COST REDUCTION OF STUDENT PROGRAMMING EXERCISES

by

Paul Gervase Gregory III

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
Massachusetts Institute of Technology
June, 1975

Signature of Author..........................---------------------
Department of Electrical\Engineering, June, 1975

Certified by...............                  Advisor

Accepted by..........                      Chairman, Departmental Committee on Theses
CONTENTS

Acknowledgments 4

Abstract 5

Introduction 6

Definition of Costs 7

Definition of Student Running Process 10

Possibility of Cost Reduction 12

Batch Monitor Concept 15

Evaluation of Batch Monitor Approach 17

Loop Prevention 21

A Cheaper PL/1 (F) Compiler 36

Evaluation of Cheaper PL/1 (F) Compiler 39

Appendix A: User's Guide for Batch Monitor 42

Appendix B: User's Guide for Cheaper Compiler 52
LIST OF TABLES AND FIGURES

Fixed Overhead for Batch Monitor at MIT 20
Spooling Overhead for Batch Monitor at MIT 20
PL/1 Governing Procedure Source 34
Performance Data for Cheaper Compiler 40
JCL for Running the Monitor with PL/1 48
ACKNOWLEDGMENTS

The author wishes to express his gratitude to the many individuals who made this work both possible and complete by their assistance and encouragement. In particular, Dick Caloggero of the Electrical Engineering department and Dick Steinberg of the Information Processing Center for providing the necessary development funds; George Newton of the Electrical Engineering department for providing me with the unique opportunity to actively participate in the teaching process; John Donovan and Suzan Kruger for letting me use their students as guinea pigs; the numerous students of 6.251 and 6.083 who were the guinea pigs; Rick Granger for his performance measurements of my PL/1 (F) compiler modifications; Stu Madnick for providing the necessary constant pressure; and Jack Aiello for the moral support that enabled me to finish.
ABSTRACT

COST REDUCTION OF STUDENT PROGRAMMING EXERCISES

by

Paul Gervase Gregory III

supervised by

Stuart E. Madnick

Assistant Professor of Management Science

This thesis explores the process by which student programming exercises are processed in a batch environment on IBM 360 or 370 computer systems. Presented are three programs which may be used to effect a significant reduction in the costs associated with this running process.

These programs are written in such a fashion as to be generally applicable to a wide variety of student exercises. They have been thoroughly tested by two years of use in an actual teaching environment.

Use of these programs by a course in the Electrical Engineering department resulted in a savings of fifteen dollars per student, for a total savings of $3,000 in one academic year.
The thorough understanding of a contemporary programming language and proficiency in its use is an invaluable tool for a student of computer science. To this end time is usually devoted towards teaching the use of one or more languages in computer science programs of study. The mastery of a programming language is similar to that of a sport like tennis. Although some people have natural ability and agility, almost no one learns to play merely by reading a book.

A person's style and form benefit from coaching, but there is no substitute for time on the court. As in tennis a balance exists between in class presentation of style and technique and the out of class development of basic programming skills. The normal approach to proficiency building involves the assignment of progressively harder programming exercises which the student must code and test. The object of these exercises is generally not to teach the details of a particular language's implementation or a machine's structure, but rather to instill an appreciation for the use of the language to solve problems.

Programming exercises necessarily require the use of an actual computer by the students. Associated with this use is a cost which is nonconsequential and must therefore be considered in course planning. This thesis deals with the cost of machine exercises for computer courses, first identifying the costs and then outlining steps which may be taken to reduce them without
sacrificing educational quality. This thesis is a direct outgrowth of the author's participation in the process of teaching programming in the Electrical Engineering department at MIT.

Presented are three programs designed to expedite the efficient testing of a large number of student programs in a non-interactive batch environment on an IBM 360 or 370. Use of these programs by the Electrical Engineering course for which they were designed resulted in a $15 reduction of the per student computer costs, producing a savings of $3,000 for the academic year (discounting development costs). The total development costs were $1,500 borne by the Electrical Engineering department and $2,000 borne by the Information Processing Center. These programs are currently used by the department of Management Science at MIT.

DEFINITION OF COSTS

Processing student computer assignments obviously requires certain resources with associated opportunity costs. Even if a course does not directly pay for its computer usage, surely someone pays. The costs incurred may be broken down into three crude classes for purposes of discussion:

1. Student Preparation Costs—those costs directly associated with converting a student's program and data into a computer edible format, excluding any connected
computer costs. This may include key punches, card stock, remote job entry stations and similar equipment, as well as supervision and maintenance of these facilities.

2. Computing Costs - those costs directly associated with the actual computing involved. This includes reading in the program, running the program, and finally outputing the results. Lumped into this category is machine rental and maintenance, operator salaries, energy costs (electricity and air conditioning), space costs such as janitorial services, and basic systems support.

3. Course Development and Support Costs - those costs made necessary by using a computer in a course, that are external to the actual computing costs. These include preparing assignments, grading the assignments, helping students with assignments, and overseeing the smooth operation of the student running process.

This thesis does not deal with the costs of class one, taking them to be inherent in the process. Similarly, the costs of class three are barely mentioned, the only gains being from increased efficiency of the student running process reducing staff requirements. In order to effectively discuss the second class of costs, I have assumed that the course in question does not own the computer involved, rather is a user of a centralized computer facility.
For purposes of usage monitoring and billing, a person's use of a computer facility is usually divided into the following six categories:

1. **CPU usage**—a direct measure of time spent by the computer actually executing machine instructions for the user. This is expressed in time units (seconds) and includes operating system execution overhead.

2. **Input and Output (I/O) usage**—a direct measure of the number of I/O requests issued by a user's program. This category usually includes only disk or tape I/O requests.

3. **Main Storage Usage**—an imputed quantity proportional to the opportunity cost of dedicating a fraction of the machine's real memory to a user. That is, one user executing "prevents" other users. This is calculated by adding a fixed time quantity for each I/O request to the basic CPU time from (1). Then this quantity is multiplied by the size of real memory used. This is often scaled by an exponentially increasing function of memory size. This penalizes I/O bound programs which use very little CPU time, but tie up memory for a long time; and large programs which tie up much memory.

4. **Unit Record Equipment Usage**—a direct measure of the number of cards read and lines printed by a program.
5. Secondary Storage Usage- a direct measure of the secondary (disk) storage used by a programmer. This storage is used to hold canned programs and data accessed by students in execution. This is expressed in units of (space used * time).

6. Fixed Job Fees- fixed charges levied against a job, and its job steps. These fees allegedly cover operator handling of the deck, the system overhead of interpreting a job's JCL statements, and other overhead items. This charge is computed by multiplying the number of job steps by a rate and then adding a flat job charge.

