of simultaneous updates to software modules, while allowing programmers access to the latest possible versions of the software. PAL also seeks to prevent the software from getting into an inconsistent state that could prevent users from proceeding with software development because of someone else's errors. PAL is a general-purpose tool, in the sense that it does not care what language or languages the system is being written in. It makes backups, keeps version information, and maintains documentation of changes.***

When two or more people work on development of software for a single system, certain problems frequently arise. One of these occurs when several people wish to update a given component of the system. They may all make private copies, make the changes, and then attempt to install the modified code. Unless steps are taken to avoid it, the last person to install the changes will "win," and the changes made by the others will be lost. A partial remedy is to have the editor being used maintain a list of the changes to the file, and have these "update files" applied to the original file. This remedy only works if the changes were made to disjoint portions of code.

The other major problem is that of currency. Even if all programmers are working on different areas of code, it is highly desirable that when one set of changes has been tested and approved, it be immediately installed so that everyone associated with the project can have the latest possible version. Under VM (Virtual Machine Facility/System Product), the basic minidisk support does not provide safe write access to common files for more than one person at a time. Instead, it is necessary to build a higher level of support to achieve this goal.

A possible solution to these problems, but one that is only really practical in a two-person project, is to have "private" arrangements about who works on which components of the system next, and who has the latest versions of the various components. To be at all workable, this requires constant attention to such "bookkeeping" details.

The use of a human librarian can alleviate these problems, but it is a burdensome and boring task for the individual, especially if backups are needed and change documentation must be made. It was to solve these problems that the Project Automated Librarian (PAL) was developed. A human librarian is still needed (to install and maintain PAL), but this task is small compared with that performed by PAL itself.

PAL was created to manage the software development of the POLITE[1] project at the IBM Cambridge Scientific Center. POLITE is a real-time editor-formatter which has about 44 000 lines of code. This code is being worked on by up to six people simultaneously. It is mostly the case that different programmers have different parts of the code to work on, but some sections of code need to be changed by two or more people. Because of a great concern that POLITE provide good human factors, it was seen to be critical that the latest version of the system be quickly made available for testing (and use).

**PAL allows people to gain access to the latest possible version of the system.**

PAL is written in Pascal[2] and EXEC2,[3] and runs under the Conversational Monitor System of VM (VM/CMS).[4] It is used to solve as far as possible the problems listed above.

PAL is like the Source Code Control System (SCCS)[5] in some respects. They both, for example, allow only one user at a time to modify a software component. SCCS, however, is much more oriented to making past releases accessible—in fact it attempts to record every version of every component that ever existed. It does this by storing "deltas"—representations of changes between successive versions of a component. The most recent version of a software component must be generated dynamically by applying all relevant deltas to the original version of the component. PAL, on the other hand, keeps entire copies of previous versions; the number of such backups kept is settable by the installation. SCCS does not perform the compilations, assemblies, etc., which PAL may perform once a set of changes has been resubmitted to the system.

As is described in more detail later, PAL solves the problem of simultaneous updates by "checking out" code to the first person who asks for it, as in a lending library; nobody else can modify the code until it is "checked in" again. PAL maintains a project disk where all files reside with read access given to everybody, but one can update only by going through both the check-out and check-in processes.

PAL allows people to gain access to the latest possible version of the system, because it acts as a central depository for all working components of the system. Users are discouraged from keeping "personal" copies of any part of the system, except those components under active development. Users will typically only check out "source" files; derived files such as TXTLIBs and MODULEs are generated automatically by PAL when the source files are checked back in. Thus, all users access the most recent tested versions of the components of the system, installed in the project library.

Beyond the solution of these problems, benefits of using PAL include disk space savings (due to limiting the number of identical copies of a file), automatic change history and documentation, and automatic backup and version generation.

PAL requires all users to have their identification on a single VM system. It does not cater to remote users, or to two or more parallel development efforts at different sites.

PAL was not intended to be highly sophisticated and complex. For example, it contains no security features to prevent unauthorized changes being made—it relies on trust. Furthermore, it assumes a style of modular programming to work well. However, since its *raison d'être* is to eliminate the need for a human librarian, it was designed primarily to be simple to install, modify, and use. Reaction from sites that have installed PAL confirms that these goals have been achieved.

This paper consists of four major sections. First, we give a description of PAL. Then we describe how PAL appears while it is being used. In the third section, we describe tools for building and modifying PAL itself. Finally, we discuss current implementations, future developments, and potential problems with PAL.

## Description of PAL

**General overview.** PAL runs under VM/CMS as a disconnected virtual machine. It maintains a project

disk that contains the most up-to-date version of all project files, including any executable modules or other "derived" files. Each module is regenerated

**PAL maintains a catalog of all files in its domain.**

when successful submissions of new versions of source code occur. Logically, PAL appears to the user as a librarian; the user may check out many files that are not already out and may check them back in when done. Physically, checking in and checking out are accomplished by EXECs which send files between the user's machine and PAL's machine.

PAL exerts more control, however, than simply the checking in and out of files. Although it is expected that the user will verify the changes he made to code before resubmitting the file (at least as far as syntactic correctness is concerned), this cannot be guaranteed. Thus PAL performs "appropriate" syntactic checking (e.g., zero return code from compiler) before check-ins are accepted. PAL also automatically generates any files that are derived from the files checked back in. This procedure is explained in some detail below, using Pascal compilations as an example.

The problem of getting the right files regenerated when a given file has changed, known as the problem of "consistent compilation,"[6] is the responsibility of the project librarian who sets PAL up. This problem is discussed in the third section.

PAL maintains a catalog, or inventory, of all files in its domain. Some files may be source files, others may be generated (by PAL) from these source files, others may be EXECs, and so on. Most circulate to all users, although some may have their circulation denied (typically, one will not want to allow TEXT files and MODULES to be checked out). Each file is marked accordingly in the catalog. A human librarian installs all files initially, using an aid known as LIBTOOL, which is described later. During this procedure, the librarian informs PAL what processes are to be run whenever a file is checked in. Thus, whenever a source file is changed and checked in, PAL will be able to regenerate the module, as well as any other derived files.

