A Proposal For An

Interactive Version Of PL/C

Richard W. Conway
Thomas R. Wilcox*
Steven L. Worona

(71-119)

Department Of Computer Science
Cornell University

Research Report 71-119

December 1971

* Current Address: University Of Illinois

A. Introduction

PL/C is a compiler for a dialect of PL/I produced by a development group at Cornell. Its purpose was to explore the limits of diagnostic assistance that could be provided by a compiler operating in a conventional batch-processing environment. For the most part this assistance is implicit and is provided automatically by the compiler. The most remarkable characteristic of PL/C is its perseverance-- it completes translation of every program submitted and continues execution until a user-established error limit is reached. This requires that the compiler repair errors encountered during both translation and execution.

At least initially we viewed this 'error repair' as an alternative to real-time interaction between the programmer and the translator. Since the programmer is 'not available' at the time that difficulty is encountered PL/C attempts to take an action such as he might if he had the opportunity. Hopefully in at least some cases this action is useful and reduces the number of cycles required in the diagnostic dialog. Now we are interested in the possibility that even when the programmer is available it may be useful to have the translator attempt to repair errors. The difference would be that now the repair would be offered to the user as a suggestion that he could accept or reject. To the extent that the translator can construct valid repairs this should help the user and modestly accelerate the dialog.

The second point at issue is processing efficiency -- the required consumption of machine resources. In order to achieve the flexibility necessary to respond to the user's requests most interactive systems have resorted to translation by interpretation. That is, the 'generation' of the actual machine instructions is postponed until run-time, and generally is repeated each time that a source-language statement is to be executed. We are interested in exploring the question of whether this relatively costly mode of translation and execution is really required. We would like to try to design an interactive system that is sufficiently flexible to satisfy most user requirements, and at the same time retains the inherent efficiency of compiled translation.

We see two rather distinct approaches to this task and intend to pursue both. One could possibly construct an 'incremental compiler' in which the generated code is segmented by source statement and is replaceable on a source statement basis. Alternatively one could employ a more conventional compiler and, if it were fast enough, simply recompile whenever the user required changes that made this necessary. This present proposal follows the latter course. We think that possibly we can design a system that uses many of the major modules of the existing PL/C compiler, that provides sufficiently interactive performance for most users, and that limits the frequency and extent of recompilation so that

fficiency is preserved. We would expect the system to require everal times as much cpu time as the same task would require nder conventional batch PL/C but this might still be a fraction f that required for an interpretive system.

The proposal is based on design discussions that included ark Bodenstein, Robert Fisher, David Gries, William Maxwell and ohn Williams of Cornell, and Howard Elder of the University of orth Carolina, as well as the authors. The purpose of the roposal is to solicit reaction to the utility of the system efore we undertake the implementation, and to solicit detailed uggestions as to syntax, function and strategy while the system s still pliable. If such a system is implemented it will have o have a name. We know from sad experience that preliminary ames--especially those chosen in jest-- have amazing tenacity. he hypothetical interactive version of PL/C does not yet have a ame but we need some convenient identifier in the proposal. I-L/C is used in the hope that it is so unattractive and clumsy hat it cannot possibly endure.

The Environment

We expect that I-PL/C will be used in close cooperation ith batch PL/C. To facilitate this use we intend to provide bsolute bi-directional compatibility between the two versions f PL/C. Programmers should be able to use whichever version of L/C is appropriate to a particular task without having to emember which features are present in which, or what special estrictions are required. A programmer should also be able to ove back and forth between the two versions with a single rogram with ease and confidence. The obvious use will be to nter, edit and test a program under I-PL/C and then submit it o batch PL/C for execution. Since PL/C will retain its present pward compatibility with the IBM compiler for PL/I there will ctually be three possible modes of execution.

I-PL/C is designed to run under OS rather than outside it. t does not require the dedication of a partition or a articular set of devices. The I-PL/C system does not include ither a terminal manager or a time-sharing supervisor. It elies on TCAM for the former and optionally, upon TSO for the atter. That is, the user can invoke I-PL/C as a normal OS job nd communicate with it through TCAM as long as the job is xecuting. In this mode each user would require a separate artition and must hold that partition throughout the period hen he is active from the terminal. If the terminal is ctually being used in an interactive mode the session will ikely be long (relative to the amount of cpu time required) and here will be relatively long periods when the partition is in act idle as the program waits for response from the terminal.

In order to avoid these idle periods a supervisor can ttempt to have two or more users share the same core. At least nitially I-PL/C will not have such a supervisor of its own and

will be set up so that it is adaptable to the TSO supervisor which is supplied (without charge) by IBM. This is a time-slicing, roll in/roll out supervisor that should permit a very small number of users to share a partition. It would not appear to us that TSO is going to permit large numbers of students to run from terminals simultaneously regardless of what sort of language processors it employs. Until some successor to TSO is available it seems likely that high volume instruction will continue to be batch oriented and that systems such as I-PL/C will complement the batch systems for relatively small numbers of users with special requirements and more generous funding.

If a significant number of users wish to use I-PL/C under TSO it may be preferable for the installation to place at least the program entry phase of the compiler (which is entirely reentrant) in the TSO linkpack area. This will greatly reduce the partition size needed during program entry, which will both increase the number of partitions that can be used and reduce the TSO swapping load.

We assume that the signon dialog is a supervisory function handled outside of PL/C. This dialog concludes with the equivalent of an EXEC I-PL/C. PL/C begins at this point with an acknowledgement line:

PL/C READY

and continues in control until the I-PL/C command

%FINISH

is given by the user. (Throughout this paper output produced by I-PL/C is shown underlined. This is for clarity here and will not be done on the terminal.)

The present proposal assumes that the user will employ a character-at-a-time typewriter-like terminal. It is assumed that the terminal has the equivalent of two non-character control keys that we will call attention and end-of-line (eol). eol will always be used to signal the completion of an entry line. It marks the end of a particular user response. The significance of the attention signal depends upon context. If attention is signalled during a user response (that is, prior to eol) it simply deletes the partial line that may have been entered. If attention is signalled when the computer is transmitting to the terminal it aborts the remainder of the particular transmission and causes the system to proceed exactly as if that transmission had been normally completed. If attention is signalled at any other time the system will be interrupted (at the next convenient point) and return to the terminal for instructions. The typewriter terminal can of course be simulated by a multiple-line display terminal (CRT) with ease. Perhaps a future version of I-PL/C will exploit the additional capability of such display terminals.

The I-PL/C Source Language

The I-PL/C source language contains four distinctly ifferent types of elements: 1) PL/C source statements, 2) PL/C ontrol cards, 3) data, and 4) I-PL/C commands.

The source statements for I-PL/C are exactly the same as atch PL/C statements which, in turn are a compatible subset of he PL/I language. I-PL/C will be linked to Release 7 of PL/C hich will represent very little functional addition relative to he current Release 6.5. That is, in comparison to full PL/I elease 7 will still lack list processing, compile-time acilities (except for INCLUDE), multi-tasking, direct-access uxiliary files and miscellaneous minor features. The only ource language changes will be the addition of a HALT statement nd the addition of a SYSTEM option to the PUT ON and PUT OFF tatements. HALT will be considered a null statement by batch L/C, and will signal a programmed return to the terminal in I-L/C. The PUT SYSTEM OFF statement will suppress all system-enerated output during execution. (This includes prompting for ata input, halt location announcements, error messages and the ermination message.) PUT SYSTEM ON will resume the printing of his output. This selective suppresion will permit a user to ave complete control over the content and format of execution utput, and hence offers the opportunity to construct other nteractive systems using I-PL/C. PUT SYSTEM ON and PUT SYSTEM FF are both considered null statements by batch PL/C. nternally the batch Release 7 will involve some restructuring o accomodate the introduction of I-PL/C but this will not ffect its functional capability, nor significantly degrade its atch performance.

The I-PL/C control cards are also exactly the same as those or batch PL/C. These are:

```
*PLC options
*PROCESS options
*INCLUDE pds name
*DATA
```

he meaning of the control cards is the same except for *DATA hich does not automatically signal the start of execution under -PL/C. (See Section F.) A third parameter is permitted on the IME option on the *PLC control card to specify the frequency of -PL/C execution progress reporting (see Section E.2)

To preserve compatibility with batch PL/C all data entered rom a terminal under I-PL/C is assumed to be card images. xactly 80 columns of the terminal line are employed and the ules for placement, punctuation and continuation over a 'card oundary' are exactly the same as for real cards under batch L/C.

It is necessary to distinguish very carefully in this iscussion between lines and statements. The input process

under I-PL/C is line-oriented (just as the input process under batch PL/C is card-oriented) and the editing function of I-PL/C is also necessarily line-oriented. However, the compiler itself is statement-oriented. We believe that the numbering scheme described here largely masks this distinction from the user and permits communication in a natural and obvious way. Numbers are assigned to line segments. A line segment is a portion of a line -- not crossing a statement boundary. For example:

```
X =
   A + B;
Y = C + D; Z = E + F; W =
G + H; GOTO L;
```

represents four lines, five statements and seven line segments. All line segment numbers are unique so that they can be used directly during editing, and I-PL/C can use those segment numbers that correspond to the beginning of a statement for normal PL/C statement numbers. However, this does mean that the statement numbers produced by PL/C for a given program will not be the same under the batch and interactive versions.