This complex system of separate usage charges is designed to accurately charge a user for his or her total computer usage. By covering many categories, special cases such as I/O bound, CPU bound, large versus small, etc. are all fairly billed. A direct consequence of this scheme is that if one or more usage categories can be reduced, while holding others fixed, certainly the total computing resources used and subsequent billing will be less.

DEFINITION OF STUDENT RUNNING PROCESS

A typical programming exercise consists of the functional description of a program which the student is to write. The student prepares an input deck containing his or her program.
This program is then tested and the results examined to determine a grade.

Partially to protect the student from the system and partially to protect the system from the student, a student's prepared program (deck) is generally entered into the computer system by a course employee, who has knowledge of Job Control Language and such things. As far as the student is concerned, he or she prepares a program, leaves the deck somewhere, and comes back later for the results. While keeping the student somewhat in the dark about the details of actual job submission procedures, it does free the course to address itself to only the pure, system independent aspects of programming.

Once a student's program is prepared, it must be tested. This testing process involves the following four steps, which must be performed for each program:

1. Input the student's program. This entails reading in the deck which the student has prepared.

2. Translate the student's program into machine language (via a compiler or assembler).

3. If no errors were detected during translation, bind the program with any external subroutines it may require and initiate its execution.

4. Print the output from translation and (if any)
In IBM terminology, a job step is, "...that unit of work associated with one processing program." (1) The language translators written by IBM accept as input a source program and output machine language when they are invoked. No further processing is performed. The loader accepts this machine language, loads it into memory, and executes it.

Since each job step may specify only one major program to be executed, two job steps are required to test a student's program. One to translate the program and another to execute it. Thus, if the straightforward and obvious method were used for testing n students, 2n job steps would be required.

POSSIBILITY OF COST REDUCTION

The charge associated with reading a student's deck and printing his or her output falls into category four, unit record equipment usage. The amount of output a program produces would seem to be an inherent quantity, just like the size of the program deck. For this reason, the costs associated with steps one and four of the student running process are not amenable to reduction. Certainly, if a particular translator could be made more efficient, the costs of step two could be reduced, this will

be discussed in detail later. There is a simple efficiency improvement with attendant cost reduction which may be realized from examining the microstructure of steps two and three.

That component of the IBM operating system which reads Job Control Language (JCL) statements and starts each job step is called an initiator. Initiators perform the following ten functions for each job step:

1. Initialize the user's memory space.

2. Interpret all JCL statements and build appropriate system control blocks.

3. Decide if the current step should be executed, based on results from the previous job step's execution (the condition code facility).

4. Allocate any I/O devices and temporary datasets required by the job step.

5. Determine the particular program a user wishes to execute and locate it in the file system. Indicate an error if it cannot be found.

6. Load the desired program into memory, indicating an error if the available memory space is insufficient.

7. Begin execution of the desired program.
8. Interpret any system error conditions encountered by the program.

9. When the program has either normally or abnormally terminated, release any temporary resources such as datasets or I/O devices assigned to the step.

10. Release all system control blocks allocated for the job step.

This processing is necessarily complex, since a job step may specify the execution of virtually any type of program, with a broad spectrum of possible requirements. It should be noted that the needs of the student running process are much simpler than the general case. There are only two programs we are concerned with—the translator and the loader. These programs have very well defined resource requirements in terms of I/O devices, scratch files, memory, etc. Clearly, Initiator functions one through five, nine, and ten need only be performed once for the entire student running process. Similarly, we only need to locate the two programs in the file system once. Thus, it should be possible to design a program, which invoked by one job step, would repeatedly perform Initiator functions six through eight for the students. Due to the strictly defined nature of the programs to be executed (translator and loader) it should be possible to make this special purpose program more efficient at performing these steps than the Initiator.
BATCH MONITOR CONCEPT

The program mentioned above is called a batch monitor. The basic idea is to eliminate repetitious overhead normally incurred by each student, by incurring it once for all students run in a batch. The monitor adds its own overhead to the execution of each student, as discussed below, but job step initiation overhead is only incurred once. Assuming the overhead added by the monitor is lower than step initiation overhead, this approach lowers the marginal cost of running a student program.

The fact that the monitor incurs one-time initialization overhead, which is not initiator related, may mean that the average cost of a student program will not be lower than the no-monitor cost until several student jobs are run together and the lower marginal cost brings down the average cost. This "break even" point and its determination will be discussed later.

The batch monitor is executed by one job step, which contains all the file definitions (JCL) for the translator and loader. For input, the monitor takes a stream or "batch" of student program decks, separated by special delimiter cards. After its execution is initiated, the monitor performs the following processing:

1. Interpretation of parameters entered from the JCL. These parameters control the optional features of the monitor, such as logging of students and their results,
and convey required information like the names of the translator and loader programs.

2. Location of the translator and loader programs in the file system. This involves use of the RLDL macro.

3. Spooling of a student's deck. If the input file is empty, proceed to step eight, else copy cards from the monitor input file to a disk scratch file, until a terminator is reached, or there are no more cards. At this point the scratch file contains one complete student program.

4. Invocation of the translator. The specified translator is invoked, passing it the program collected in the scratch file. Output from the translator is saved in another scratch file.

5. Return code checking. Examine the translator return code to see if errors were encountered, if so, repeat step three to process the next student.

6. Student execution. The program specified as the loader is invoked to bind the student's program with any needed external subroutines, load it into core, and execute it.

7. Loader termination handling. When the loader terminates, the student's program has finished
executing. Note that this may be an abnormal termination. When the current student is finished, step three is repeated to process the next student.

8. Monitor termination handling. When all students have been processed, the monitor must terminate its execution. This releases any resources allocated to the monitor's job step.

In addition to this bare minimum processing, the monitor may perform several amenities which expedite the student running process. These include: keeping a log of student names, along with their translator and loader return codes; printing paging separators identifying student output; and rewarding of students. A very flexible general purpose batch monitor has been implemented and is described fully in Appendix A.

EVALUATION OF BATCH MONITOR APPROACH

When the batch monitor is used, it incurs a certain cost in initializing itself to accept student input. This cost is a function of monitor options and includes any job step charge. After the cost of this initialization is paid, each student saves the two job step charges, plus the overhead of steps one through five, nine, and ten of job step initiation. However, each student is penalized with an overhead cost, previously nonexistent. This is the cost of spooling the student's deck onto an intermediate scratch file before translation.
Without the monitor, each student's deck is read only once. With the monitor, each student's deck is read once and written once to get it on the scratch file, and then it is read again. This introduces an overhead factor proportional to the size of a student's deck. For suitably large decks this overhead factor outweighs the two step charges and initiation overhead savings.

