L. LEE PRYOR

DEVELOPING AND MANAGING A LARGE COMPUTER PROGRAM

Successful development of a large computer program requires careful planning and coordination. The management techniques described in this article are the techniques learned during development of the orbit determination program for the Navy Navigation Satellite System.

INTRODUCTION

The development of large computer programs is a management art with its own subtleties. The absence of these subtleties is frequently experienced by managers caught in the software quagmire—software that is 90% "complete," behind schedule, going to be completed next week. . .and then next week. . .and then next week. Managers who realize that adding more people will make bad things get worse are painfully aware that something has gone awry. Just as bad is to have things going well and lose one or two key people from a software effort.

We will describe here our way of implementing large computer programs. Our definition of "large" is a program that requires 10 to 20 people several years to complete, e.g.,the Orbit Determination Program (ODP) that we wrote for the Navy and that has been an ongoing development since 1959. One characteristic of a large computer program is that if it is of any consequence it is *never* completed. More typically, the program continuously evolves in response to changes in the operating environment (changes in computer hardware, instrumentation, physical models, print formats, constants, etc.). The program had best be designed to accommodate change. To do otherwise is to create a dinosaur having limited evolutionary potential.

The current ODP is coded in PL/1 language and consists of 176 subroutines with about 15,000 source statements. The generated machine instructions are about a million bytes on an IBM 3033 computer. A structured overlay scheme permits the program to run in a region of 600K bytes. A typical orbit computation takes about 10 minutes of 3033 time. On the operational computer, an IBM 360/65, it executes in one hour. The program is supported by seven "utilities" that are part of the executable program library. The total cost for development of the program is estimated to be $3 million.

DESIGN OBJECTIVES

A number of design objectives were established at the beginning of the project. Some probably resulted from the fact that the predecessor program, which survived for 17 years, was coded entirely in assembly language. It was recognized that the program would process all possible data types, e.g., bit string data, character string data, integers, floating point numbers. It was also recognized that the vehicle machine, defined by Brooks[1] (Chapter 12) as the machine on which the program would be built, would be APL's IBM 360/91. The target machine, defined as the machine for which the software was being written,[1] would be the IBM 360/65. (The IBM 360/65 was nearly equivalent in computing power to the then operational IBM 7094.) Nevertheless, one objective was to develop a program that would be as machine-independent as possible.

Programs with a projected lifetime of 10 to 15 years undergo continuing stages of maintenance. Accordingly, it was desirable to have a program code that would be readable by many users. The system of the previous generation depended entirely on sequential access devices (tapes) and, therefore, had some obvious limitations (e.g., the data had to be time-ordered, and data and results files had to be synchronized).

The newer computer systems were oriented toward direct access devices (disks) and the requirements of the program were better served by direct access logic. Hence, the ODP was designed to be file driven and controlled, and presently uses 20 distinct data files. The file-driven nature of the program permits adjustment of the program to changes in operational conditions without the need to recompile any source code. New satellites in the constellation or new station equipment are examples of new entries in a peripheral data file. Time-dependent functions that cannot be modeled, such as the position of the earth's axis of rotation, the ultraviolet activity of the sun, or the activity index of the earth's magnetic field, are also kept as data files. Even fundamental constants in the gravity model of the earth (an array of about 900 coefficients) can be changed by means of a data file, i.e., without disturbing any logic in the program.

It was desirable from an operational standpoint to make the program as automatic as possible. The program must run reliably for 10 orbit computations per day and should not require the attention of computer technicians or analysts for routine daily operations. A graceful embedding in the operational environment meant that the program had to have numerous threshold tests and error messages, and that the program had to make many decisions internally.

The program was designed (a) to be machine independent (it would run on a number of PL/1 machines); (b) to be readable with a highly-segmented code for ease of maintenance; (c) to be file driven and controlled to permit model changes without forcing code changes; and (d) with the target machine and running time in mind.

The selection of language was difficult. We had to have a higher-level language for readability and machine independence. If there were language limitations, we did not want to compound our problems by resorting to a second language for special functions (see below); we wanted to operate solely in whatever language was chosen. We needed enough efficiency to achieve reasonable running time on our target machine. "Reasonable" meant that the execution time should not exceed that of the older program.