POLITE, the project that provoked the development of PAL, is written in Pascal. PAL will typically circulate to users (members of the POLITE project team) either COPY files or PASCAL source files. In the generation of a POLITE MODULE during check-in, there are four stages of processing, any of which may be required from the original source files:

1. MACLIBs are built up from COPY files.
2. TEXT decks are generated by compiling PASCAL files with MACLIBs.
3. TXTLIBs are generated from TEXT decks.
4. A MODULE is made from TEXT decks and TXTLIBs.

The notion that up to four stages of processing may be required on checking in any file has been generalized in PAL to remove language dependencies. Thus, the fact that POLITE is written in Pascal is irrelevant to the general operation of PAL, which would work just as well compiling any language, as long as the commands to do so are put into the catalog initially. In fact, as is described in a succeeding section, PAL can be used for maintaining system documentation instead of (or as well as) maintaining software.

When a file is checked in, PAL generates all those files that are derived from it. If there are any errors during the process (such as compilation errors), the entire set of changes is rejected and sent back to the originator, along with an appropriate message. If there are no problems, the newly checked-in files are installed and the catalog is updated. Messages are sent to all users linked to the project disk informing them of the update.

**PAL in more detail.** The architecture of PAL is shown schematically in Figure 1. PAL runs in a virtual machine which is normally disconnected. The machine maintains two minidisks, the B-disk which contains the library and all related execs, and the A-disk which is used for temporary files created during the check-in process. PAL runs an exec that puts it in a wait state until input arrives in its virtual

card reader. When this happens, PAL "wakes up" and begins processing the input.

A typical user is linked to PAL's B-disk with read access. To check out a file, he runs the main PAL exec, which resides on that disk. After all the files he wants have been named (which PAL checks to see are both in the library and are permitted to be lent out), PAL is asked to process these requests. This causes a CONTROL file to be generated and transmitted from the user's virtual machine to PAL's virtual machine. On receipt of this file in its reader, the PAL machine wakes up and reads in the file. The file contains a list of tasks to be performed. PAL sees that the user has requested some files to be checked out, so sends him the message

"BEGINNING CHECKOUT PROCESSING"

followed by

"I AM CHECKING OUT filename filetype FOR YOU"

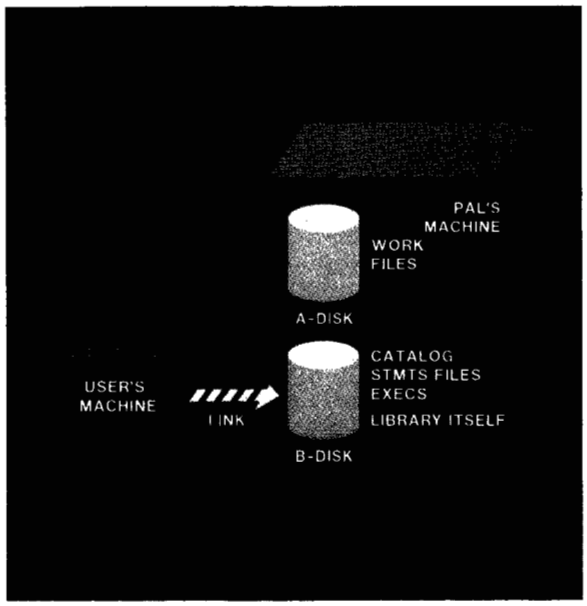
for every file to be checked out. When PAL finishes, it goes back to sleep and waits for more input in its reader. PAL uses special class descriptions for CONTROL and library files, in order to distinguish them from each other and from any other random files that might get sent to its machine. PAL only (initially) pays attention to CONTROL files, which it reads in and analyzes. If a CONTROL file specifies a check-in function, PAL looks at the other files in its reader.

The user will read the files that PAL sends him onto his own private disk(s), and edit and test the changes there. Clearly the user will not usually have a copy of the complete system under development. Since PAL maintains the latest versions of all files, both source and derived, the user need merely be linked to PAL's B-disk in order to have read access to any part of the system he has not checked out.

A user tells PAL that he wishes to check in files in much the same way that he checks them out. For each file to be checked in, PAL asks if any changes have been made. If the user answers in the affirmative, PAL prompts the user to supply some documentation of the change. This documentation, which is either typed in on the spot or copied from a previously prepared file specified by the user, is put in to a DESCRIPT (description) file.

When this part of the process has been completed, PAL constructs a CONTROL file listing all the files to be checked in, along with the names of the

**Figure 1 PAL-user relationship under VM**




DESCRIPT files. The CONTROL file and all the others are then sent from the user to PAL, which then wakes up, scans its reader for a CONTROL file, and reads it in. It issues the message