Line segment numbers are assigned automatically by I-PL/C. Two different increments are used. The 'long' increment (default value is 10) is used between statements in initial entry. The 'short' increment (default value is 2) is used for statement continuations in initial entry, and all line segments during editing. To illustrate, the lines given above would have line segments numbered as follows:

```
10 X =
12    A + B;
22 Y = C + D;
32 Z = E + F;
42 W =
44 G + H;
54 GOTO L;
```

The I-PL/C commands are listed in Tables 1 and 2 along with a brief description. Examples showing their use, and the system response are given in Section E. Table 1 gives the editing and control commands which may be entered at any time. The %REPLACE, %DELETE, %STEP, %SHIFT and %RENUMBER commands control the construction and alteration of a file of input line images. These commands all assume that the system is in 'program entry mode' -- or return it to that mode if it is not. The %GENERATE and %RUN command cause the system to leave program entry mode and proceed to generate object code and begin execution.

Table 2 gives the execution commands which will be accepted only after a program has been compiled and has entered execution. Perhaps the most important characteristic of I-PL/C execution is simply the fact that the terminal serves as the primary input and output device, and that this input and output occurs in real-time while the program is executing. This alone

ermits types of interactive computing that are not possible
ith a batch system--even when operating from a terminal under a
emote-job-entry system. In addition to this capability I-PL/C
rovides commands designed to facilitate debugging of the
rogram during execution. These commands specify the conditions
nder which control is to return to the terminal, and provide
he equivalent of a very small subset of the PL/C source
tatements that can be executed directly from the terminal. The
arious forms of the %HALT command specify the conditions under
hich the system should return control to the user at the
erminal. He may also regain control at the terminal by
ignalling 'attention'. When the terminal has control the
DISPLAY, %PUT SNAP, %PUT FLOW, %PUT ALL and %PUT ARRAY commands
ermit the user to examine the state of his program; the %FLOW,
OFLOW, %CHECK, %NOCHECK, %PUT ON and %PUT OFF commands allow
im to alter the diagnostic options in effect; and the %SET and
GOTO commands allow him to redirect the program in very limited
ays. These commands have the same effect as would the
xecution of the analogous source statement in object code, but
he commands are executed immediately upon entry and do not
ecome part of the object program.

The general restrictions on the use of the four different
ypes of elements are the following:

1. Statements, control cards, data and commands can all be entered from a terminal.
2. Commands can only be entered from a terminal.
3. Statements, control cards and data can reach I-PL/C from other sources via INCLUDE.
4. Statements, control cards and data can be routed from I-PL/C to batch PL/C by either the INCLUDE or SUBMIT mechanisms.
5. Statements are free-field, and subject to user-specified source margins exactly as in batch PL/C. The user may enter several statements per line (although this practice should probably be discouraged) or use several lines for a single statement.
6. Data is entered from a terminal as a continuous stream of 80 column card images, exactly as real cards under batch PL/C.
7. Both commands and control cards must be prefaced with a special recognition character, and this character must appear in the first user column in an input line. The recognition characters * for control cards and % for commands are used in this description but the selection of recognition character will be an option available to each installation. (An installation may elect to use the same recognition character for both commands and control cards.)
8. The source carriage control characters of batch PL/C will be accepted but ignored by I-PL/C.

Table 1 I-PL/C Edit And Control Commands

```
Notation:
     [ ] denotes an optional element
     [ ]* denotes an optional repeatable element
     [ ]1* element must appear once, may be repeated
     n1, n2, ...   are line segment numbers
     ,=-()  are all equivalent to a blank to the command scanner
     only the first 4 characters (including %) are scanned

%REPLACE n1 -- delete line segment n1, and prepare to
          receive a new segment to take its place
%n1 -- prepare to receive line segment n1 (where n1 does not
          already exist)
%DELETE n1 [,n2] -- delete line segments from n1 to n2
          inclusive. START may be given for n1; END for n2
%LIST n1 [,n2] -- list the contents of line segments
          n1 [ to n2]. START or ALL may be given
          for n1; END may be given for n2.
%STEP d1 [,d2] -- replace value of 'long' line segment
          number increment by d1; replace 'short' increment
          with d2. Default 'long' is 10; 'short' is 2.
%SHIFT n1, n2 [,n3] -- increase line segment numbers from
          n1 to n3 by an amount n2-n1. If n3 is not given n3=n1.
          (Line segments cannot be reordered.  n3' must be < n4
          where n4 is original number of line segment after n3)
%RENUMBER n1, d, n3 -- renumber line segments from n1 to n3
          so that their numbers are n1, n1+d, n1+2d, ...
          (As with SHIFT segments cannot be reordered)
%TAB [,c1]1* -- delete all old tabs and set tabs at c1, c2, ...
%GENERATE -- perform the semantic and code generation phases
%RUN [,ni]* -- perform the semantic and code generation phases
          (if not already done) and begin execution.
          '%RUN n1' is equivalent to '%RUN %HALT AT n1'
%SUBMIT -- submit the statement-control card-
          data file (as edited) to batch PL/C
          (This command may be given repeatedly, and has
          no effect on ability to also continue under I-PL/C.)
%SAVE pds-name -- save the current source file as an OS pds
%COPY pds-name -- copy the pds into the source file
%CLEAR -- completely reinitialize the source file
          (Since the effect of an accidental %CLEAR could
          be disastrous the system will produce a warning and
          ask for confirmation before executing.)
%FINISH -- return control to the operating system
```

Table 2 I-PL/C Execution Commands

```
FLOW
NOFLOW
CHECK
NOCHECK
GOTO label
PUT ON
PUT OFF
PUT SYSTEM ON
PUT SYSTEM OFF
PUT SNAP
PUT FLOW
PUT ALL
PUT ARRAY
```

Each of the above commands is analogous to a source statement and its execution has the same effect as if that statement were encountered in object code. Note however that the command is executed once when given, and is not inserted into the object program. Note also that the command is more restricted than the statement in some cases--no parameters on FLOW, no DEPTH.

```
DISPLAY variable -- equivalent to execution of
        PUT LIST( variable );

SET variable = value -- equivalent to execution of
        GET DATA;

HALT AT [ni]1* -- insert HALT immediately before statement
        ni in object code
HALT AT ALL -- insert HALT before each statement in
        object code
HALT ON CHECK -- insert HALT as the first statement
        to be executed when the CHECK condition is raised
HALT ON FLOW -- insert HALT as the first statement
        to be executed when the FLOW condition is raised
HALT ON CALL -- insert HALT before each CALL

NOHALT AT ... NOHALT ON ... -- negates effect of HALT

)1 or %eol -- null command, resume execution
        (equivalent to an end-of-file if given in response to
        a request for data from the terminal)
```

D. General Strategy Of I-PL/C

The basic strategy of I-PL/C is very simple:

1. As long as the user continues to enter new lines in normal sequential order I-PL/C proceeds essentially like batch PL/C except that it also builds a file of the source line images.

2. Whenever the user edits a line previously entered I-PL/C alters the source file as directed, and then using that file automatically and invisibly recompiles the program.

The recompilation frequency and effort is reduced by several means:

a. Since it is assumed that the very large majority of editing actions will affect the most recently entered line (presumably in response to error conditions reported by the system) special procedures are used for this case. I-PL/C 'conditionally compiles' each line as it is initially entered and does not actually accept the line until the user has declined to edit it and has moved on to enter a new line. Hence editing of the most recently submitted line involves recompilation of only that line and not the entire program.

b. The system automatically checkpoints itself so that for any editing the recompilation begins at the last checkpoint before the edited line and not at the beginning of the program.

c. The recompilation that is done takes place in stages, so that if the user edits a number of previous lines in order of increasing line segment number he incurs only one recompilation and not one recompilation for each line edited.

The efficiency of this strategy obviously depends importantly on the user's behavior. We believe that the design favors behavior that is natural and convenient. However, except possibly for rank beginners, users should be told roughly what is taking place so that they will know what I-PL/C considers to be 'natural'. For example, the difference between 'top down' and 'bottom up' editing will be substantial and the user should be warned that the former is preferred.

An overall view of information flow in a dual PL/C system (both batch and interactive) is shown in Figure 1. A detailed description of the structure and management of the source file, and the tactics for achieving conditional compilation of the most recent statement, and invisible recompilation of the entire program is given in Section F.