We needed a broad range of input/output methods (sequential indexed sequential, direct access) and a broad range of data types. We knew from experience that to control storage allocation we would need program overlay logic. It was considered desirable to have a language that had been defined as a standard. Among the candidates at that time, FORTRAN and PL/1 were the major contenders, particularly since the vehicle machine was an IBM 360/91 and those two languages were readily available. FORTRAN had already been defined to be a national standard[2,3] and was (and still is) the most widely available higher-level language. But compared to PL/1, FORTRAN had some limitations: it did not have as much generality in input/output methods, in data types, or in control of storage allocations. There was pressure from the Navy to choose FORTRAN because of its universal acceptance. But there was pressure from the programmers to choose PL/1 because it was felt to be a richer language and because it had the compile time facility that we saw as a powerful tool in establishing program standards and conventions. PL/1's data and program structure features, its interrupt handling "on conditions," and greater choice of input/output methods made PL/1 more attractive than FORTRAN. Finally, the Navy yielded the point and PL/1 was accepted. Later, PL/1 was also defined to be a national standard computer language.[4]

THE NEED FOR FOUR LANGUAGES

One of the petty tyrannies of the current computer world is that we cannot avoid dealing with the software systems that control the various peripheral devices on the computer and or allocate storage within core. This, in turn, forces us to "speak" three other languages in addition to PL/1, the mother tongue; four languages are actually needed to implement the orbit determination program. Basically, of course, the program is coded in PL/1. But to run the program one must allocate files and create data sets; run utility programs; and route output to disk files, printers, or card punches. All this requires a knowledge of the Job Control Language. Creation of the program from its subroutine components involves the use of program overlays and if one wishes to be system-independent, one must be very careful in the selection of system modules. These factors require knowledge of the operating system's linkage editor which has its own language (LKED). There are machine language (MACHINE) considerations in the design of data structures and choices of input/output methods. In particular, bit string lengths, integer precision, floating point precision, character string lengths, and structure alignments were chosen to reflect the underlying machine language.

DOCUMENTATION

It was planned from the beginning that the program listing would be the definitive document on each subroutine. That program listing was augmented, wherever necessary, with pages of analytical text to provide the "complete" program documentation. The program documentation evolved along with the program in a series of internal memoranda that were accumulated in looseleaf notebooks. When the time came to publish the document for release to the user, a final editing of those working papers became that document. Moreover, the documentation accurately reflected what was programmed. Brooks[1] (p. 169) has pointed out the advantages of using self-documenting programs to reduce the need for coordinating two independent media (program and document).

LANGUAGE-INDEPENDENT STANDARDS

Orbit determination is primarily a problem of physics and mathematics. Some program standards and conventions could be adopted independently of the language chosen and oriented to the analysis behind the problem. It is important for the programming staff to develop its own standards. The staff is in the best position to know what specific standards are needed but, more importantly, development within the staff provides the strongest motivation to conform to the standards[5] (Section II). These standards were also applied to the assembly-language predecessor of the PL/1 Orbit Determination Program. Some of the language-independent standards that were adopted are the following:

1. There is a pool of fundamental constants from which all other constants are derived.
2. Units of length are consistently carried internally in earth radii. As a consequence, all lengths are about unity, and anomalies (errors in the first several significant figures) are easy to see.
3. Units of velocity are carried internally in earth radii per second.

4. The time scale is UTC (Universal Time Coordinated) and time is specified in a format of three numbers consisting of the year minus 1900, the day number of the year (January 1 = 1), and the seconds of the day. The format for time is designated "epoch."
5. The program processes data from one satellite at a time. Where combined solutions involving more than one satellite are needed, they are accomplished by writing summary files and combining the summary files with an independent program.
6. Each major subprogram processes data in entities of one pass of a satellite over a station. This seemingly arbitrary decision added logical order to the program.
7. Satellites and tracking stations are always identified by unique numbers. Associated with these numbers are files of station and satellite characteristics.
8. Subroutines are liberally endowed with commentary giving program name, programmer,[6] function, date of implementation, modification dates, input and output description, references for algorithms, and descriptive comments along with the code (Fig. 1).