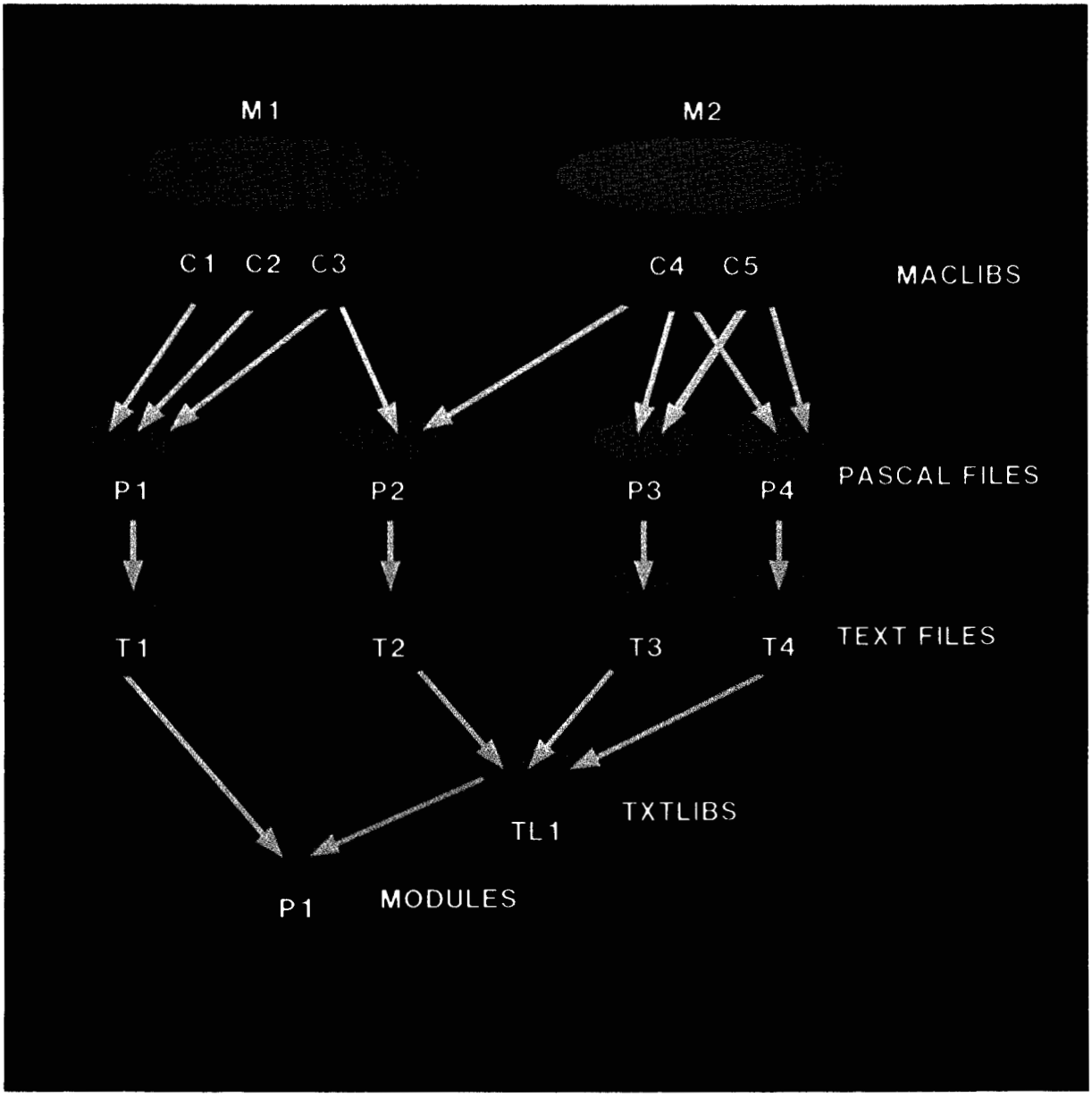
"BEGINNING CHECKIN PROCESSING"

and then reads in all files specified in the CONTROL file.

As mentioned earlier, PAL is configured to expect up to four sequential stages of processing to occur when a file is checked in, although this number can easily be changed. For Pascal, these stages correspond to MACLIB, TEXT, TXTLIB, and MODULE generation, and they must happen in that order, although there may be several steps during each stage. For example, a given COPY file may be INCLUDEd in several Pascal programs, so that when this file is checked in, several compilations take place. As far as PAL is concerned, there is no particular semantics associated with these stages, which we will call STAGE1, STAGE2, STAGE3, and STAGE4.

PAL maintains the library catalog and four STMTS (statement) files, corresponding to STAGE1-4. For each file in the catalog, there is a (possibly empty)

**Figure 2 Example of software structure**




list of entries for each of the four stages. These entries are labels and refer to lines in the STMTS files that will be executed when the files are checked in (with changes). Thus the STAGE1 STMTS file contains all of the calls to generate MACLIBs, the STAGE2 STMTS file contains all of the Pascal compilation calls, and so on. Rather than having the operations to be performed listed with each file in

**Figure 3 Contents of four STMTS files**

```
                    contents of STAGE1 STMTS file

M1:  MACLIB GEN M1 C1 C2 C3
M2:  MACLIB GEN M2 C4 C5



                    contents of STAGE2 STMTS file

P1:   EXEC PASCALVS P1 ( LIB (M1) NOPRINT)
P2:   EXEC PASCALVS P2 ( LIB (M1 M2) NOPRINT)
P3:   EXEC PASCALVS P3 ( LIB (M2) NOPRINT)
P4:   EXEC PASCALVS P4 ( LIB (M2) NOPRINT)



                    contents of STAGE3 STMTS file

TL1:  TXTLIB GEN TL1 P2 P3 P4



                    contents of STAGE4 STMTS file

P1:   EXEC PASCMOD P1 TL1
```

the catalog, the indirection is preferred because many files may share some processing (e.g., they may all generate the same MODULE).

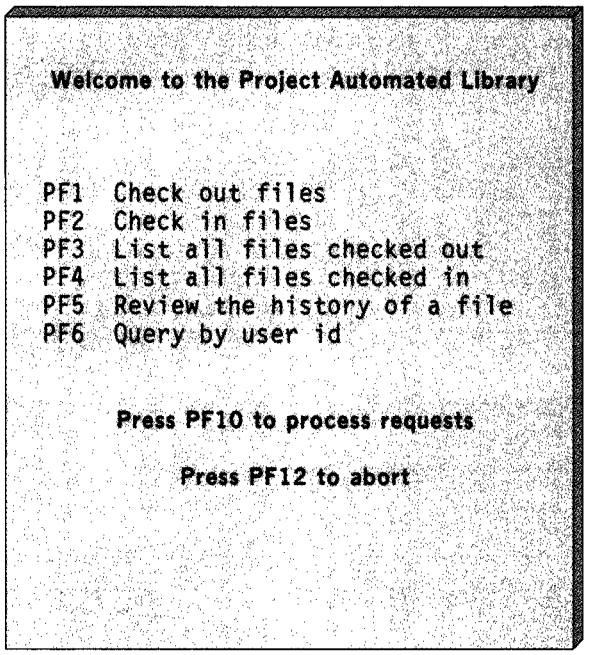
When checking in a file with changes, PAL looks it up in the catalog to determine its associated labels. PAL then looks in each of the four STMTS files in turn for the corresponding EXEC statements and performs the requisite processing.