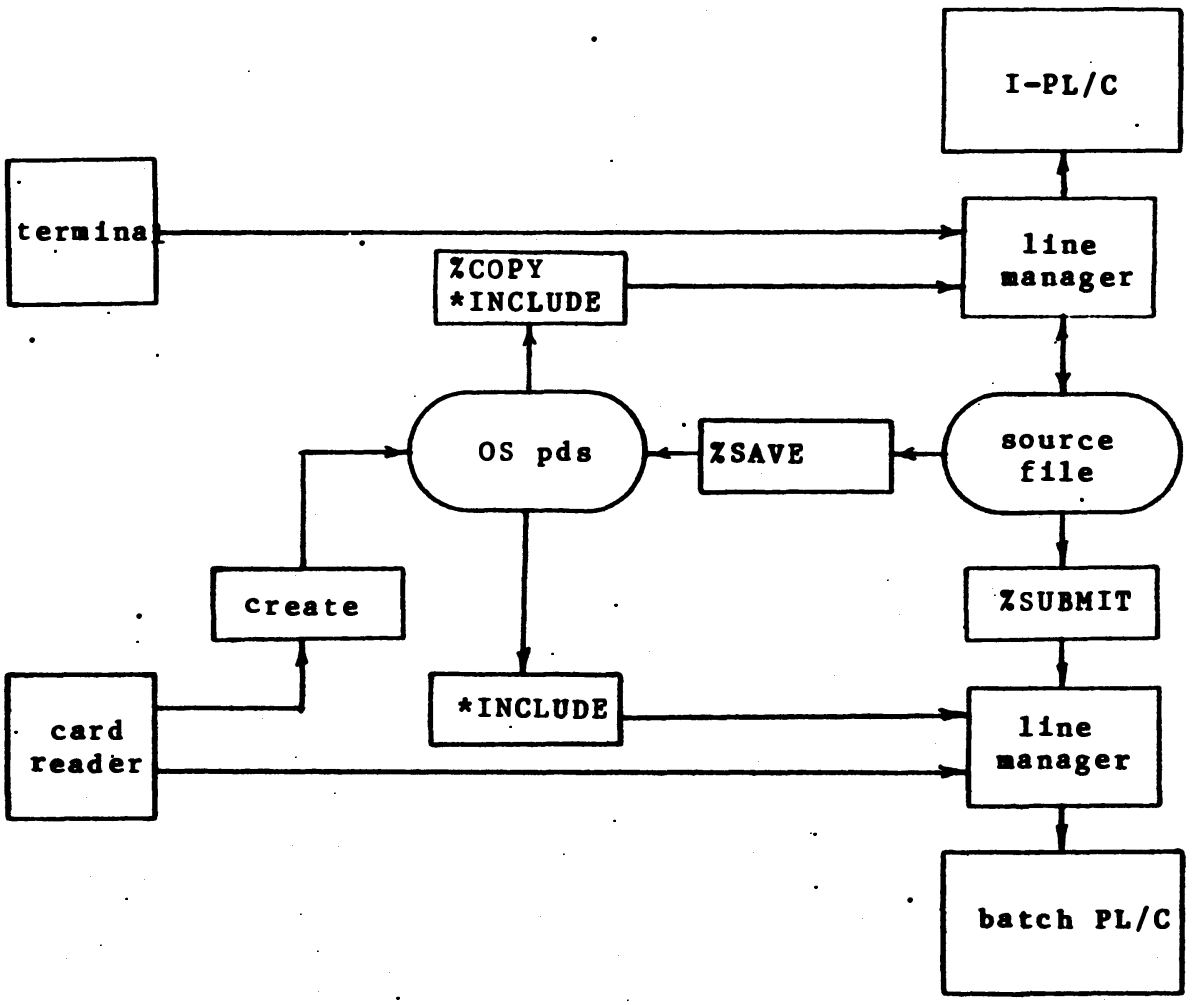

Figure 1. Input Data Flow In A Dual PL/C System

E. Operation Of I-PL/C

The general nature and external appearance of I-PL/C will be described by means of a sequence of examples of the dialog that might occur between a user and the system. (As noted previously all system-generated output will be underlined in this paper.) A minimum complete program dialog would be something like the following:

```
PL/C READY
10 *PL/C eol
20 P1: PROC OPTIONS(MAIN); eol
30       PUT LIST('HELLO'); eol
40 END P1; eol
50 %RUN eol


HELLO
IN STMT 40 PROGRAM RETURNS FROM MAIN PROCEDURE
COMPILE TIME 0.11 SECONDS.
EXECUTION TIME 0.10 SECONDS.
PM DUMP WILL FOLLOW
% FINISH eol
PL/C FINISHED
```

A longer and more illustrative dialog is the following:

```
PL/C READY
10 *PL/C ID='WAGNER' eol
20 RAW: PROC OPTIONS(MAIN); eol
30       PUT LIST('BEGINNING OF EXECUTION OUTPUT'); eol
40       PUT SKIP LIST('TEST OF SQRT ERROR') eol
C42       ; eol
52 %60 eol
60       X = -4.0; eol
```

```
70        Y = SQRT(X); eol
80 %LIST 40 eol
40        PUT SKIP LIST('TEST OF SQRT ERROR')
80 %LIST 42-70 eol
42        ;
60        X = -4.0;
70        Y = SQRT(X);
80        Z = (X + Y; eol
IN 80 ERROR SY04 MISSING )
PL/C REPLACES 80 WITH Z = (X + Y);
90        W = ( X + Y * Z ; eol
IN 90 ERROR SY04 MISSING )
PL/C REPLACES 90 WITH W = (X + Y * Z);
100 %90 eol
90 ALREADY EXISTS
100 %REP 90
90        W = (X + Y) * Z; eol
100 %STEP 5 eol
95        STOP; STOP; STOP eol
C107        ; eol
112 END RAW; eol
117 %LIST 52 eol
52 DOES NOT EXIST
117 %LIST (100,107) eol
100             STOP; STOP
107       ;
117 %GENIROGL eol
COMPILATION COMPLETED IN 0.16 SECONDS.
```

```
% LIST 10 eol
10 *PL/C ID='WAGNER'
% RUN eol

BEGINNING OF EXECUTION OUTPUT
TEST OF SQRT ERROR
****** IN STMT 70 ERROR EX8D   THE ARGUMENT IS NEGATIVE
% DISPLAY X eol
X = -4.00000E+00
% SET X = 6.4 eol
% eol
IN STMT 95 PROGRAM IS STOPPED
COMPILE TIME 0.16 SECONDS.
EXECUTION TIME 0.11 SECONDS.
PM DUMP WILL FOLLOW
% SUBMIT eol
SUBMITTED TO BATCH PL/C
% FINISH eol
PL/C FINISHED
```

In statement 80 in this example the user accepted the repair made by I-PL/C and in statement 90 he substituted his own. Since he never edited any statement except the most recently entered the system was never forced to recompile. The following fragments of dialogs illustrate commands that do require recompilation.

```
. . .
210 X = Y ; eol
220 END; eol
230 %135 eol
135 DECLARE Y FIXED BINARY; eol
```

```
137 %DELETE 180 eol

180 %230 eol

230 EN4: PROC(Q); eol

240 . . .
```

en the user denied the initial 230 prompt and specified edit statement 135 I-PL/C went into recompilation mode. Beginning the last checkpoint before 135 (presumably the beginning of e program with statement numbers this small) it has recompiled l of the source line segments whose numbers are less than 135 en it gives the 135 prompt. Statement 135, supplied from the rminal, is compiled and the system prompts for 137 (using the nort' line number increment since it is in recompilation de.) The %DELETE 180 command causes recompilation to proceed to 180. (%REPLACE 180 would be equivalent in this case.) nce no new text is given for 180 the %DELETE takes effect and compilation proceeds to 230, which is the point where the ginal entry left off. If the edited statement were erroneous e dialog might appear as follows:

```
. . .

210 X = Y ; eol

220 END; eol

230 %135 eol

135 DECLARE Y FIXED BINARY; eol

IN_135_ERROR_SX27_MULTIPLE_DECLARATION

PL/C_REPLACES_135_WITH_DECLARE_$V001_FIXED_BINARY;

137 %135 eol

135 DECLARE YY FIXED BINARY; eol

137 %210 eol

210_ALREADY_EXISTS

137 %REP 210

210 X = YY;

212 %230 eol

230 EN4: PROC(Q); eol

240 . . .
```

Another possibility is that the edited statement itself is syntactically correct, but the change results in an error in a following statement -- one that had previously been correct:

```
. . .

210 X = Y ; eol

220 END; eol

230 %135 eol

135 DECLARE Y FIXED BINARY;

137 %230 eol

150 DECLARE Y CHARACTER(5);

IN 150 ERROR SY27 MULTIPLE DECLARATION

PL/C REPLACES 150 WITH DECLARE $V001 CHARACTER(5);

152 %135 eol

135 DECLARE YY FIXED BINARY; eol

137 %REP 210 eol

210 X = YY; eol

212 %230 eol

230 . . .
```

In this example the system began recompilation, and then began again when the user discovered that his first change was not correct.

In the previous examples %REPLACE and %DELETE are completely interchangeable. Given either the system recompiles to the specified line segment and returns to the terminal for the replacement text (which is null for a deletion.) %DELETE exists to permit block deletions as shown below:

```
. . .

210 X = Y; eol

220 END; eol

230 %DELETE 130-150 eol

150 . . .
```

1 Data_Input_From_The_Terminal

GET statements for which no file specification is given, d those with FILE(TERMINAL) will look to the terminal as the urce of data. However the system will first use any data that ght have been entered after *DATA and before *GENERATE or UN. When this data (in the source file) is exhausted the stem returns to the terminal for further requirements. The stem expects to receive 80_column_card_images_from_the rminal_in_exactly_the_same_format_as_cards_for_batch_PL/C. If l is signalled before a full 80 columns have been entered the stem will pad the entry on the right with blanks to create a ll card image. If eol is signalled for a null entry this is terpreted as an end-of-file. In retaining complete mpatibility with batch PL/C, I-PL/C acquires some slightly rprising characteristics for a terminal system. For example, ppose a program has two GET statements:

```
. . .

150 GET LIST( A );
. . .

220 GET LIST( B );

. . .
```

en statement 150 is encountered in execution:

```
GET 150

5, 8 eol
```

e value 5 is taken for A, and the value 8 is held for the next ta request. If this is statement 220 the value 8 is assigned B -- and_statement_220_does_not_return_to_the_terminal_for ta.

If a PUT SYSTEM OFF statement has been executed (and no PUT STEM ON has been executed since) the prompting message 'GET ' will not be given. This will permit the user to generate s own data request messages without having his output uttered with system prompts.