This gives rise to a "break even" student deck size, above which the monitor is not cheaper. It may still be worthwhile to operate in this region if some monitor features are being used to expedite operational efficiency of the running process. A quantitative determination of this break even point is necessarily tied to the type of machine and billing structure being used.

The fixed and deck-size dependent factors of batch monitor overhead may be quantitatively determined by a relatively simple experiment. However, this simple experiment makes one assumption which may be unreasonable. This assumption and its affects will be discussed below. The experiment involves setting the translator and loader names to IEFBR14, a system provided utility which merely returns once it is invoked. To determine the fixed overhead, the monitor is executed with all appropriate options and an empty input stream. The monitor must undergo all fixed overhead items, including job step initiation, monitor initialization, monitor termination, and job step termination.

After the fixed overhead has been determined, the spooling overhead may be found by repeating the experiment with one deck
of one hundred cards in the monitor input stream. Subtracting the cost of the previous experiment yields a high estimate of the spooling cost per hundred cards, assuming linearity of this cost. The reason this is a high estimate is that it includes the cost of invoking the translator and loader (IEFBR14), which is only performed once for each student.

This experiment assumes that the user is charged the same rate for reading the cards, whether they are in card form or in a disk file. Note that IEF3R14 did not read any cards, but we assumed that the overhead was writing the cards on disk, after reading them. It is possible that the cost incurred in reading the cards from the disk will be higher than the cost of reading the cards from the card reader, so to speak. If this is the case, the spooling overhead figures obtained will be lower than the actual cost.

If the assumptions are all valid, though, the experiment may be performed at any installation using its accounting procedures and rates to determine the break even point, or estimate the gains derivable from use of the batch monitor.

These experiments were performed on the MIT Information Processing Center's Model 168 running OS/MVT. Tables one and two summarize the results obtained using MIT's pricing structure ($12.00 per CPU minute, $1.20 per thousand I/O operations, $.10 per job step, and exponential memory costs). CPU times are in seconds:
Using these figures, a simple example can be worked. Suppose you are given six decks which average 300 cards, the spooling overhead should be $0.99/deck. The fixed overhead, assuming log and separators, is $22/6 = $0.04/deck. Thus the total monitor overhead is $1.13/deck. Since the step charges are $0.10, you would expect a savings of $0.07/deck.

If this stream of student decks which actually run at MIT, this gain would not be realized. The reason is the assumption made concerning equality of card reading and disk reading rates. MIT uses LASP, a system in which the CPU overhead for reading cards (from a card reader) is not billed fairly. The historical reasons for this unfair price structure are rooted in the inefficiency of AS3. A flat rate is charged for I/O operations directed through LASP to the card reader. A similar flat rate is charged for disk I/O, but the disk I/O rate is higher, in
addition to the fact that you are billed for your CPU overhead in disk I/O.

LOOP PREVENTION

IBM operating systems enforce execution limits at the job step level. That is, a programmer may specify limits on execution time and printed output for a job step. When one of these limits is exceeded, the job step is terminated in such a fashion that the terminated program is not aware. When the batch monitor is used, many students are to be run under one job step. If normally, each student would be allowed (say) ten seconds of CPU time, then the catch monitor's job step must be given a time limit equal to ten times the number of students.

Now suppose the first student, out of a batch of twenty, codes a program which loops indefinitely. This student is going to use the job step's time limit before the operating system terminates the monitor. In addition to being a costly proposition, this sort of thing tends to happen over and over again. Since the operating system terminates a step which exceeds its time limit in an invisible fashion, there is no way to tell the student where he or she was within the program, or the values of any variables. Since the program ostensibly looked correct in the first place, the student will probably not be certain of the error.
So the student tends to submit the program over and over until the loop goes away. Without the monitor, the student's step time limit is set to a level which makes this cost tolerable, although a definite nuisance. With the monitor, this cost is insufferable. The simplistic approach would seem to involve the monitor acting like the initiator in monitoring the CPU usage of the loader when it is invoked for a student. The IBM operating system makes this approach very complex, and not fool-proof. Also this approach will probably not help the student, much.

The correct approach is to execute the student's program concurrently with a loop preventing governor. This governor would trap any potential loops and perform the correct diagnostic action. This would be a simple program dump with PSW and registers, for assembly language problems. For problems using a higher-level language such as PL/1, this processing is more complicated.

A set of subroutines and procedures has been implemented to effect the correct governing behaviour, or at least make it easy to add to existing programs. The essential precept of this governing approach is that the student's program be executed as a subroutine of a procedure which establishes handlers for looping conditions. I will outline the procedures for assembly language problems (they are fairly simple), and enumerate in detail the procedures used for PL/1 problems.
An assembly language student assignment should be handled in the following fashion:

1. All students code their solutions with the same given CSECT label.

2. A governing procedure is coded which is placed in the loader inout stream physically before the student's program. Additionally, a special CSECT is placed after the student's program in the loader inout stream.

3. This governing procedure uses the STIMER macro with the TASK option to set up a CPU timer.

4. The exit routine specified in the STIMER macro should first close all open files, then obtain the student's registers and PSW, following the procedure outlined later. Then the student's program should be dumped, from its start (remember, we know its CSECT name), to its end, which is found from the special CSECT we stuck after the student. Following the dump, the student's program is terminated.

5. The student's program is called, with any suitable parameters. When the student returns, the timer is cancelled.

For student assignments written in PL/1, a very complicated governing package has been developed. This package consists of
four assembly language subroutines and one PL/1 procedure. This package is used in conjunction with a loader option to provide what has to be the most desirable governing for PL/1. The condition ERROR is raised at the point where the student runs out of time. Although this package is written for the PL/1 (F) language, it may be easily converted to execute with the PL/1 optimizer. The following description assumes a working knowledge of the F-level PL/1 implementation details. (1)

Assembly Language Governor Subroutines (SETTIME, CANTIME, USETIME)

The purpose of these routines is to provide the PL/1 programmer with a convenient interface to the STIMER timing facilities of the operating system. This package is fairly complicated internally, and strongly tied to OS/MVT or VS. Quite simply stated, this package provides a mechanism for limiting the amount of time a PL/1 program will be allowed to use.

To use this package a programmer calls an initialization entry point specifying a primary and secondary time allocation, both in hundredths of a second. Once the primary time allocation has been used the secondary limit is installed and the programmer defined condition (TIMEOUT) is raised.