LANGUAGE-DEPENDENT STANDARDS

The language-dependent standards took advantage of features within PL/1. The fact that commentary could be injected into and interspersed between statements meant that the right-hand side of each page could be entirely devoted to explanatory comments. Uniformity of definition for many of the basic program components was insured by means of the PL/1 compile-time library. This library contains data definitions and macros (program statements that are generated at compile time) that can be included with the programmer's code at compile time, hence relieving the programmer of having to produce redundant blocks of code with each subroutine. Those blocks of code, which must be identical between subroutines, are kept in one place—on the compile-time library—thus insuring identity between subroutines (Fig. 2). Some examples of items from the compile-time library include the following:

1. The precision of the variables. (All the algebraic computations are performed in floating point double precision, and we wished to insure that all variable were declared with the same precision).
2. The declarations and values for all fundamental and derived constants such as pi, the speed of

```
/*  ADEN      ATMOSPHERIC DENSITY                                            */
/*    FUNCTION SUBROUTINE                                                    */
/*    PROGRAMMER: ARIE EISNER                            DATE: 9/19/68       */
/*                                                     REVISED: 8/03/70      */
/*    PROGRAMMER: L.L. PRYOR CHANGED FOR MAGSAT          DATE:05/29/80       */
/*                          LOWER ALTITUDE THRESHOLDS                        */

/*    PURPOSE:                                                               */
/*      ADEN COMPUTES ATMOSPHERIC DENSITIES AND IS BASED ON THE              */
/*      JACCHIA 1965 MODEL. THE PROGRAM WAS MODIFIED TO GENERATE             */
/*      DENSITIES FOR ALTITUDES ABOVE 500KM AND EXOSPHERIC TEMP.             */
/*      OF 600-2100 DEG.-K. PROCEDURE ASSUMES BOUNDARY AT 500KM,             */
/*      DETAILS MAY BE FOUND IN S1A-413-68 BY ARIE EISNER.                   */
/*      IN ADDITION TO ADEN THREE ADDITIONAL ENTRIES ARE PROVIDED:           */
/*        1.ADENI  - DENSITY INIT., GENERATES BOUNDARY VALUES AT             */
/*                   500 KM AND ALLOCATES CONTROLLED STORAGE.                */
/*                   ADENI MUST BE INVOKED ONCE PRIOR TO THE FIRST           */
/*                   CALL TO ADEN.                                           */
/*        2.ADEND  - NEW DAY ENTRY, MUST BE INVOKED WHENEVER DAY             */
/*                   NUMBER CHANGES. COMPUTES SOLAR AND SEMIANNUAL           */
/*                   CONTRIBUTIONS.                                          */
/*        3.ADENT  - DENSITY TERMINATOR, FREES ALL STORAGE ALLOCATED         */
/*                   BY ADENI. ADENT SHOULD BE CALLED WHEN NO                */
/*                   ADDITIONAL DENSITY COMPUTATIONS ARE NEEDED.             */

/*    USAGE:                                                                 */
/*      PARAMETERS TO ADEN ARE:                                              */
/*                                                                           */
/*    NAME          DESCRIPTION                                              */
/*                                                                           */
/*    PAR1(3)       INPUTABLE PARAMETERS IN SOLAR HEATING                    */
/*                  VARIATION COMPUTATION.                (ADENI)            */
/*    PAR2(2)       INPUTABLE PARAMETERS IN GEOMAGNETIC HEATING              */
/*                  COMPUTATION.                          (ADENI)            */
/*    PAR3(5)       INPUTABLE PARAMETERS IN SEMIANNUAL                       */
/*                  VARIATION COMPUTATION.                (ADENI)            */
/*    PAR4(6)       INPUTABLE PARAMETERS IN DIURNAL                          */
/*                  VARIATION COMPUTATION.                (ADENI)            */
/*    IPRNT         DEBUG PRINT FLAG '1'B=DEBUG-PRINT     (ADENI)            */
/*    DAYN          DAY NUMBER (JAN 1 = 1)                (ADEND)            */
/*    ADSUN         RIGHT ASCENSION OF SUN ON 'DAY'(RAD)  (ADEND)            */
/*    DDSUN         DECLINATION OF THE SUN ON 'DAY' (RAD) (ADEND)            */
/*    FCA           3-6 MONTH RUNNING AVERAGE OF THE SUN                     */
/*                  SOLAR INDEX S 1.E-22 WATT/M2/CYCLE/SEC(ADEND)            */
/*    FC            DAILY VALUE OF S.                                        */
/*    RLV(3)        SATELLITE POSITION VECTOR (RO)        (ADEN)             */
/*    KCP           MAGNETIC INDEX                        (ADEN)             */
```

Fig. 1—Program listings are nearly self-documenting. Each subroutine contains a prologue of descriptive commentary.

light, the rotation rate of the earth, the gravitational constant, and the radius of the earth in kilometers, etc. (Some of these constants change at rare intervals. This implementation assures that the same values are used throughout the program and, if changed, are changed throughout.)