Since under I-PL/C a single device serves for both input d output there is a synchronization problem that does not ist in batch PL/C. For example, suppose a program includes e following sequence of statements:

```
. . .

100 PUT LIST('A');

110 PUT LIST('B');
```

```
120 GET LIST(C);
130 PUT LIST('D');
140 PUT LIST('E') SKIP;
. . .
```

We assume that in most cases the user would rather have execution output look like:

```
A                B

GET 120

5 eol

D
```

rather than:

```
GET 120

5 eol

A                B               D
```

so a PUT SKIP is assumed before each GET statement. Some mechanism (as yet undesigned) will have to be provided to permit the user to suppress this default assumption. The question of how to interpret a GET SKIP with respect to output lines is also undecided.

E.2 Progress Reporting During Execution

During execution of a program under I-PL/C every t cpu seconds (where t is the third parameter of the user-supplied time limit option) the system will print a time/progress message. For example, with the default value of 10 this would appear:

```
10 SEC
20 SEC
30 SEC
```

These messages will normally be interspersed with other terminal input and output and will indicate the relationship between cpu time allocated to this task and real time, and help the user detect unwanted loops. The user can, of course, effectively eliminate these messages by setting a large value of t in his time option.

Internal Structure Of I-PL/C

The following discussion assumes that the reader has a general knowledge of the internal structure and terminology of batch PL/C -- such as is given in the second reference. The general organization of the batch version of PL/C is shown in Figure 2. Relative to this organization the implementation of I-PL/C involves the following:

1. The replacement of CONTROL and SYSTEM INTERFACE with new modules.
2. The addition of a new module to manage a temporary file for the source program. (Batch PL/C does not retain source card images.)
3. The addition of an EXECUTION COMMAND module to execute the 'immediate' commands originating from the terminal.

The other modules of batch PL/C will undergo minor changes to accomodate the dual nature of the system, but for the most part will not be seriously altered from their batch PL/C form or function.

The resulting organization for I-PL/C is shown in Figure 3. Basically it is the task of the new CONTROL, with assistance from the SOURCE FILE MANAGER (SFM), to shelter the rest of the compiler from any knowledge that it is going to operate in a very curious way.

A more detailed view of the principal procedure modules and information flow during program entry in I-PL/C is shown in Figure 4. All of the other modules are completely controlled by the CONTROL module. (The control and information flow relative to CONTROL have been omitted from Figure 4.) CONTROL is called by the operating system interface when a user has successfully signed on. CONTROL initializes the compiler by clearing the workspace, loading the 'program entry modules' (those shown in Figure 4), by making initializing calls upon the syntactic analyzer (SYNA) and SFM, and by printing the message: PL/C READY. The first call to the prompter produces the 10 *PL/C prompt and the dialog is under way.