(1) For reference, see "IBM System/360 Operating System- PL/I(F) Subroutine Library Program Logic Manual", IBM publication number GY26-6901.
This condition handler should close and re-open SYSPRINT to clear any hanging I/O. If the (TIMEOUT) condition handler returns, the timing package will cause the ERROR condition to be raised at whatever point execution was suspended. If the secondary time allotment is used up either by the (TIMEOUT) or subsequent ERROR handler, the task is abnormally terminated with a completion code of 322.

Another set of entry points is provided to tell how much time has been used out of the allotments. One of these entry points cancels the timing as well. Thus if your program doesn't use its whole allotment (or at least if it doesn't use up the secondary one) you can find out how much time you did use.

This package is used by a special PL/1 governing procedure in the following manner (refer to the source listing of this procedure, below):

1. First SETTIME is called to establish limits for the student. The values passed to SETTIME may be altered at execution time.

2. The governor establishes a (TIMEOUT) handler which closes SYSPRINT, opens it, prints an informative message for the student, and then returns.

3. The governor establishes a FINISH condition handler which uses the USETIME entry point to determine how much time a student has used. This value is printed.
whenever the FINISH condition is raised.

4. The student is called. If the student returns, the end of the governor is reached and the FINISH condition is raised. Otherwise the student does something that causes the FINISH to be raised.

PL/1 Callable Entry Points:

SETTIME( FIXED BINARY(31), FIXED BINARY(31));

The first argument is the primary time limit, in hundredths of a second (i.e., a value of 100 corresponds to 1 second). The second argument is the secondary time limit, again in hundredths of a second.

This entry point performs initialization for the timing package. It should be called as soon as possible in a program to be monitored. Note that the program calling this entry point must provide a condition handler for the programmer defined condition (TIMEOUT). This handler, in turn must close and reopen the file SYSPRINT, to avoid problems.

CANTIME(FIXED BINARY(31), FIXED BINARY(31));

This entry point cancels the timing feature and returns an indication of how much of the original interval remains. The first argument is a switch. Its
value is one if the primary allocation was used, and zero if the primary allocation has not been used. Obviously if this value is one you are operating in the secondary limit.

The second argument is the number of timer units remaining in the most recently set interval (the secondary interval if the first switch is on). A timer unit is approximately equal to 26.04166 microseconds.

The code necessary to determine the amount of time used is shown in the governor source listing, below.

```c
USETIME(FIXED BINARY(31), FIXED BINARY(31));
```

This entry point functions exactly as CANTIME, except that the remaining interval, be it primary or secondary is reestablished. This entry point is recommended in case you have a loop in your (TIMEOUT) or ERROR handler.

**Timer Internals:**

When SETTIME is called it issues a STIMER macro naming EXITRTN as the exit routine and the primary time interval as the interval. Then the secondary time interval is stored in a known location for later use and the entry returns.

When CANTIME is called it issues a TTIMER macro instruction with the CANCEL option. This returns the remaining time interval
and cancels the timing. Next a switch is checked to see if a previous interval has expired. This switch and the remaining interval are returned.

The entry USETIME functions exactly like CANTIME, except that before returning it uses an STIMER macro to reestablish the remaining interval specifying SABEND as the exit routine. This routine loads 322 into register one and abends.

EXITRTN is called by the operating system when the primary time limit has been exceeded. The first thing this routine does is issue an STIMER macro instruction to establish the secondary time limit with SABEND as the exit routine. Then a switch is set so the CANTIME and USETIME entry points will know that the primary allocation has been used.

The rest of this routine is very complicated. This routine is entered by the operating system stage two exit effector. For proper results this routine must return to the operating system or abend. Further, there is no attempt on the operating system's part to provide you information about where your program was when it ran out.

To satisfy the condition that we return to the operating system we must first find out where we were and what the registers were (method described later). Register 13 is used as the save area pointer by OS convention. For a PL/1 procedure or begin block, bit 0 of this save area (the first bit of the PL/1
reserved header word) will be on. If we are in a PL/1 procedure we perform the following:

1. Using the PSW, which points to the next instruction to be executed in the halted program, we save the next 14 bytes of the program. A switch is set in EXITRTN for later use.

2. We move into these "evacuated" 14 bytes a sequence of instructions which, when executed will cause control to flow to the label STEP2 in EXITRTN. Care is taken to check the alignment of the starting address so the code will function on a 360.

3. When STEP2 is entered it checks its switch. Noting that it was on it restores the instruction sequence previously saved.

4. EXITRTN returns to the operating system, only to receive control back at STEP2.

If this bit is not on, we are in either a PL/1 run-time library routine or a system I/O routine. Either way it is dangerous to suspend activity, and unlikely that these routines would loop. What we do is thread back up the save area chain until we find a save area with the magic bit on.

This save area is the lowest level procedure or begin block involved. It is used by the subroutine which was called by the
procedure. We place the address of STEP2 into register 14's slot in this save area. When the subroutine is finished it will return to STEP2.

STEP2 then signals the programmer defined condition (TIMEOUT). If this condition returns then the condition ERROR is signalled. Just in case the ERROR condition returns, the next code is for SAGEND.

Finding Registers and PSW in a STIMER Exit:

This method may be used to find the registers and PSW that were active when an STIMER exit was effected. This code functions under OS/MVT and VS releases one and two. This algorithm may work under OS/MFT with the multitasking option, but it hasn't been tried. The algorithm goes something like this:

1. Finding the TCB address. An EXTRACT macro is used with the FIELDS=(FRS) option to get the address of the floating point save area for the task. This area is a prefix to the TCB. Adding 32 (decimal) to this address gives the start (offset 0) of the TCB.

   The first field in the TCB (that is, at offset 0) is the anchor pointer for the Request Block (RB) chain.

2. The first RB on this chain is the IRB created by the stage two exit effector for the STIMER exit routine. The RBGRSAVE field of this IRB (offset 32 decimal)
contains the registers of the interrupted program, stored in the order 0-15.

3. The RBLINK field of this IRR (offset 28 decimal, low order three bytes) points to the previous RB on the RB chain. This is the PRB for the student. The RBOPSW field of this PRB contains the PSW current when the timer ran out. The low order word (offset 20 decimal) of this field contains the address of the next instruction to be executed when the STIMER exit routine returns to the operating system.

The Control Stealing Package

These routines provide the support necessary for the governor program to "steal" control from a student's program that has been compiled with the "OPTIONS(MAIN)" clause. This process requires assembly language subroutines since the student could ostensibly name his or her procedure anything, hence a simple call statement can not be coded in the monitor.