For example, consider the simple software structure depicted in Figure 2. C1 to C3 are COPY files in MACLIB M1, and C4 and C5 are COPY files in M2. P1 is a PASCAL file which includes C1 to C3, and P2 includes C3 and C4, whereas P3 and P4 both include C4 and C5. T1 to T4 are the TEXT decks formed by compiling P1 to P4. TL1 is a TXTLIB formed from T2 to T4; T1 and TL1 are linked to form the module P1.

The STAGE1 STMTS file contains the CMS statements to generate M1 and M2 from the COPY files. The STAGE2 STMTS file contains the statements to compile P1 to P4, with reference to the appropriate MACLIBs. The STAGE3 STMTS file contains the statement to generate TL1, and STAGE4 STMTS contains the one to generate the P1 MODULE (see Figure 3). How these statements are keyed to the check-ins of the respective source files is described later in the section on the maintenance of PAL.

The necessity for identifying and distinguishing among different processing stages becomes appar-

**Figure 4 Top-level PAL screen**




ent when several files are checked in at once. Consider a TXTLIB that is generated from two compiled Pascal files, both of which have just been checked in (e.g., TL1 from P2 and P3). The STMTS files will specify for both Pascal files a compilation and TXTLIB generation. Since the TXTLIB generation is the same in both cases, that operation should occur after the two compilations, so that it need only be done once. As another example, if a collection of PASCAL files and COPY files have been checked in, the MACLIBs should be generated from the COPY files before the PASCAL files are compiled.

These considerations (and analogous ones for non-Pascal systems) force PAL to adopt a certain strategy for processing check-ins. For the set of files checked in for any single transaction, PAL looks up all the STAGE1 statements (ignoring duplications), and then processes them. The same then happens for STAGE2, STAGE3, and STAGE4 in turn. This strategy both minimizes the total processing required and guarantees that each file is processed with the most up-to-date versions of associated files at all times.

All checked-in files reside at first on PAL's A-disk, the work disk. All dependent files that are generated by the check-ins are also stored on the A-disk.

**PAL presents the user with a screen showing a menu of possible actions.**

When all processing that has been triggered by the check-in has finished successfully, PAL issues the message

"I HAVE CHECKED IN filename filetype WITH CHANGES"

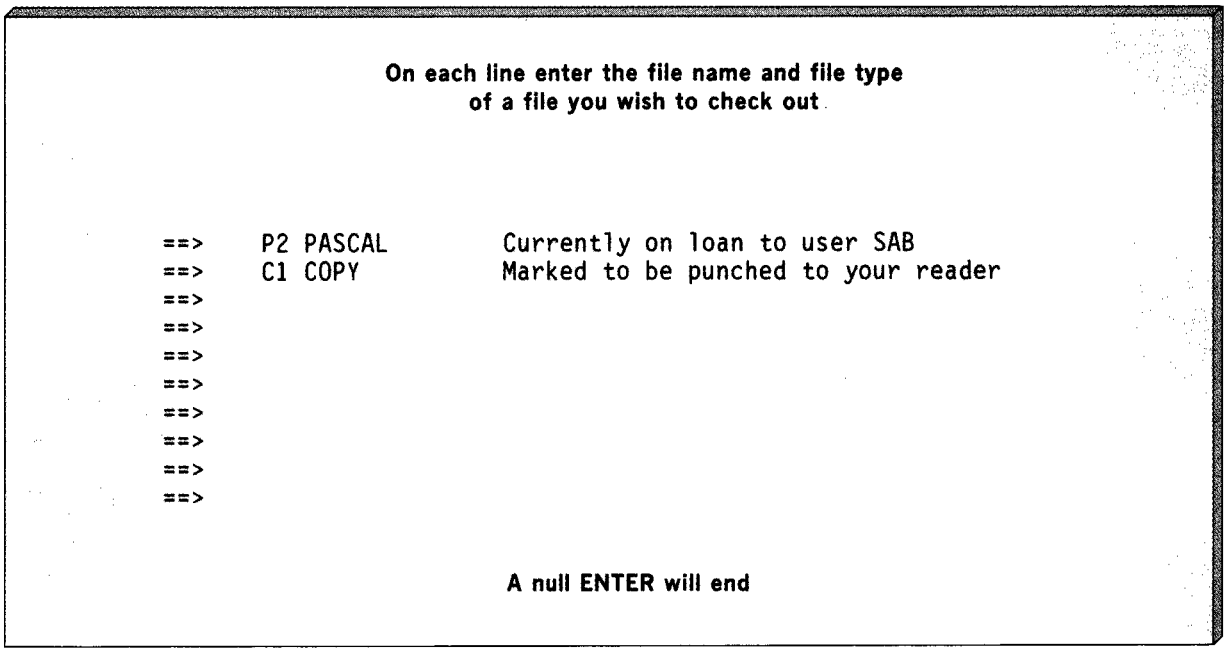
for each such file, copies all checked-in and generated files from the A-disk to the B-disk (the library disk), erases them from the A-disk, and goes back to sleep. If a file is checked in with no changes, no processing is required. If one of the processing stages fails, due to, say, compilation errors, the entire job is terminated, the files are sent back to the user, and a message is given to the user explaining what has happened.

### How PAL is used

PAL presents the user with a screen such as that depicted in Figure 4. The user may check files in or out, list those files checked in or out, review the history (documentation) of a file, or list all files most recently checked in or out by any user. There are sorting and scrolling facilities for all of the listing functions. We discuss here only the checking in and out of files. The system returns to this menu after any of its subsystems (accessed by PF1-6) have terminated.