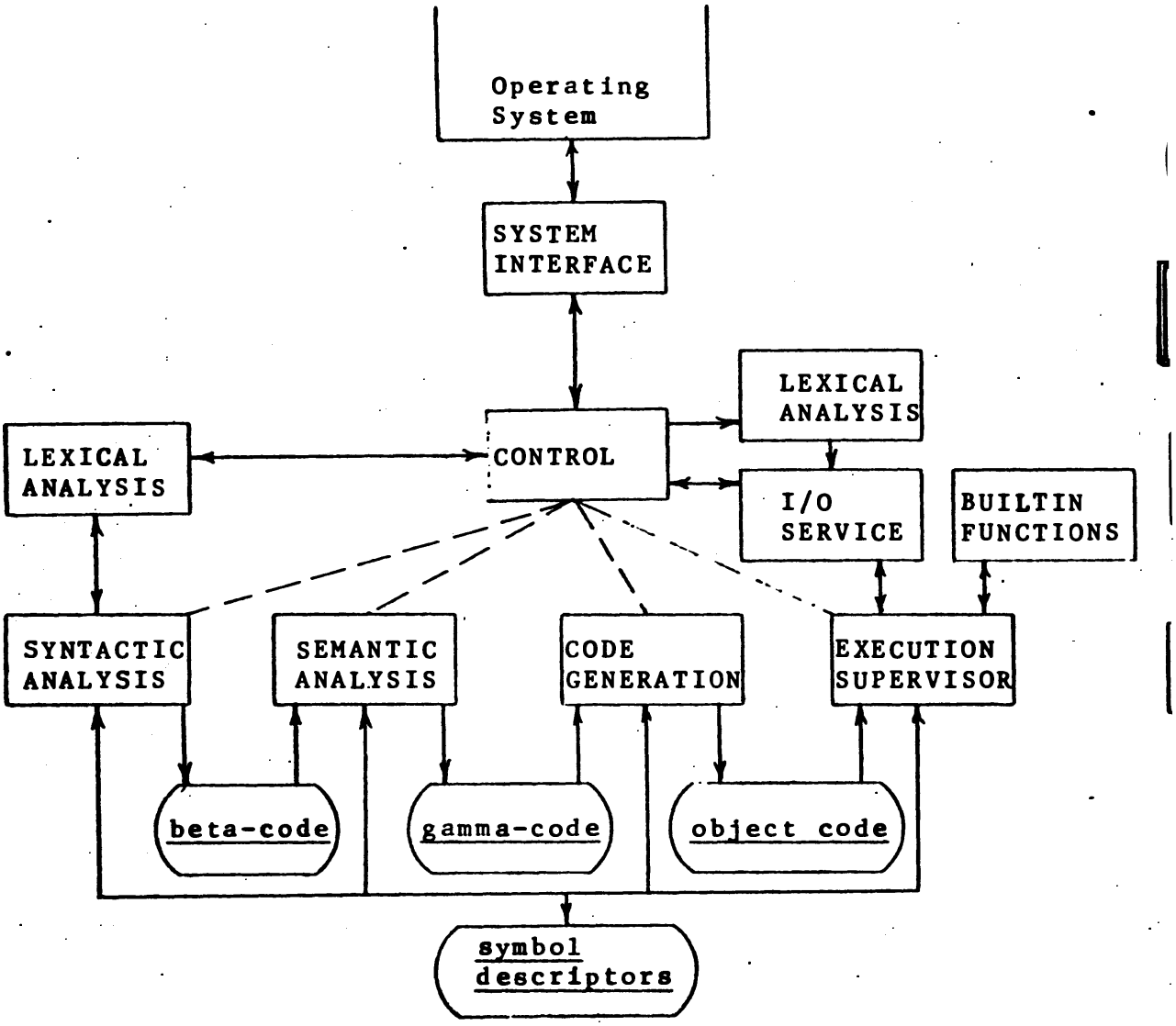

Figure 2. Organization Of Batch PL/C

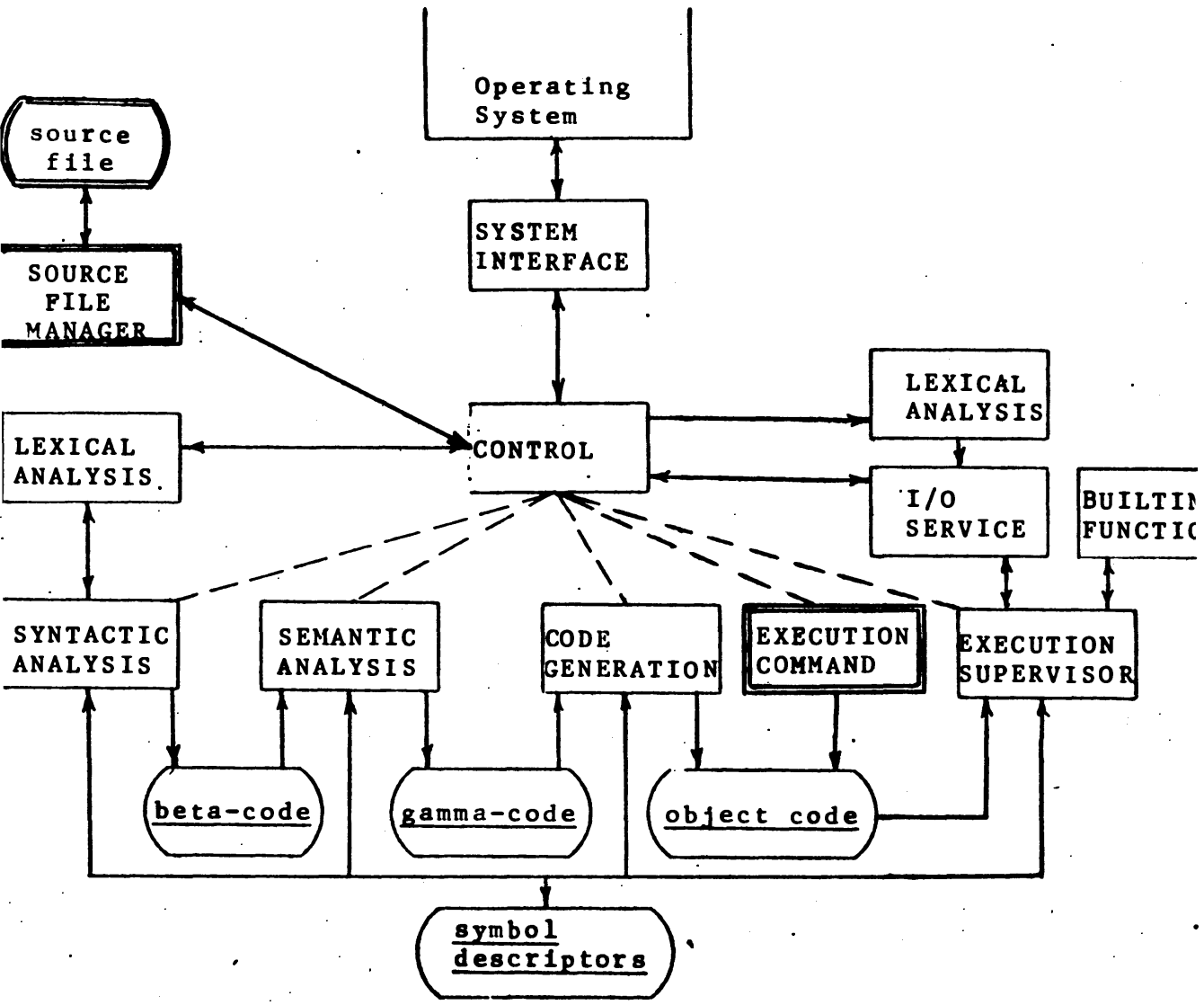

Figure 3. Organization Of I-PL/C

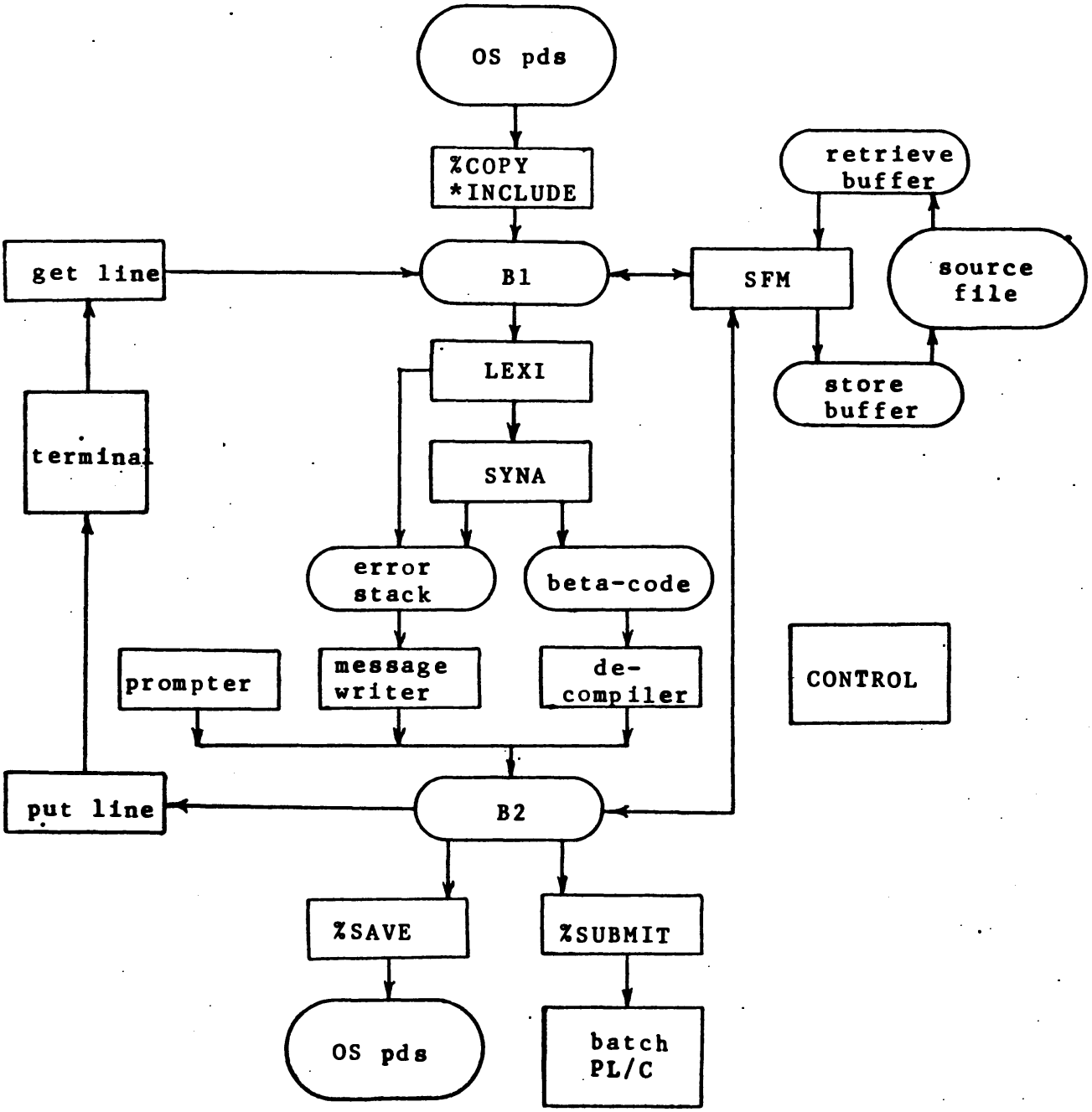

Figure 4. Information Flow During Program Entry In I-PL/C

A line received from the terminal is placed in buffer B1 and analyzed by CONTROL to distinguish between source and command entries. When a command is detected it is interpreted and executed by CONTROL. For a source line SYNA is called to conditionally compile a statement. SYNA begins by checkpointing, as described in Section F.1 below, and then calls the lexical analyzer (LEXI) to pass the tokenized symbols from B1, one at a time. If SYNA successfully completes a statement (usually by processing a semi-colon) before LEXI exhausts the supply of tokens, SYNA returns to CONTROL. However, if SYNA's requests for symbols run LEXI off the end of B1, then LEXI 'hangs up' and gives a 'continue' call upon CONTROL. CONTROL invokes a continuation prompt, obtains another line from the terminal and returns to LEXI. LEXI can then supply the next symbol to SYNA and the process continues until eventually SYNA completes a statement and returns to CONTROL. Hence SYNA remains completely statement-oriented and is unaware of the occasions when LEXI and CONTROL must collaborate to replenish B1.

When SYNA has completed processing of a statement CONTROL tests the error stack to determine whether error messages and reconstruction are required. If no errors have been detected the prompter is invoked for the next statement. If errors have been detected CONTROL invokes the message writer and the decompiler to initiate the error dialog with the user. CONTROL must elect which of two statement entries to SYNA to employ for each statement. The 'confirm' entry confirms the compilation of the previous statement by taking a new checkpoint, and then compiles the new statement. The 'delete' entry deletes the previous statement by restoring to the previous checkpoint, and then compiles the next statement (which is presumably a replacement for the previous one.)

Calls to SFM to preserve the source must be synchronized with this process. CONTROL calls SFM to copy a line segment from B1, and indicates whether or not this segment represents the beginning of a statement. When errors are detected in a statement, as the beta-code is decompiled into B2 SFM is called to accept this text in place of the original source for that statement. If the user then submits another version of the statement, this text from B1 replaces the decompiled beta-code text in the source file.

When an edit command is received that refers to a line segment that is not part of the most recent statement CONTROL begins the recompilation process. The retrieval side of SFM is called to restart and CONTROL uses the checkpoint returned by SFM to restore the compiler. SFM is then called to deliver the source lines from that checkpoint up to the statement specified by the edit command. These lines are placed in B1 just as lines that arrive from the terminal and, in fact, LEXI and SYNA do not know the source of the contents of B1. When the proper time comes CONTROL ceases retrieval from SFM and obtains a new line from the terminal.

The 'program entry phase' of Figure 4 terminates whenever the user gives a %GENERATE or %RUN command. In either case CONTROL loads and executes the semantic analysis and code generation phases of the compiler. These phases differ from the corresponding ones of batch PL/C only in that certain of the error messages return to the terminal to have the user supply a missing value -- such as an array dimension -- for which a default value would automatically be supplied under batch PL/C. The %RUN command will cause CONTROL to load the execution supervisor and begin execution of the compiled program. At any time -- even during execution of his program -- the user may enter an edit command. This causes CONTROL to restore the program entry phase and execute the edit command just as if the system had been in program entry phase all along. This is commendably flexible, but it is also costly since a recompilation will take place. (Neither the semantic analysis nor code generation phase has any checkpointing capability and a complete repetition of those phases is required if the user makes any change in the source program.) The user must understand that an edit command given during execution will cause the entire program to be recompiled and execution to begin again from the beginning of the program. The one feature of an interpretive system that I-PL/C lacks is the ability to significantly alter the source program <u>during execution</u> and then <u>resume execution</u> from the point where it was suspended.

The %LIST, %SAVE and %SUBMIT commands are essentially independent of the compilation process. These may be given at any time and after their execution I-PL/C remains in exactly the same state as before their entry.

The treatment of the *DATA control card under I-PL/C is interesting. Recall that under batch PL/C this card signals the end of the source program and invokes semantic analysis, code generation and execution. The treatment under I-PL/C is quite different. The *DATA line is entered in the source file -- so that it will be passed to batch PL/C if a %SUBMIT command is given. CONTROL notes that a *DATA has been received (it will not accept another) and notes the line segment number that it has been given in the source file. I-PL/C continues in program entry mode after receipt of a *DATA and enters lines received in the source file, but no longer calls upon SYNA to process those lines. Data lines can be edited in the source file just like source statement lines but editing of a line segment whose number is not less than the number assigned to the *DATA line does not invoke recompilation. A further distinction accorded to the data lines in the source file is that they can be edited during program execution without requiring a restart from the beginning of the program. After editing of data lines execution can resume where it left off.

Under I-PL/C the system returns to the terminal for input for any GET statement that has either no file specification or FILE(TERMINAL). Since the user may enter data from the terminal when called by execution of the program, the only reason for

ntering it before the %RUN command would be if he expects to xecute the program more than once (and wants to avoid having to eenter certain data) or if he intends to %SUBMIT the program to atch PL/C. Under I-PL/C the execution supervisor will draw pon data in the source file to satisfy GET statements, and eturn to the terminal for data only when the source file data s exhausted.