The linkage editor is used to create a load module containing the governor, timing package, and control stealing package. Using linkage editor control statements STEAL is identified as the entry point for this load module. A linkage editor LIBRARY statement is used to stop resolution of the external references to IHEMAIN, IHENTRY, GOVERN, and IHESAFQ.
When the student is to be executed, his or her compiler output is appended to this load module as input to the linkage loader. Use of the loader parameter "EP=STEA", causes the STEAL routine to be entered by the loader when loading is complete. It saves the parameters passed, saves the address of the student's main procedure, which is contained in the CSECT IHEMAIN, and places the address of STAGE2 in this CSECT.

Then the routine IHEENTRY is called. This is the initialization entry point for the PL/1 run time library. It creates the PL/1 environment and then branches to the address contained in IHEMAIN. This is the mechanism whereby the run time initializer knows the "name" of the main procedure.

This causes STAGE2 to be entered, with the PL/1 environment created. STAGE2 creates a String Jope Vector (SOV) for the parameters saved by STEAL, and calls the PL/1 routine GOVERN. This routine is the actual governor, which processes parameters, establishes default error handlers, and execution limits.

When the governor is ready to start a student, it calls the entry point PAYBACK. This routine retrieves the student's address which STEAL garnered from IHEMAIN, places this address in register 15, and branches to it.

When the student is finished, or when an exceptional condition causes control to return to GOVERN, it performs any post processing necessary. Now a slight crock enters into play.
Since we want the IHEMAIN CSECT to point to the student's procedure, the GOVERN procedure must not use OPTIONS(MAIN). However, the procedure epilogue for a main procedure is slightly different from that of a subroutine.

If GOVERN's epilogue is allowed to gain control, it will receive a protection check (system completion code of 0C4). The entry point BAILOUT has been provided to get around this problem. This entry point causes control to be passed to the PL/1 run time library routine IHESAFQ. This routine closes all files, frees all automatic storage and returns to the operating system (loader, in our case).

PL/1 Callable Entry Points:

PAYBACK - This entry causes control to be passed to the student's main procedure. It takes any arguments the governor wishes to pass to the student's program.

BAILOUT - This entry is called by the governor when it is finished processing and wishes to terminate the student's program.

The following governing procedure may be used with any student program written in PL/1 (F). It is designed to function with student programs which have been written with *OPTIONS(MAIN)*. If the student programs are being run as subroutines of some "grader" program, this grader should be
modified to use the timing package described above.

Governing Procedure Source

/* GOVERNING PROCEDURE ESTABLISHES LIMITS FOR STUDENT PROGRAMS, USING THE TIMING AND CONTROL STEALING PACKAGES. THE LIMITS ESTABLISHED ARE FOR CPU TIME AND LINES PRINTED ON SYSPRINT. THIS PROGRAM ACCEPTS THE FOLLOWING EXECUTION TIME PARAMETERS:

MAXPAGES- NUMBER OF SYSPRINT PAGES ALLOWED. Default is 15.

MAXTIME- MAXIMUM AMOUNT OF CPU TIME A STUDENT MAY USE BEFORE HE OR SHE IS INTERRUPTED, IN HUNDREDTHS OF A SECOND. DEFAULT IS 500, .5 SECOND.

STLIMIT- AMOUNT OF TIME A STUDENT IS ALLOWED IN HIS OR HER ERROR HANDLER AFTER MAXTIME HAS BEEN EXCEEDED. DEFAULT IS 100 (.1 SECOND).

LINESIZE- SYSPRINT LINE SIZE, IN CHARACTERS. DEFAULT IS 120 CHARACTERS.

PAGESIZE- SYSPRINT PAGESIZE, IN LINES, DEFAULT IS 50.

AN EXAMPLE PARAMETER STRING WOULD BE:

PARM="MAXTIME=200,STLIMIT=50,MAXPAGES=5"

SINCE THE STEAL PACKAGE IS USED, IF THIS GOVERNOR IS USED WITH THE LOADER THE LOADER PARAMETER "EP=STEAL" MUST BE GIVEN. */

GOVERN: PROCEDURE(PARMS);
DECLARE
PARMS CHAR(100) VARYING,
MAXPAGES FIXED BIN INIT(15),
STLIMIT FIXED BIN(31) INIT(100),
NUM_PAGES FIXED BIN(0),
IPARMS CHAR(101) VARYING,
SWITCH FIXED BIN(31),
INTERVAL_REMAINING FIXED BIN(31),
MULT FLOAT DEC(8) STATIC INTERNAL INIT(2.604166E-3),
TIME_USED FLOAT DEC(8),
INTERVAL FIXED DEC(8,2),
LINESIZE FIXED BIN INIT(120),
PAGESIZE FIXED BIN INIT(50),
MAXTIME FIXED BIN(31) INIT(500),
SETTIME ENTRY(FIXED BIN(31), FIXED BIN(31)),
USETIME ENTRY(FIXED BIN(31), FIXED BIN(31)),
BAILOUT ENTRY,
IMESARC ENTRY(FIXED BIN(31));
/* PARAMETER PROCESSING BLOCK */
BEGIN;
  ON ERROR PJT FILE(SYSPRINT) SKIP EDIT
    (** ERROR *** THE PARAMETER STRING: "****", PARMS,
    *** CONTAINS AN UNRECOGNIZABLE PARAMETER") (A,SKIP,A,A,A);
  IPARMS = PARMS II;
  GET STRING(IPARMS) DATA (MAXPAGES,MAXTIME,STLIMIT,LINESIZE,
    PAGESIZE);
END;

/* OPEN SYSPRINT WITH SPECIFIED PARAMETERS */
OPEN FILE(SYSPRINT) STREAM OUTPUT PRINT PAGESIZE(PAGESIZE)
LINESIZE(LINESIZE);

/* SET UP CONDITION HANDLERS */
ON FINISH BEGIN;
  ON ERROR GO TO EXIT;
  CALL USETIME(SWITCH, INTERVAL_REMAINING); /* GET TIME LEFT */
  TIME_USED = FLOAT(DECIMAL(INTERVAL_REMAINING,8,0),8);
  TIME_USED = TIME_USED * MULT; /* TIME USED IN HUNDREDTHS */
  INTERVAL = - FIXED(TIME_USED,8);
  INTERVAL = INTERVAL + DECIMAL(MAXTIME,8,0);
  IF SWITCH ^= 0
    THEN INTERVAL = INTERVAL + DECIMAL(STLIMIT,8,0);
  PUT FILE(SYSPRINT) SKIP EDIT ('STUDENT PROGRAM USED 
    INTERVAL/100.0, " SECONDS.") (SKIP(2),A,F(8,2),A);
  CALL IHESARC(FIXE(BINARY(INTERVAL,31),31));
EXIT:
  CALL BAILOUT; /* AVOID EPILOGUE */
END;