If the user indicates that he wishes to check files out, PAL presents him with a screen such as that in Figure 5. The user may specify any number of files, and, assuming that the files are in the library and that they may circulate, PAL internally queues up a series of actions to check out the indicated files. In

**Figure 5 Check-out menu**




this instance, the user has requested two files (names in italics) and the system's responses are shown. The first request has been rejected because the file is already on loan. The second request has been accepted. When the user terminates this screen and issues the PROCESS command at the top-level screen (Figure 4), the transactions are actually executed, and the files sent to him.

Alternatively, the user may elect to check files in (see Figure 6). PAL allows him to check in only those files which he himself has checked out. PAL asks for the filename(s) and whether any changes have been made. Even if no changes have been made, this check-in procedure must still be followed (although no files are actually transmitted), so that PAL can be informed that the file is available to others (i.e., put back into circulation). Figure 6 shows the state of the screen after the user has entered the name of a file (C1 COPY) to be checked in, the system has asked if the file was modified, and the user has replied "yes." The user has already checked in the files listed at the top of the screen during this transaction.

If changes have been made, PAL presents the user with a screen (see Figure 7) in which he gives a description of the changes he has made to the file. The user might have written the documentation in another file prior to the check-in session. In this case, PAL allows the user to specify the name of that file, and PAL reads it in as the description of the change.

When all check-ins have been entered and the PROCESS command given at the top level, the files are sent from the user to PAL's disconnected machine. On receiving each file, PAL will run the appropriate execs as specified in the catalog. If all compilations, assemblies, etc. proceed without error, the derived files (if any) are generated, all new files are installed and put back in circulation, and all affected users are notified of the change.

The previous versions of the files just checked in are not written over. They are renamed, and for any derived file such as a module, a record may be kept of all the files from which the penultimate version of it was derived. Thus, PAL keeps track of successive

Figure 6 Check-in screen

```
                    checkin   P4 PASCAL
                    checkin   P3 PASCAL


File to check in: ==> C1 COPY
Was it modified (yes/no)? ==> yes


                    Press ENTER to finish
```

"releases" of the module and the associated source files. How many such backups are to be permitted is a decision to be made at the local installation. We recommend that files be archived onto tape when the maximum number of backups is reached.

PAL keeps a complete record of changes made to files in its catalog. Figure 8 shows the change-log for a file C1 COPY that has just been checked in (see Figure 6) and documented (Figure 7).

### Maintaining PAL

As mentioned earlier, the utility program named LIBTOOL is provided to the human librarian for updating PAL's data base and catalog. It contains facilities for adding and removing catalog entries, specifying or canceling entry properties such as whether a file gets circulated or backed-up, providing version information, and specifying what is to be done to a file when it is checked back into the library.

We will not present much detail of these functions here, save to say that LIBTOOL does as much check-

Figure 7 Entering descriptions of changes

```
        Description of modifications to C1 Copy
Enter DESCRIPT file name, if any, to use ==>

Defined new type : POLITE_COMMAND_UNIT, a
record for passing around variables and states
related to the execution of a command.
```

**Figure 8 Viewing the change log for a particular file**

```
C1 COPY                              Borrowed by:    PRAGER
Status: Modified                     Checked out: 11/24/81 12:06:30
                                     Checked in: 12/02/81 12:28:29

          Description of changes
          ----------------------
     Defined new type : POLITE_COMMAND_UNIT, a
     record for passing around variables and states
     related to the execution of a command.


**********************************************************************
C1 COPY                              Borrowed by:    PRAGER
Status: Modified                     Checked out: 11/13/81 16:01:45
                                     Checked in: 11/18/81 12:06:31

          Description of changes
          ----------------------

            PF7 to scroll forward   PF8 to scroll backward
                      Press ENTER to end
```

ing and prompting as possible during the process to help ensure that meaningful entries are made to the catalog. To complete the example given in Section 1, though, we show a few screens generated by LIBTOOL for the specification of the dependencies of some of the COPY and PASCAL files of Figure 2.

Figure 9 shows the dependencies of C3 COPY. It says that when C3 COPY is checked in, statement M1 in STAGE1 STMTS is executed, as are P1 and P2 in STAGE2 STMTS, TL1 in STAGE3 STMTS, and P1 in STAGE4 STMTS (see Figure 3 for these statements). Clearly, given the dependencies depicted in Figure 2, these actions are necessary and sufficient for correct regeneration of all files that depend on C3. If other files are checked in along with C3 COPY, all of the associated STAGE1 statements are executed before any of the STAGE2 statements, and so on.

Figures 10 and 11 show the dependencies of C4 and P2, which again can be verified by reference to Figure 2. Note that the headings on PAL's screens can be tailored to the particular local usage; for example, the "STAGEn statements" headings in Figures 9 to 11 can be changed to "MACLIB generations," "PASCAL compilations," etc., or as appropriate.

Now, it frequently happens that identical processing is required for two (or more) files when they are updated. For example, it may happen that two COPY files, say, belong to the same MACLIB, are INCLUDEd in the same PASCAL files, and so are pertinent to the generation of the same TEXT decks, TXTLIBs, and MODULEs. In our example, C4 and C5 have the same dependencies, except that only C4 is included in P2. For such occasions, LIBTOOL allows

the user to say that processing required when file C5 is checked in is the same as that already specified for file C4 (possibly with certain modifications). This feature greatly speeds up the building of the library catalog and reduces errors.

LIBTOOL also asks the librarian for a description of any file being entered into the library. A description here of the general function of a file, along with any documented changes to it (see previous section), provides at least a first pass at up-to-date overall system documentation.

## Current uses of and future enhancements to PAL

PAL was created as a general software development tool, although the POLITE project at the Cambridge Scientific Center was its intended (sole) user. It is being used successfully at Cambridge and is in active use at several other IBM facilities.