.1 Conditional Compilation Of The Last Statement

Since it is assumed that a very large majority of editing ommands will affect only the most recently entered statement rovision is made to handle this task in an especially efficient anner. This involves both SFM and SYNA. SFM contains a pecial mechanism to replace the most recently entered line with text from either the decompiler or the terminal.) SYNA is odified to provide 'conditional compilation'. As noted above YNA has two different statement entries. The 'confirm' entry ccepts the previous statement and proceeds to the next. The delete' entry deletes the previous statement by overwriting it ith the next.

There are two basic strategies that might be employed for onditional compilation. The first would be the literal mplementation -- modify SYNA so that it never writes directly n beta-code, parse stacks or the symbol table, but makes onditional entries in a special workspace. When the signal is eceived to confirm a statement these conditional entries would e transcribed into the permanent areas. The second strategy ould be to simply checkpoint the entire workspace (the compiler rocedures are reentrant) at the beginning of each statement. onfirmation would mean to ignore this checkpoint and proceed to he next statement; deletion would mean to restore the workspace o the state that existed prior to the compilation of the last tatement by copying the checkpoint back into the workspace. he first strategy has the disadvantages of requiring numerous nd significant changes to SYNA, and of adding the overhead to opy from the conditional workspace for the correct (confirmed) tatements, which hopefully are more numerous than the incorrect deleted) ones. The second strategy has the disadvantage of equiring a considerable area to be copied for every statement.

It is argued below that a composite strategy should be mployed, involving modest changes to SYNA in order to reduce he area that must be checkpointed. Before discussing this it s informative to attempt to roughly estimate the performance egradation that might result from the use of the simple heckpoint strategy. Suppose that an average of 12K must be aved and that a second core area is available so that auxiliary torage does not have to be used. On a 360/65 with 64 byte emory access this would require 384 accesses to read and write, r something like 300 microseconds. Since this same machine ould be expected to process approximately 20,000 source tatements per minute through the lexical-syntactic phase of

PL/C each statement takes about 3 milliseconds. Hence even this crude strategy would increase compilation time by only 10 per cent (and space by a similar amount.) We believe that even this penalty would be tolerable.

One obvious possibility to reduce the cost of checkpointing would be to reduce the frequency at which checkpoints are taken. One could imagine checkpointing only once every n statements, and recompiling from that checkpoint whenever a replacement is required within the next n statements. The prospects can be analyzed as follows:

let:

tc be the time to perform a checkpoint operation
ts be the average time to compile a statement
n be the number of statements between checkpoints
m be the average number of statements between errors.

Ignoring the time to recompile the statements that are in error (which must be done in any event) the time required to checkpoint every statement of a program of S statements is simply

$$(S)tc$$

If the checkpoint is periodic and if one assumes that an error is equally likely to occur at each point in the checkpoint cycle, then on the average there will be n/2 statements that have to be recompiled each time an error occurs. The approximate time for the checkpointing and recompilation in a program of S statements is

$$(S/n)tc + (S/m)(n/2)ts$$

It is quickly apparent that if ts is an order of magnitude larger than tc (as is suggested in the previous paragraph) no reasonable pair of values for n and m will make periodic checkpointing profitable. The conclusion is that one should checkpoint for each statement, and concentrate effort on making tc as small as possible.

There are three different work areas in PL/C and they have different checkpoint characteristics. The areas are the symbol table, beta-code and the parse stacks.

Some beta-code is written for each statement, so this area is changed, but it has two convenient properties:

1. Except for special cases arising in the 'else' and 'continue' statements of an IF construction, and the beginning of an ON-unit it is written in strict left-to-right order,
2. Beta-code does not assume that the space being used has been initialized to any particular value.

The IF and ON special cases involve the setting of pointers in the beta-code for previous statements. Except for this one can

ffectively checkpoint beta-code simply by saving a pointer to
e position of the beginning of the statement. To delete a
tatement one simply restores this pointer and overwrites the
tatement. The IF and ON pointers will require special
reatment, but this does not require extensive nor difficult
anges in SYNA. Eliminating beta-code from the area to be
eckpointed roughly halves its size.

There are four parse stacks. Two of these are completely
ternal to a statement and carry no information over a
tatement boundary. They obviously require no checkpointing.
e other two stacks are concerned with the static IF and block
ructure. Although together these stacks represent only 336
tes it appears that with modest changes to SYNA both can be
eckpointed by saving only their top levels (8 and 32 bytes,
espectively) rather than the entire stack.

The symbol table presents more of a problem. Although it
ows 'downward' in core each new entry is chained to previous
tries depending upon its name, attributes and locale, hence
anges may be made throughout the table. One or both of two
ssibilities will be explored. First, not all statements make
y change at all in the symbol table and it may be possible to
ppress checkpointing altogether if no symbol table change has
en made since the last checkpoint. Second, while changes are
spersed they do not affect large areas and it may be possible
build a table of affected areas and checkpoint very
lectively.

In summary it appears that the parse stacks and beta-code
n be checkpointed by saving at most a few hundred bytes if
me modest changes are made in SYNA. Hence the checkpointing
oblem really centers on the symbol table. It would not be
ohibitively costly to save this entire area for each
tatement, but it may be practical to be selective in both the
ea to be saved and the frequency with which it is done. In
y event it seems unlikely that conditional compilation will
sult in more than a 5 per cent degradation in the lexical-
ntactic performance of PL/C.

2 Invisible Recompilation

When the user is permitted to make an arbitrary change in
ne segment nn of the source program, after having proceeded
ll past nn in program entry, the following situation obtains:

1. No statement prior to the statement that includes line segment nn would be treated differently by the syntactic analyzer.

2. Any statement from the one including nn to the end of the program could be treated differently by SYNA.

erefore the I-PL/C response to any editing command for nn is

to begin a complete recompilation of the program starting with the statement that includes line segment nn. (Actually the recompilation starts at the last checkpoint before this statement but the difference is not conceptually important.) Having entered recompilation mode the progress of I-PL/C is controlled by the line segment numbers given by the user. The procedure is the following.

Let S(nn) stand for the statement containing line segment nn. Let IC stand for 'initial compilation' mode and RC stand for 'recompilation' mode.

1. At any time (either IC or RC), if the user has just entered nn and then edits mm, where mm <= nn and S(mm) = S(nn), the system invokes the 'delete' action of the conditional compilation process. (See Section F.1.)

2. At any time (either IC or RC), if the user has just entered nn and then edits mm, where S(mm) < S(nn), the system enters RC mode and recompiles up to (or through, depending upon the edit command) S(mm).

3. If in RC mode the system has just compiled S(mm) and the user enters an edit command referring to nn, where mm < nn, the system compiles up to (or through) nn, drawing from the source file all line segments whose numbers fall between mm and nn.

The recompilation process is designed to be invisible to the user. The only information that is displayed during this process is that which is a consequence of the source change that he has made. One problem in this regard arises in the fact that there are two different types of diagnostic messages produced by PL/C. The large majority of error messages are accompanied by a repair in beta-code and a corresponding change in the text stored in the source file. On recompilation these messages will not reappear because the condition that caused them has been eliminated by PL/C. The other type of error message is a 'warning' -- such as for an inaccessible statement, or an external-name-too-long -- and the beta-code and source text are not changed. Special provision is made in I-PL/C so that once given, such a warning will not be repeated for the same statement no matter how many times it undergoes recompilation. Of course either warnings or errors may be reported during recompilation for a statement that incurred neither during initial compilation, as a consequence of the change that has been made earlier in source.

While I-PL/C will permit random editing it will obviously be more efficient for a user who disciplines his editing so as to minimize the number of recompilations that are required. Users, except possibly for rank beginners, should be advised how to conduct their editing. Aggregate compilation performance will be heavily dependent upon the user's behavior. If all corrections are made in the conditional compilation mode the

increase in cpu time relative to batch PL/C should be negligible. If corrections can be accumulated and ordered so that a single repetition of syntactic analysis is required the compilation time will be increased by 50% (since lexical and syntactic analysis are approximately one-half of PL/C compilation.) Unfortunately there is almost no bound on how bad performance can become -- consider, for example, a user who makes numerous corrections and always on the second most recent statement. An even more pathological case would be a user who alternately entered new lines and edited the first line of his program. However we are hopeful that typical performance will not require more than two or three times the cpu time of batch PL/C.

F.3 Structure And Management Of The Source File

The source file in I-PL/C is designed to provide temporary storage (for the duration of the terminal session) of the source statements, control cards and data cards that are received by the compiler. This is done for three purposes:

1. So that the source program can be %SAVEd for later use.
2. So that the source program can be %SUBMITted to batch PL/C for processing.
3. So that the user can edit any line in the program at any time.

If the user never edits any but the most recently submitted statement, does not wish to save the program, and executes the program immediately under I-PL/C this file will never be used and the effort expended in its construction will have been wasted.

The information in the source file comes from three sources:

1. The principal source is the B1 buffer, which in turn has three sources: the terminal, the retrieval side of SFM, and the %COPY command.
2. Reconstructions of source statements are drawn from the B2 buffer.
3. Periodic checkpoints are taken automatically by SFM.