ON ENDPAGE(SYSPRINT) BEGIN;
  PUT PAGE; /* RESET COUNTER */
  NUM_PAGES = NUM_PAGES + 1; /* BUMP COUNTER */
  IF NUM_PAGES > MAXPAGES
    THEN DO;
      PUT FILE(SYSPRINT) SKIP(2) EDIT
        (** MAX PAGE LIMIT EXCEEDED, EXECUTION STOPPED") (A);
    END;
END;

ON CONDITION(TIMEOUT) BEGIN;
  CLOSE FILE(SYSPRINT);
  OPEN FILE(SYSPRINT) STREAM OUTPUT PRINT PAGESIZE(PAGESIZE)
    LINESIZE(LINESIZE);
  PUT FILE(SYSPRINT) EDIT (** EXECUTION TIME LIMIT EXCEEDED,
      ERROR SIGNALLED.") (SKIP(2),A,A);
END:

PUT FILE(SYSPRINT) EDIT
  (** STUDENT PROGRAM SUCCESSFULLY LOADED.*) (SKIP,A);
CALL SETTIME(MAXTIME,STLIMIT);
CALL PAYBACK; /* START UP STUDENT */
SIGNAL FINISH; /* AVOID EPILOGUE WHEN STUDENT RETURNS */
END GOVERN;

A CHEAPER PL/1 (F) COMPILER

The IRM PL/1 (F-level) compiler is very expensive to use, especially for small programs. Close examination of the compiler's costs discloses that it is fairly efficient in its use of CPU time, but horrendous in its usage of disk I/O. With a typically small student program (< 200 cards), and a well chosen region size (128K-bytes); most of the disk I/O performed by the compiler is the result of program fetches. That is, the I/O is requesting program segments instead of data.

When this compiler was written a design requirement was that it run in a 44K partition, that being the largest user partition on a 64K machine running OS. To meet this requirement the compiler was broken down into about 125 small overlay phases. These phases are loaded as needed, under the control of special subroutines in the compiler root phase. This loading is accomplished by either a LINK; or LOAD and DELETE combination of operating system macros.

The modifications which I made center around changing this overlay structure, making for fewer phases (segments) of greater length. To allow relatively easy changing of this structure, I have used a combination of the linkage editor overlay supervisor
and the LOAD, DELETE, and LINK facilities of OS. The linkage editor is used in conjunction with overlay control statements to produce a compiler load module. Any rarely used phases such as those required only for special features or handling disastrous errors, are excluded from this linkage edit.

When the phase control routines in the compiler root want a certain phase, a table of address constants is checked. If the appropriate address constant is zero, indicating the phase was not linked in, the old method of obtaining the phase is used. If the constant is not zero, a standard OS CALL is used to access the phase through the overlay supervisor. This approach allows the best of both worlds.

The overlay structure may easily be modified to suit different objectives, while rarely used phases do not add to the overhead of segment swapping. To aid in the determination of optimal overlay strategy, a special option 'TRACEFLOW' has been added. This option expects a file TRACE, with attributes similar to SYSPRINT. A formatted, annotated dump of control flow indicating when phases and segments were actually loaded is produced on this file. This option produces much output, hence should be used with care.

In addition to the structural changes, the initialization process of the compiler has been drastically streamlined. In the case of batch compilations, very little initialization is necessary after the first deck. Several new features have been
added to make the compiler more convenient. This includes accurate spill file metering and size estimation. A complete description of these enhancements and their use is in Appendix B.

My modifications are contained in two boxes worth of annotated input to IEBUPDTE, the IBM text maintenance utility. These changes apply directly to the IBM source for the following compiler modules: IEMAA, IEMAB, IEMAC, IEMAK, IEMAL, IEMAN, IEMBW, IEMFV, IEMJZ, IEMLV, IEMQU, IEMQX, and IEMXA. The bulk of the modifications apply to IEMAA and IEMAB.

It should be stressed that absolutely no modifications have been made to those parts of the compiler that participate in code generation or parsing. Thus, the modified compiler produces exactly the same code as the unmodified compiler, at a lower cost. Extreme caution should be exercised when changing the compiler overlay structure, since some routines make assumptions about the availability of a module, and bypass the phase acquisition routines of the root. This usually produces very subtle bugs. The way to find these is to use TRACEFLOW and compare loading activity to the old overlay structure published in the IBM compiler PLM. (1)

EVALUATION OF CHEAPER PL/1 (F) COMPILER

The overlay structure I finally produced requires a minimum partition of about 112K. It successfully compiles small (around 150 statement) programs in 128K without opening the spill file. A general rule of thumb is to add 60K to whatever region size worked for the IBM compiler without opening spill files. If you are trying a medium size compilation for the first time, try 160K.

According to results obtained by Granger and Donovan, (1) the modified compiler is in all cases ten to fifteen percent cheaper than the unmodified version. The gains are greatest for small decks and batched compilations. The modifications act to cut CPU time, disk I/O, and core residence time. The CPU reduction is from reduced operating system entry/exit sequences and I/O processing.

Since these modifications drastically reduce the fixed overhead for a PL/1 compilation, the savings are most dramatic where small (< 100 card) decks are concerned, running as high as 60 percent. Granger and Donovan compared the modified compiler to the unmodified F-level and the IBM optimizer with OPT(0) and OPT(2) specifies. Their results were obtained on an IBM Model 165, running MVT 21.7. The only parameters reported were CPU time (in minutes), disk I/O operations, and core residency time (in

(1) "The Demise of PL/1(F)", Richard H. Granger and John J. Donovan.
minutes). The prices used were $7.00 per CPU minute, $1.00 per thousand I/O operations, and since all core requirements were less than 400K, $.50 per kilobyte-hour for core residence. The results which they obtained for a typical 150 card program are summarized in Table Three, below. In fairness, it should be noted that although more expensive in the compilation phase, the optimizing compiler produces code which is cheaper to execute. However, for student runs execution time is negligible compared to compilation time.