One IBM group uses PAL for control of program development, in particular, Structured Programming Facility (SPF) screen design, and currently has a library of about 700 to 800 files. Another group initially used PAL to store documentation only, and now has about 1500 such files in the system. Because of the success which the group had with it there, a second library was established for software development, currently containing over 2000 files, although only a few hundred of these are files that may circulate. These files are of various types, including macro, assembler, exec, and other languages.

PAL's generality has been demonstrated by the way members of this group are using it to store their

**Figure 9 LIBTOOL menu for C3 COPY**

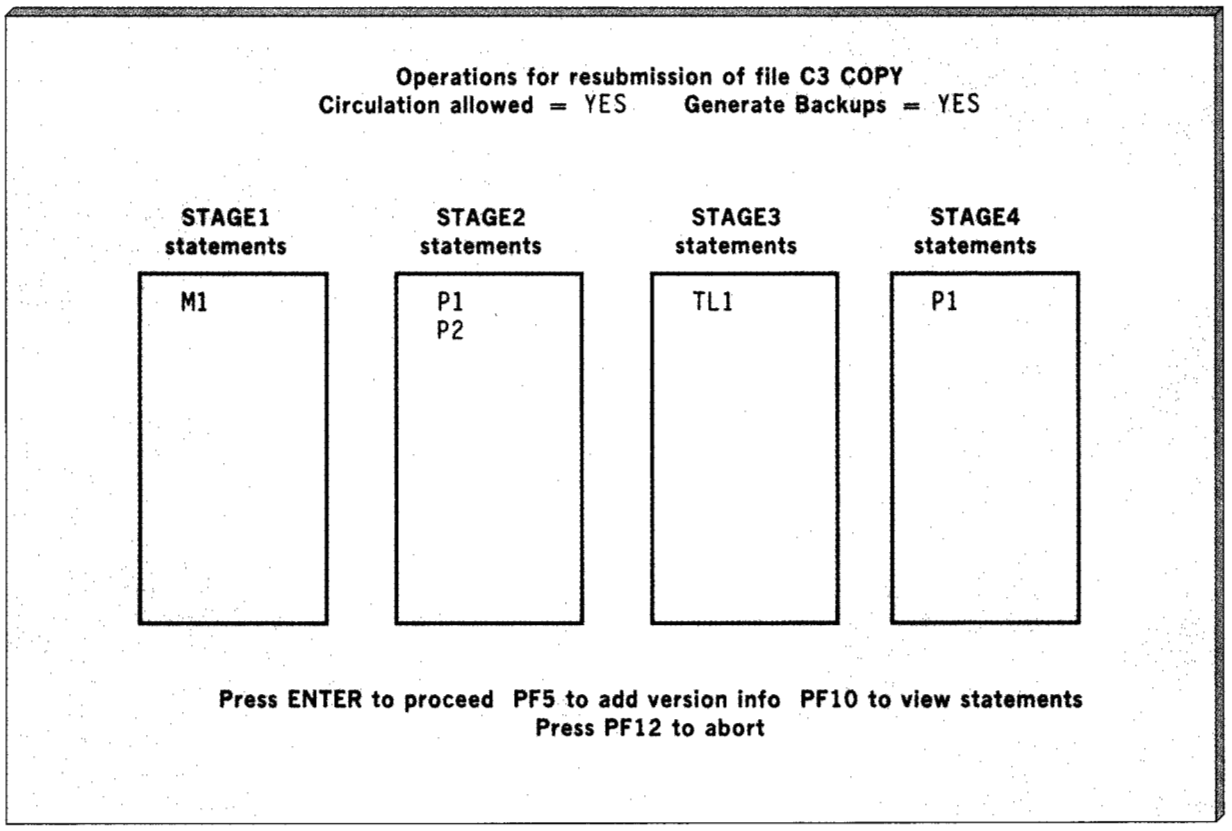
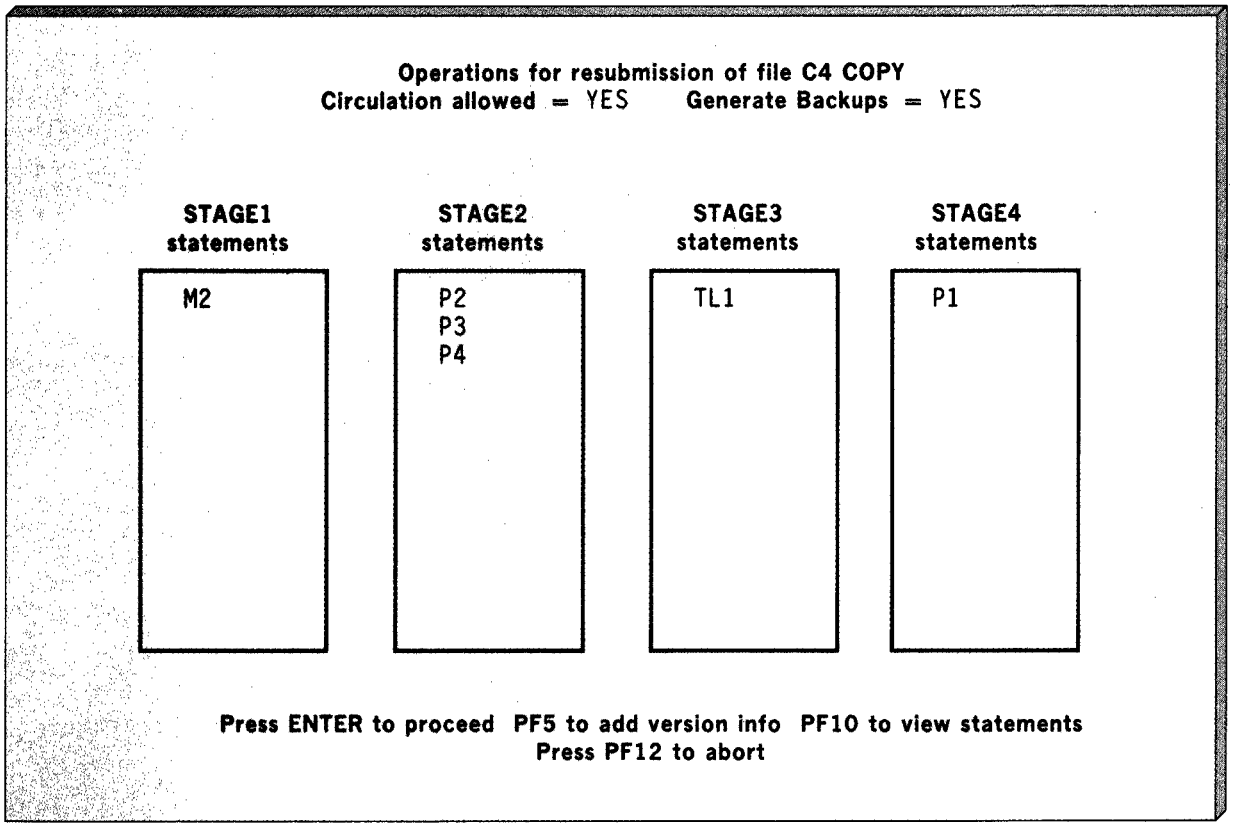