The source file consists of blocks of fixed length written sequentially on a direct access storage device. Each block is either a source block or a checkpoint block. A source block consists of a chain of line segment units and is constructed in a core buffer called the store buffer. Each line segment unit has seven components:

1. Line segment number
2. Pointer to the start of the next line segment unit (blank for the last line segment unit in a block)
3. New line indicator (LNI); =1 if this line segment began a new line in original form, =0 if not
4. Statement indicator (SI); =1 if this line segment begins a statement, =0 if not
5. Column position for beginning of text (relative to left

source margin)
6. Warning indicators; two bytes of flag bits for various warning messages that have already been given for this statement (filled in only if SI = 1)
7. Line segment text, including comments. The text is compressed by having leading and trailing blanks removed.

Each time that SFM is called to copy a line segment from B1 or B2 it creates a line segment unit. If there is not enough room for the complete line segment unit in the store buffer the buffer is written out to auxiliary storage and a new buffer is begun -- a line segment unit does not cross a block boundary.

On the first 'new statement call' for each source block SFM takes a checkpoint of the compiler workspace. (It is essential that this checkpoint be taken on a statement boundary and not just an arbitrary line segment boundary. This is a full copy of the workspace rather than the abbreviated version described in Section F.1) This will generally require several blocks, the number depending upon the size of the workspace at that point and the file block size being used. These checkpoint blocks are chained together--a fixed location at the beginning of each checkpoint block pointing to the next. The location of the beginning of the chain is stored in fixed location at the beginning of the new source block. (The first source block lacks a checkpoint chain and obviously does not need one.) This automatic checkpointing reduces the burden of recompilation somewhat by supplying a starting point later than the beginning of the program. While it was argued in the preceeding section that for the conditional compilation of the last statement complete checkpointing is always preferable to recompilation, that does not apply to the complete program. Since all checkpoints must be saved in this case auxiliary storage must be used and the auxiliary storage access time dominates the statement compilation time. Moreover the storage volume to checkpoint each statement, even in auxiliary storage, would be formidable.

A core-resident source block directory is maintained. For each block this directory gives the location of the block (in core or auxiliary storage) and the number of the first source statement (the first line segment with SI=1) in that block. The entire checkpointing and block-buffer management is of course completely internal to SFM and invisible to the rest of the system.

When the user specifies an editing command for a line segment other than one that is part of the last statement entered the retrieval function of SFM is invoked. This involves two stages. The first is a restart phase. SFM is called with a particular line segment number as argument. SFM first writes the current incomplete source block into auxiliary storage and logs it into the source block directory just as if it had been completed normally. Then a search of the source block directory determines which source block contains the specified line segment, and that source block is copied into a second source

lock buffer called the retrieve buffer. (If the line segment appens to be a continuation segment at the beginning of a block he process must start with the previous block.) The location f the beginning of the checkpoint chain is copied from the etrieve buffer into the store buffer. All of the initial continuation' line segments (segments for which SI=0) are also opied from the retrieve to the store buffer. Having done all his SFM returns with the location of the beginning of the heckpoint chain from the retrieve buffer. If the checkpoint is ull CONTROL completely initializes the compiler; otherwise it s restored to the state given by the checkpoint returned from FM.

The second phase of retrieval is recompilation from the heckpoint up to the region of the line segment specified in the dit command. In order to accomplish this CONTROL makes a equence of calls upon SFM, each asking for the next line egment from the retrieve buffer. The first line segment eturned is the first one in the retrieve buffer for which SI=0 - that is, the beginning of the statement immediately after the heckpoint was taken. SFM will automatically move the next ource block to the retrieve buffer as necessary. Eventually FM will signal CONTROL that the end of the source file has been eached -- if CONTROL has not ceased calling SFM before this ccurs. These line segments are copied into the B1 buffer and outed through the lexical-syntactic phase of the compiler ractly as if they were newly arriving from the terminal. No odules except CONTROL and SFM are aware of the change of mode. particular these line segments are routed back to the storage ide of SFM to be stored in a new version of the source file. ie storage and retrieval functions of SFM are completely idependent; the storage function proceeds as described eviously building and saving source blocks, regardless of ether the input in B1 originated at the terminal or in the trieval side of SFM. The retrieval function responds to peated calls for the next line segment, copying blocks from ixiliary storage as required. The only interaction between the o functions of SFM is in the management of auxiliary storage ace. Once a source block has been copied back into the trieve buffer its block and its associated checkpoint blocks auxiliary storage are made available to the storage function SFM. SFM writes new blocks sequentially (so no pre- rmatting of the space is required) but will reuse blocks leased by the retrieval function to reduce the total auxiliary orage space required.

Note that the %REPLACE and %DELETE logic is in CONTROL and t in SFM. SFM does nothing more than store and retrieve idividual line segments and manage the file that holds these ne segments. At any time CONTROL can revert to terminal input mply by ceasing to call for further input from SFM. It can so repeat the restart process at any time, destroying the ntents of the retrieve buffer and beginning recompilation from different point.

Note also that the source file is line segment oriented so that the user can edit individual segments. Sufficient information is preserved with the segment to enable the system to restore it to its original form. It is, of course, possible for the user to replace an original line segment with a longer one. This presents no problem in managing the source file (which is rewritten during recompilation) but the output formatting routines must be prepared to eject to a new line when the length, column position, and NLI of successive line segments are incompatible.

The I-PL/C user has two different ways of including the contents of a pds in his program. If he gives a *INCLUDE control card the contents of the pds are passed to the system and appear in beta-code, just as under batch PL/C. The *INCLUDE card itself is placed in the source file -- but the contents of the pds do not go into the source file. This means that the individual lines of this included material cannot be edited under I-PL/C. (They can presumably be edited in the pds by other utilities.) If a %SUBMIT command is subsequently given the *INCLUDE control card is passed to batch PL/C as part of the source file and is expanded in the normal manner by batch PL/C.

Alternatively the %COPY command can be used. %COPY also passes the contents of the pds to I-PL/C and beta-code but in this case the contents of the pds are included in the source file and may be subsequently %LISTed and edited. If a %SUBMIT command is subsequently given the included material is passed to batch PL/C as part of the source file.

If an error is encountered in processing the input from either a %COPY command or a *INCLUDE control card the processing is interrupted to execute the usual error dialog. When the error question is resolved the user may resume the inclusion (from the point where it was suspended) by repeating the original %COPY or *INCLUDE without giving a pds-name. (If the name is repeated the inclusion will begin again from the beginning of the pds.) For example:

```
. . .

850 END; eol

860 *PROCESS eol

870 %COPY TRAX eol

1130 X = (Y + Z * W ;

IN 1130 ERROR SY04 MISSING )

PL/C REPLACES 1130 WITH X = (Y + Z * W);

1140 %REP 1130 eol
```

1130 X = (Y + Z) * W; eol

1140 %COPY eol

1270 . . .

The %LIST, %SAVE and %SUBMIT commands are also implemented in SFM. In the case of a %LIST the source block containing the specified line segment is located through the source block directory. To avoid requiring a third block buffer the %LIST process shares the retrieve buffer. If the retrieve buffer is in use in a recompilation when %LIST is invoked the retrieve buffer is copied out to auxiliary storage and restored when the %LIST processing has been completed. The %SAVE and %SUBMIT commands are similar except that they always process the entire source file and route the images to editing routines that put the line segments into appropriate form for pds storage and batch PL/C processing, respectively.

The average line segment unit in the source file is likely to be approximately forty bytes long. If a 7000 byte buffer is employed (a full track for a 2314) this will accomodate some 180 line segments per block. This would mean that for many student programs the complete source file would be less than one block and auxiliary storage would not be used. Even for large programs there would be a relatively small number of blocks required and auxiliary storage traffic would be very modest. On the other hand, since checkpointing is related to this block size there would be relatively large amounts of recompilation. It may well turn out that it is preferable to use a significantly smaller block size (an installation option) and accept the increased auxiliary storage traffic in order to reduce the actual amount of recompilation required.

A summary of the calls on SFM is given below.

'UTLINE(nn,b,p1,p2,LNI) - where nn is the line segment number;), p1 and p2 specify the line segment (buffer, left and right character positions, respectively); and LNI is 1 or 0 depending upon whether the line segment does or does not begin a new line. Action is to create a new line segment unit in the store buffer.

STATEMENT. Action is to note that the next line segment received will start a new statement (and hence should have SI=1.)

)ELETE. Action is to delete the line segments for the last statement -- that is, back to and including the most recent line segment unit with SI=1.

INITIALIZE. Action is to clear the source block directory and prepare the store buffer to begin a new program.

EDIT(nn) -- where nn is a line segment number. Action is to immediately 'complete' the current source block; locate the

source block containing the specified line segment by search of the source block directory; copy the required source block into the retrieve buffer; copy the initial continuation line segments, if any, from the retrieve buffer to the store buffer; copy the checkpoint chain pointer from the retrieve buffer to the store buffer; return the value of the checkpoint chain pointer.

GETLINE. Action is to copy the next line segment from the retrieve buffer into B1.

START_LIST(nn) -- where nn is a line segment number. Action is to save the retrieve buffer if it is in use; locate the proper source block by search of the source block directory; move the source block into the retrieve buffer; copy line segment nn from the retrieve buffer to B2.

LIST. Action is to copy the next line segment from the retrieve buffer to B2.

STOP_LIST. Action is to restore the retrieve buffer to the state that existed before START_LIST.

F.4 The Execution Phase Of I-PL/C

The modifications to the execution phase required to accomplish these tasks appear to be less severe and more easily isolated than those for the program entry phase. Most of these changes can be concentrated in a single module denoted as EXECUTION COMMAND in Figure 3. This module will be called by CONTROL when it appears likely that an execution command has been received. EXECUTION COMMAND is responsible for scanning and parsing, interpreting and excuting these commands.