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>OPT(1)</th>
<th>OPT(2)</th>
<th>MOQ-E</th>
<th>IBM-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>.162</td>
<td>.194</td>
<td>.074</td>
<td>.082</td>
</tr>
<tr>
<td>DISK I/O</td>
<td>516</td>
<td>719</td>
<td>413</td>
<td>828</td>
</tr>
<tr>
<td>MEMORY</td>
<td>.423</td>
<td>.526</td>
<td>.283</td>
<td>.498</td>
</tr>
<tr>
<td>COST</td>
<td>$2.18</td>
<td>$2.77</td>
<td>$1.25</td>
<td>$1.80</td>
</tr>
</tbody>
</table>

Table Three: Performance Data for Cheaper Compiler

I have performed similar comparisons between the modified and unmodified compilers, using representative student decks. One of these decks contained 76 cards and performed very little I/O, but manipulated linked lists; the other was 348 cards and performed much I/O and list processing. The tests were performed on an IBM Model 158 running OS MVT release 21.8, using the same accounting procedures as used by Granger and Donovan. The pricing structure was different, however. The rates were $12.00 per CPU minute, $1.20 per thousand disk I/O operations, and $.65 per K-byte hour for memory. The results obtained were:
<table>
<thead>
<tr>
<th>Category</th>
<th>100-F</th>
<th>10M-F</th>
<th>MOD-F</th>
<th>IRM-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0.14</td>
<td>0.21</td>
<td>0.28</td>
<td>0.30</td>
</tr>
<tr>
<td>DISK I/O</td>
<td>330</td>
<td>877</td>
<td>367</td>
<td>906</td>
</tr>
<tr>
<td>MEMORY</td>
<td>205</td>
<td>461</td>
<td>214</td>
<td>481</td>
</tr>
<tr>
<td>COST</td>
<td>$0.88</td>
<td>$1.97</td>
<td>$1.08</td>
<td>$2.21</td>
</tr>
</tbody>
</table>
APPENDIX A: USER'S GUIDE FOR BATCH MONITOR

This appendix describes a general purpose batch monitor designed to expedite the processing of a large number of student program assignments. This is a complex, generalized program which may be used for virtually any two step computing task. It is driven from a large set of execution time parameters, to allow greater flexibility.

Basically, this program reads an input stream of student decks, performing the same processing for each deck. Decks are preceded by a "$JOB" card specifying the student's name, and terminated by a "$EOJ" card. The processing performed for each deck is fairly simple, as follows:

1. Each student's deck is spooled onto a dataset defined by SYSWORK. This spooling process strips off the two control cards.

2. If the separator option is specified, a program named by a parameter is linked to, to print student separators. This routine's input specifications appear later.

3. A parameter-defined translator program is invoked for the student. The alternate ddbname list (1) option is

(1) See, for example, either page 47 of "Assembler(f) Programmer's Guide", IBM publication number GC26-3756, or page 62 of "PL/I(F) Programmer's Guide", IBM publication number GC28-6594.
used by this call, therefore the translator must be IBM compatible. The ddname SYSIN is changed to SYSWORK, the ddname SYSLIB is changed to TLIB, and the ddname SYSLIN is changed to SYSGO. These changes avoid confusion with loader and run-time datasets, which would result from all of the JCL for two steps appearing at once.

4. The translator's return code is compared to a parameter to see if the student will be allowed to run. If not, the loader phase is skipped.

5. If the student's translator return code was alright, the loader is attached to run the student. Parameters may be specified for the loader and the student, and the loader program's name is a parameter.

6. If the rewarding option has been specified, the loader completion code is examined. Except for the case of loader errors, this is a return code from the student's program. If this value is equal to a supplied parameter, a program is linked to. This allows grading programs to indicate a student's score, for possible rewarding. The input description for this routine is given later.

7. If the logging option is specified, the student's name from his or her "$JOB" card is saved, along with
translator and loader return codes. This information is formatted on the file SYSLOG.

8. This concludes processing of the current student. Step one is repeated for the next student, until the input stream is empty.

This monitor is very tolerant of student control card errors, including omission. All but the most pathological cases are correctly handled. The loader program is attached, so a student abend will not affect the monitor, unless the STEP option was coded. The monitor is reentrant, so may be installed in the Link Pack Area to stop student damage.

The monitor accepts many options, which are detailed below. Some of these options may contain subfields, for instance to pass the translator two parameters 'LOAD' and 'NODECK', you would code 'TP=(LOAD,NODECK)'. The parentheses are necessary for correct monitor parameter parsing, but are not passed to the translator. Whenever a program name is called for, it goes without saying that this length must be less than or equal to eight characters in length. The monitor options, with their abbreviations, and default values, are:

SEP, SEPERATOR- If specified, the paging separator option is turned on. This causes a special module to be LINK'ed to to print output separators for a student. The default for this option is on.
NOSEP, NOSEPARATOR- If specified, turns off the student output separator feature.

SN, SNAME- Specifies the name of a load module in the current STEPLIB or LINKLIB, which will be invoked to print separators, if the SEP option is in effect. The default for this parameter is SEPRIN.

LOG- If specified, causes the student log to be maintained. This requires the monitor file SYSLOG. The default for this option is on.

NOLOG- If specified, turns off the logging option.

LC, LINECNT- An unsigned integer specifying the number of lines to be printed per page of the SYSLOG dataset. The default is 55.

REW- If specified, indicates the student rewarding option is in effect. The default for this option is off. This option uses the following two parameters.

RS, REWARDSCORE- An unsigned integer specifying the loader return code which will cause the rewarding module to be invoked. The default is 100.

RN, RNAME- Name of module invoked to perform student rewarding when the student's loader return code matches the rewarding score. Same restrictions as SNAME. The default is REWARD.
NOREW— It specified, turns off the student rewarding feature.

TN, TNAME— Specifies name of translator program. This name should exist either in the JOBLIB or STEPLIB; or in LINKLIB. The default is ASMBLRF.

TP, TPARMS— A parenthesized list of options to be passed to the translator. The default for this parameter is (LOAD,NODECK).

MR, MAXRET— An unsigned integer specifying the maximum allowed translator return code. Students with a value higher than this will not be loaded. The default is 4.

LN, LNAME— Name of loader program. Same restrictions as TNAME. The default is IEWLDGO.

LP, LPARMS— A parenthesized list of options to be passed to the loader program. The default is (NOPRINT).

GP, GPARMS— Parenthesized parameter string passed to the grader, subject to the ZS parameter below. The default is (), i.e., null.

ZS, ZSCORE— A character string which will be appended to the specified GP string whenever the current student had a control card error. This allows the monitor to communicate to grading programs. This parameter should be parenthesized and its default is (BADJCL=1).
These parameters may be specified in any order in the PARM field of the EXEC statement for the monitor's job step. The batch monitor requires the following files, with qualifications mentioned:

MONIN- Always required, contains the student decks with their monitor control cards. This file should have the DCB attributes RECFM=FB, LRECL=80, and BLKSIZE= some multiple of 80.

SYSWORK- Always required, used to spool the student decks. Should have DCB attributes similar to MONIN, but not necessarily the same block size.

SYSLOG- Required if the LOG option is in effect. This file should have the DCB attributes RECFM=VBA, LRECL=125, and BLKSIZE= n*125+4 (ie. 129, 2004, etc.).