Figure 10 LIBTOOL menu for C4 COPY



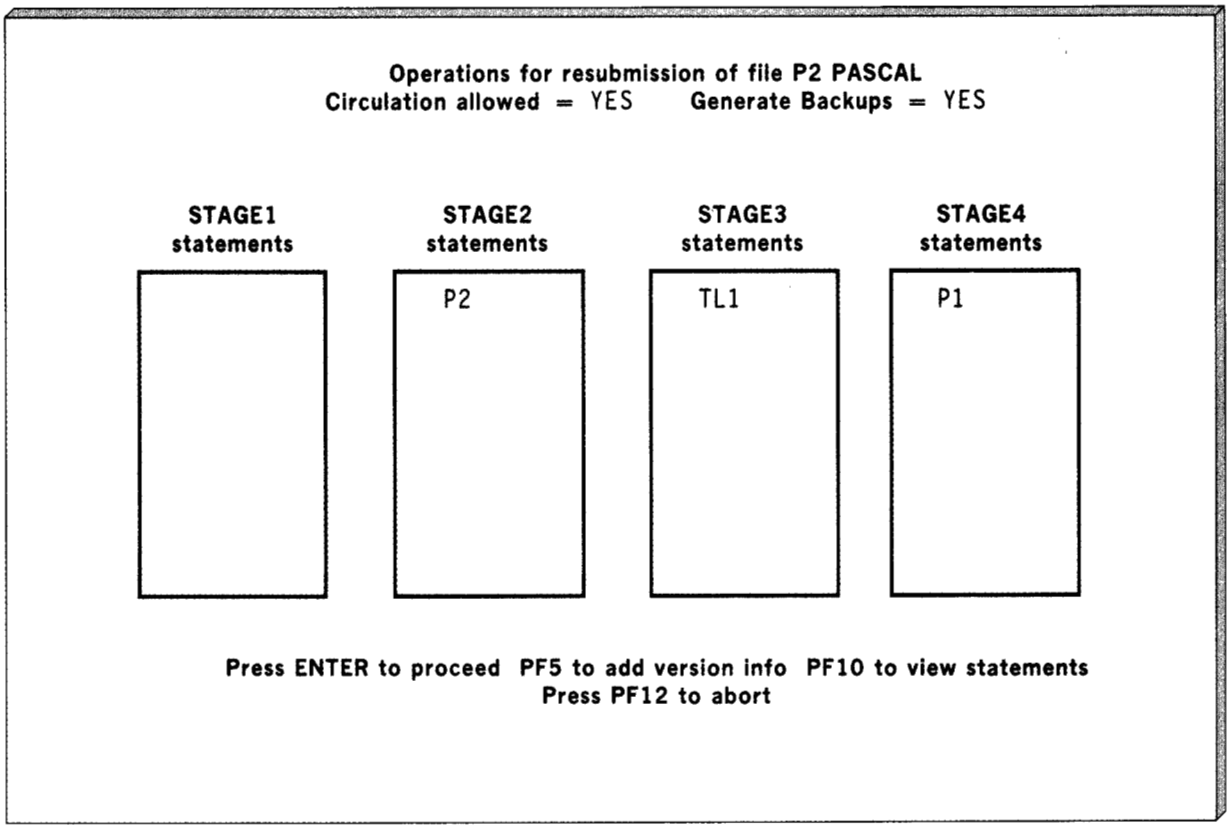

software documentation. They send it Document Composition Facility (DCF)[7] source files, and PAL runs the DCF processor on the files checked in. The files are accepted if DCF issues a zero return code. This library consists entirely of DCF source files, but one can imagine, were disk space plentiful, that PAL could save fully formatted documentation in print format (instead of executable modules in the case of program development).

Although PAL probably does not completely meet any one person's needs, it has received a very enthusiastic response from its users. PAL has been said to be very valuable to its users, because it is flexible and well-structured, and fits their needs well. Users state that it is faster and better than other systems that they know of, and some even claim it to be indispensable.

This attitude is surely due to the need in general for tools such as PAL for project management, rather than because PAL is ideal. In fact, there are several areas in which PAL could benefit from extensions or improvements. These deficiencies have arisen because the intensive use to which PAL is being put was not anticipated when it was written (POLITE then had approximately two dozen source files).

Since PAL accepts check-ins only from the original borrower, and since check-ins are accepted only if the new code compiles cleanly, a change in the interface between code managed by two people

Figure 11 LIBTOOL menu for P2 PASCAL




cannot easily be achieved, since neither one can be the first to check the new code in. This situation can easily be overcome by introducing to PAL the concept of a "job," which consists of a set of files checked out to possibly several people, and which will be processed only when all are received back.