F.4.1 Return To The Terminal

There are six ways by which control is returned to the user at the terminal during the execution of a program under I-PL/C:

1. HALT statements included in the source program.
2. HALT statements inserted in the object code during execution by the %HALT command.
3. Attention interrupts from the terminal.
4. GET statements (after any data in the source file has been exhausted.)
5. Error conditions discovered during execution.
6. Completion of execution (caused by encountering a STOP statement in the source program or returning from the MAIN procedure.)

On every return to the terminal the system will specify what caused the return and exactly where (in terms of source statement numbers) the program is in execution:

HALT_AT_nn

HALT_ON_CALL_AT_nn

HALT_ON_FLOW_AT_nn

HALT_ON_CHECK_AT_nn

INTR_AT_nn

GET_nn

IN_STMT_nn_PROGRAM_IS_STOPPED

IN_STMT_nn_PROGRAM_RETURNS_FROM_MAIN_PROCEDURE

The obvious method of handling the HALT source statement ould be to have the I-PL/C code generator compile a call to the erminal routine for this statement. (The code generator for atch PL/C will treat it as a null statement.) However, it is robably preferable to have the code generator in I-PL/C also reat HALT as a null statement, but make provision so that at he beginning of execution a HALT is inserted in front of this ull statement just as if the user had specified this from the erminal with a '%HALT AT nn' command. The virtue of mplementing the source HALT in this way would be that the user ould then negate its effect with a %NOHALT command from the erminal. He is inevitably going to want to do this, and it eems unreasonable for I-PL/C to maintain that there is an mportant difference between a source HALT and a HALT resulting rom a terminal command.

There are five different forms of the %HALT command. Three f these -- %HALT ON FLOW, %HALT ON CHECK and %HALT ON CALL -- re very simply implemented. Three different 'terminal return lags' will be created for the three cases and the EXECUTION UPERVISOR will be modified to interrogate the appropriate flag ach time that the FLOW or CHECK condition is raised, or a CALL tatement is executed. The appropriate flag is set by the %HALT ommand and cleared by the corresponding %NOHALT command.

There are several ways that the '%HALT AT nn' command could e implemented. An interpreter would simply maintain a table of halt locations' and call a routine that would search this table t the beginning of execution for each source statement. lternatively, a compilation strategy, which will take more time o execute the %HALT command but will not slow the execution of he object program, would be to actually insert a HALT in the bject program at the specified location. PL/C provides a articularly convenient way of doing this. At the beginning of he object code for each source statement (in either batch PL/C r I-PL/C) there are a pair of 'move immediate' (MVI) nstructions that move the source statement number for the tatement about to be executed into a fixed location. (This ermits the error message writer to report the source statement

number whenever an error is encountered in execution.) The eight bytes in object code for these two instructions for a particular statement number are recognizable so that the beginning of a particular statement can be found by a simple scan of the object code and it is not necessary to create a table of statement starting points during code generation. When the beginning of the appropriate statement is located by the %HALT routine one of the MVI instructions is changed to an invalid operation code. The routine that handles the hardware 'operation exception' when this instruction is encountered will recognize the particular invalid code as a %HALT code, determine the statement number from the argument of the invalid instruction (and the argument of its successor), and return to the terminal with an appropriate message. A 'halting location' table will be maintained so that the '%NOHALT AT nn' command can be implemented by searching this table rather than scanning the object code in order to find the proper location to restore the invalid operation code to an MVI.

While the '%HALT AT ALL' command could be implemented in essentially the same way as '%HALT AT nn' there is a much more efficient method. The object code for each source statement in PL/C includes a 'branch and link' (BALR) instruction that calls a routine in CONTROL. In batch PL/C this is used to enforce the time limit. Normally this routine consists of nothing but an immediate return to the object code, but when time limit has expired this return is replaced by a branch to a termination routine. In I-PL/C this routine will be generalized to handle the '%HALT AT ALL' command and the attention interrupt as well as the enforcement of time limits. The execution of the '%HALT AT ALL' command will alter this control routine to cause each source statement to call the 'return to terminal' routine. Similarly the attention interrupt will alter the control routine to interrupt the execution of the object program at the beginning of the next source statement and return to the terminal.

The principal problem in implementation of the GET return is to anticipate the possibility that once the user is in control he will elect not to supply the requested data but will enter some other command. Some commands -- such as %LIST, %DISPLAY, %SET or %HALT AT nn -- represent temporary diversions, and when their execution is completed the system should repeat the announcement GET nn and try again to obtain the data necessary to continue execution. However, other commands -- such as %GOTO label, or %REPLACE nn -- either redirect execution or terminate execution altogether. In anticipation of this all of the I-PL/C data service routines must be constructed so that they can be aborted during execution without harmful side-effects.

The execution error routine will be modified so that any error that requires a repair in order to continue execution will invoke a return to the terminal. In this way the user will be offered the opportunity to make the required repair. To do so

will use the normal I-PL/C commands -- no special controlled alog is provided. In most cases the %SET command will be ployed. If the user does not make a change and simply gives a ll command (eol) I-PL/C will effect the same repair as batch /C and continue execution. If the user does make a change its equacy is tested and either the error message is repeated or ecution is continued. As in the case of the GET return, the ror routine must be prepared to be aborted harmlessly since e user's response to the error may well be to redirect the ogram (with a %GOTO command) or to re-enter the program entry de by giving an edit command. The error return to the rminal will permit I-PL/C execution to survive some errors at are immediately fatal in batch PL/C -- such as a GOTO atement with an invalid label. Not all error conditions ring execution will include a return to the terminal. For ample, when a statement is deleted by the compiler during ntactic analysis a special null statement is placed in beta-de. A message is produced each time that such a statement is countered during execution. This error does not automatically turn to the terminal since the user was advised of the tuation during program entry and there is essentially nothing at he can do during execution. He can, of course, use the HALT AT nn' command to cause a 'deleted statement' to return the terminal.

The return at the end of execution requires no special plementation. The normal completion message is routed to the rminal followed by the compile and execution time statistics. e post-mortem dump is not produced automatically under I-PL/C. less the user is running under a NODUMP option the system will nounce the coming of the dump and offer the user the option to oid it by entering another command:

PM_DUMP_WILL_FOLLOW

null command (eol) will start the dump; any other command will oid the dump. Although any command may be entered at this int many of them are patently pointless since execution cannot resumed (except by restarting from the beginning with %RUN.)

4.2 Immediate_Execution_Commands

The immediate execution commands of I-PL/C are functionally milar to some of the PL/C source statements. These commands ffer from their source statement counterparts in that they are tended to be executed only once, at the time of entry, and do t become part of the stored program for possible repeated ecution. The commands are also more restrictive than the rresponding source statement. Only elements such as those at could appear in a GET DATA statement can appear on the left de of the %SET command (the immediate assignment command) and ly constants can appear on the right side; %DISPLAY (PUT LIST) rmits only scalar variables (no expressions); %FLOW and %CHECK nnot have parameters. These restrictions have been imposed

simply to reduce the magnitude of the initial implementation task. In principle one could have the entire PL/I source language available in this mode.

The execution of the %FLOW, %NOFLOW, %CHECK, %NOCHECK, %PUT ON, %PUT OFF, %PUT SYSTEM ON and %PUT SYSTEM OFF commands simply involves setting and clearing the same control flags that their source statement counterparts would alter when encountered during program execution.

The execution of the %GOTO command is also quite similar to that of the compiled GOTO statement. Under PL/C the GOTO is executed semi-interpretively, always going through the symbol table (to permit label frequency counting) and this same mechanism can be invoked for the %GOTO command.

The implementation of %DISPLAY and %SET draws heavily upon the existing input/output modules of PL/C. Because of the necessity of interpreting data-directed input the input/output service routines of any PL/I compiler are necessarily semi-interpretive and readily adaptable to interactive mode.

F.4.3 Execution Efficiency

Execution of a program under I-PL/C will certainly take longer in real time than under batch PL/C unless there are no returns to the terminal whatever. Execution will also require some modest additional cpu time to execute whatever commands may be given from the terminal. However, except for this there is no reason to believe that cpu requirements for execution under I-PL/C will be any different from requirements under batch PL/C. That is, program execution time will still be roughly comparable to that obtained under the IBM PL/I-F compiler -- ranging from slightly less to twice as much, depending on the type of program and the specific options employed.

G. Possible Further Development Of I-PL/C

If I-PL/C ever reaches the state described in this proposal there are some interesting and useful extensions that could be made.

The most intriguing would probably be the implementation of the generalized ON-condition and the controllable reversing of the direction of execution. Both of these features have been implemented in an experimental version of batch PL/C, (see reference 2) but they both make more sense in an interactive system.

A very practical and useful extension would be to add a private storage facility to save programs between sessions. In theory the %SAVE and %COPY commands will provide this ability, but in practice the management of pds's for a large number of

sers is almost prohibitive. In fact, it may well be that a ystem such as I-PL/C is not generally useful unless such a rivate library facility is provided -- which is a sad ommentary on an important aspect of OS data management.

eferences

onway, R. W., H. L. Morgan, R. A. Wagner and T. R. Wilcox: 'PL/C- The Cornell Compiler For PL/I; User's Guide To Release 6', Dept. of Computer Science, Cornell University, August 1971

onway, R. W. and T. R. Wilcox: 'Design And Implementation Of A Diagnostic Compiler For PL/I', Research Report 71-107, Dept. of Computer Science, Cornell University, September 1971