SYSPRINT- Required if the SEP or REW options are in effect. Also used by translator and student. DCB attributes may be any commonly used translator compatible attributes.

SYSGC- Required by translator to output machine code. Same DCB restrictions as MONIN.

TLIR- Required if students are to use macros (assembler) or include files (PL/1). Should point to a PDS with correct attributes for translator.
SYSLIN- Provides input file for loader program. Usually this defines a set of concatenated file definitions, with SYSGO’s dataset appearing somewhere (refer to the example, below).

SYSLIB- Used to specify a dynamic linking library for the loader. If PL/1 is used this should point to the dataset “SYS1.PL1LIB.”

Any other files required by the translator, loader, or student should be freely declared, as long as they do not conflict with the required file names. Note that SYSIN is free for student use.

The following example depicts the monitor being used to process PL/1 assignments. A prelinked load module is used to hold the governing procedure which watches the student’s execution. Therefore, the student is second in the concatenating order. The allocation step cuts down overhead in the monitor step.

**JCL for Running the Monitor with PL/1**

```jsp
//A EXEC PGM=IEFBR14
//001 DD DSN=&&GO,DISP=(NEW,PASS),UNIT=SYSDA,,
// SPACE=(TRK,(10,4)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=1600)
//002 DD DSN=&&MK,DISP=(NEW,PASS),UNIT=SYSDA,,
// SPACE=(TRK,(10,4)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=2000)
//003 DD DSN=&&T,DISP=(NEW,PASS),UNIT=SYSDA,,
// SPACE=(TRK,(10,4)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=1600)
//M EXEC PGM=MONITOR,
// PARM=('TP=(LD,N0,NOL,NE),TN=IEMAA,SEP',
// 'LP=(NOPRINT,EP=STEAL)'
//STEPLIB DD DSN=U.M20153.11272.MONITOR.LOAD,DISP=SHR
```
DO DSNU=U.M20153.11272.COPY.MITPL1,DISP=SHR
//SYSLOG DO SYSOUT=A,DCB=(RECFM=VBA,LRECL=125,BLKSIZ=2004)
//SYSSWOK DO DSNU=*A.U0J2,VOl=REF=*A.CD2,
//DCB=*A.U0J2,DISP=(OLD,PASS)
//SYSPRINT DO SYSOUT=A,DCB=(RECFM=VBA,LRECL=125,BLKSIZ=2004)
//SYSLIB DO DSNU=*,A.0D2,VOl=REF=*A.DD,
//DISP=(OLD,PASS)
//SYSUT1 DD DSNU=*,A.003,VOl=REF=*A.DD3,
//DISP=(OLD,PASS)
//SYSGO DD DSNU=*,A.0D1,VOl=REF=*A.DD1,
//DCB=*A.DD1,DISP=(OLD,PASS)
//SYSLIB DD DUMMY
//SYSLIN DD DSNU=U.M20153.11272.PLIMP.LOAD(P1),DISP=SHR
//DSNU=*A.0D1,VOl=REF=*A.DD1,
//DCB=*A.DD1,DISP=(OLD,PASS)
//SYSLIN DD DSNU=U.M20153.11272.PLIMP.DATA(P1),DISP=SHR
//MONIN DD DATA,DCB=(RECFM=FB,LRECL=80,BLKSIZ=2000)
$JOE name of student one, in free form

deck for student one

$EOJ
$JOE name of student two

deck for student two

$EOJ

repeated for other students

*  

The monitor requires from eight to twenty k-bytes of storage, depending on the options specified and buffer sizes (file block sizes). This should be taken into account before assigning a region size. Any parameter errors will cause the monitor to terminate, with an explanatory message.
Separator Routine Input Specifications

The separator routine is invoked to print output separators on the file SYSPRINT, if the SE? option is in effect. This routine is invoked dynamically by the monitor, via the LOAD macro. The name used is gleaned from the SN parameter. Then the routine is BALR'ed to for each student.

Upon entry to this routine, register 15 contains a base address, register 14 contains a return address, and register one points to a five word parameter list, as follows:

1. Pointer to DCB for SYSPRINT. This DCB is opened, with any DCB attributes compatible with the translator being used. If the user leaves the DCB information off of the SYSPRINT DD card, the default attributes of RECFM=VBA, LRECL=125, BLKSIZE=129 are supplied.

2. Contains the address of a character string student identifier, gleaned from the student's $JOB card.

3. Length of the student identifier from above.

4. Pointer to 80 character control card one, which was supplied by student.

5. Pointer to 80 character control card two.

The supplied default separator routine prints the student's control cards at the top of an even page, then the student
Rewarding Routine Input Specifications

If the rewarding option is in effect, this routine is LINK'ed to when a student's loader return code matches a supplied rewarding score. The name used in the LINK request is supplied as parameter RN.

Upon entry to this routine, register 15 contains a base address, register 14 contains a return address, and register one points to a one word parameter list. This word contains the address of an open DCB for the file SYSPRINT. The DCB attributes of this file may be any translator compatible combinations.

The supplied default rewarding module skips to the top of a page, prints a congratulatory message, opens the file REWARDIN, and copies it onto the SYSPRINT file. It assumes that the dataset specified by REWARDIN has the DCB attributes RECFM=FBA, LRECL=121, and BLKSIZE= some multiple of 121. It correctly handles any popular SYSPRINT DCB attribute combination.
APPENDIX B: USER'S GUIDE FOR CHEAPER COMPILER

The modified compiler is operationally identical to the unmodified compiler, with a few exceptions. The best way to use it is to create a PDS which contains the modified compiler's load modules. Then take whatever JCL you use for the standard compiler and add a STEPLIB or JOBLIB DD statement referring to the modified compiler's PDS.

The modified compiler accepts four new options, controlling two new features. These options are:

TF, TRACEFLOW- If specified, the user must provide a file definition for the TRACE print file. The DCB attributes for this file should be RECFM=VBA, LRECL=125, and BLKSIZE= n*125+4, like 129 or 2004. If this option is specified, a formatted, annotated dump of compiler control flow is printed. The default for this option is off.

NOTF, NOTTRACEFLOW- This option turns off the tracing option.

OFF, OFFSET- This option causes a statement offset map to be produced, irregardless of other compiler options.

NOOFF, NOOFFSET- This option suppresses the printing of a statement offset map.
In addition, changes have been made to the handling of "*PROCESS" cards. If you specify "*PROCESS(*);", the system default options will be used for the compilation. If you specify "*PROCESS;", the same options supplied to the compiler upon invocation will be used.

If a 1 is coded in the column identified for control characters (usually column one), the title line is reset and an eject to a fresh page is inserted. This may be used to provided neater listings.