The PF-key usage is very nonstandard, and should be changed to conform to any standard adopted by the installing location.

In order to add or change entries to the catalog via LIBTOOL, the human librarian must log on to the PAL machine, thus temporarily removing it from active service. It has been suggested that updates to the catalog take place remotely, just as checking in and out of files is performed. It has also been suggested that the user not have to link to the PAL machine, but instead send it messages. The current method of linking is annoying at times, because users must reaccess every time the library is updated. The lack of a physical link would also permit use of PAL across different machines over the Remote Spooling Communications Subsystem (RSCS).[8]

A desirable function to be executed when code is checked back in is one that automatically tests the newly generated module(s) to ensure that no regression has occurred. The roadblock here is not so much a problem with PAL, which would treat the test as simply one more step in the check-in process,

but that automatic testing is intrinsically a very difficult problem. It is still in the domain of computer science research, and is rarely tractable except in very simple situations. One potential deficiency with PAL in this regard is that the ultimate answer from such a test must be summed up in a simple return code, which cannot express the full range of possible outcomes.

A code developer who uses PAL will usually test new code before checking it back in. As mentioned before, PAL will only check for syntactic correctness; if the new code passes the test, new systems will be generated and immediately made available to everybody, system developers and end-users alike. It may happen that changes should undergo more extensive checking than a code developer is willing or able to perform, before the new code is made available to users. This checking is easily achieved by setting PAL up with three disks instead of two. The A-disk will be the work disk as before, the B-disk will be the system disk containing all source and most recent modules, etc., and the C-disk will contain the most recent modules that have been adequately tested. So the modules on the C-disk may be as up-to-date as those on the B-disk, or they may lag behind.

**PAL is an automated system that performs the same functions as a human librarian.**

Updating the C-disk is very easily done. An EXEC called INSTALL is written which contains statements to copy to the C-disk all files of interest to end users. INSTALL EXEC is put into PAL using LIBTOOL, and PAL is informed that when INSTALL EXEC is checked in, INSTALL EXEC will be executed. Now systems testers will link to the B-disk to access the latest code; users will link to the C-disk. When project management is happy that the latest changes have been adequately tested, someone will check out INSTALL EXEC and then check it back in, which will trigger the updating of the C-disk. The usefulness of this device was recognized at Cambridge very soon after use of PAL was begun, and the fact that it can be done with the simple addition of an exec points again to PAL's versatility.

## Summary

During the course of a software development effort involving several people, it is often the case that two or more people wish to work on the same pieces of code. It is also the case that access is required to the most up-to-date versions of components of the system being developed. This can be handled by "private" agreements between two or more people, or by using the services of a human librarian to mediate all code transfers. Both of these approaches are unsatisfactory.

PAL is an automated system that performs the same functions as a human librarian. By checking files in and out, it ensures that only one person at a time may modify a file, but allows all project members to have read access to the system at all times. It automatically generates new modules and other dependent files whenever source files are changed, and it keeps documentation of the changes. PAL is a general-purpose tool, in the sense that it does not care what language or languages the system is being written in. It makes backups as files are changed, and keeps version information so that earlier releases of the software may be regenerated.

A further benefit of PAL is that one can be guaranteed of having a working version of the system available at all times. This has proved especially important to POLITE, which has recently been in a "demonstration-intensive" mode. Even if the most recent changes cause a regression to occur, thereby invalidating the latest modules, a past working version can easily be found or created, since PAL keeps backups.

PAL requires its users to describe the changes made to files when they are checked back in to the library. This has the effect of enforcing communication among several people who may be working (in turns) on the same code. This will be especially useful when PAL is extended to allow usage from

remote machines, as outlined in the previous section. In that case, the people involved may have much less personal contact than if they all worked at the same site, and may be less likely to be aware of changes made by one another.


## Acknowledgments

PAL was designed and written by two Massachusetts Institute of Technology students, John C. Gonzalez and Michael J. Wissner, who worked on the POLITE project during the summer of 1981. Thanks are due to Forest Gordon and John Webster for making enhancements to PAL, and to Peter Hardy and Sheldon Borkin for their support and suggestions.


## Cited references


1. J. M. Prager and S. A. Borkin, *POLITE Project Progress Report*, Technical Report G320-2140, IBM Cambridge Scientific Center (April 1982); available through IBM branch offices.
2. *Pascal/VS Language Reference Manual*, SH20-6168, IBM Corporation; available through IBM branch offices.
3. *IBM Virtual Machine/System Product: EXEC2 Reference Manual*, SC24-5219, IBM Corporation; available through IBM branch offices.
4. *IBM Virtual Machine/System Product: CMS User's Guide*, SC19-6210, IBM Corporation; available through IBM branch offices.
5. M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1,** 364–370 (December 1975).
6. T. R. Horsley and W. C. Lynch, *Pilot: A Software Engineering Case Study*, Xerox System Development Department Technical Report, Palo Alto, CA (July 1979).
7. *IBM Document Composition Facility: User's Guide*, SH20-9161, IBM Corporation; available through IBM branch offices.
8. *IBM Virtual Machine Facility/370: Remote Spooling Communication Subsystem (RSCS) User's Guide*, GC20-1816, IBM Corporation; available through IBM branch offices.



**John M. Prager** *IBM Academic Information Systems, Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Dr. Prager is a project leader at the IBM Cambridge Scientific Center. Since joining IBM in 1979, he has worked in the office systems area, in particular on the POLITE project, for most of that time. His current interests include the development of user interfaces for powerful personal workstations using techniques from artificial intelligence. He has published several papers and technical reports and is a member of the Association for Computing Machinery, the British Computer Society and the Institute of Electrical and Electronics Engineers Computer Society. Dr. Prager received a B.A., Diploma in Computer Science (with distinction), and an M.A. from the University of Cambridge and a Ph.D. in computer science from the University of Massachusetts at Amherst.



Reprint Order No. G321-5192.