3. Fundamental data structure definitions for orbit description, time (epoch), and input/output records.
4. Algorithms that are frequently used, such as the magnitude of a vector, the cross-product and outer-product of two vectors, and operations with epoch (year, day, seconds) time.
5. Entry declarations for all subroutines in the system. (This is a PL/1 language feature whereby the compiler checks calling sequences for validity and prevents errors from incorrectly coded argument lists.)
6. The satellite property structure. (Each satellite has unique properties, some of which are time-dependent. The overall philosophy is to have the programs process data for each satellite independently. The satellite property structure is made available at all levels of the subroutine hierarchy so decisions may be made at any level.)
7. The tracking station property structure. (The tracking station property entries are unique for each tracking station. The structures for all stations are kept on a peripheral file. The structure for each particular station is loaded only if data from that station are present in the data batch.)

PROGRESS MONITORS

The initial stages of design required a description of the overall logic and major processors, establishment of program standards and conventions, and descriptions of data structures and peripheral files. Once the initial design was documented, much of the work could proceed in parallel, with people working independently on input/output packages, data utilities, or major processors. At this stage, when the initial design has been documented and work is proceeding in parallel on program components, the basic program structure is frozen[1] (p. 42). Progress on the programs was monitored by means of a completion schedule and a change log.

The completion schedule was merely a chronological chart showing the activities that were proceeding in parallel, with an indication of the starting and ending times, and the duration. An example of a completion schedule is given in Fig. 3. The program was intensively worked on for three years, then was delayed for four years, and was finally finished in three years (in 1979). The change log provided for upgrading of older programs that might have become outdated because of revisions in the implementation. Older programs could become outdated in several ways. An outstanding example is the release of a new compiler. The PL/1 optimizer was released during our implementation; after careful evaluation, we decided to recompile our entire library with the new compiler. There were a few language incompatibilities and thus many older subroutines had to have minor modifications for the new compiler. Other changes were made to enlarge record structures for the peripheral files or

```
MEMBER CPI
DCL(CPI INIT(  3.141592653589793E+0),
    C2PI INIT(  6.283185307179586E+0),
    CHFPI INIT(  1.570796326794897E+0)  ) FLT EXTERNAL ;

MEMBER CC
DCL CC     INIT(  47.00293156371283E+0)    /* VELOCITY OF LIGHT IN       */
                           FLT EXT;        /*   VACUUM IN RO/SEC =       */
                                           /*   299792.5KM/SEC(AMS-55)   */

MEMBER CRO
DCL CRO   INIT(  6378.166E+0) FLT EXT;     /* RO TO KILOMETERS           */

MEMBER EPHEMP
    DCL 1 EPHEMP CTL,                      /*EPHEMERIS RECORD.           */
          2 ID CHAR (8),                   /*  IDENT = 'EPHEMP'          */
          2 RE FXD,                        /*  RESERVED                  */
          2 NPAR FXD,                      /*  NUMBER OF PARTIALS        */
          2 CARTE SSCO,                    /*  CARTESIAN ORBIT INCLDS:   */
                                           /*  CEP- EPOCH                */
                                           /*  RRD- RLV-POSITION IN RO   */
                                           /*       -RLDV-VELOCITY RO/SC*/
          2 RLDDV(3) FLT,                  /*  CARTESIAN ACCELERATION    */
                                           /*  IN RO/SEC**2              */
          2 FCSMV(3) FLT,                  /*  SMALL FORCE VECTOR        */
          2 RLFV(3)  FLT;                  /*  CARTESIAN POSITION        */
                                           /*  WHERE SMALL FORCE WAS     */
                                           /*  EVALUATED                 */

MEMBER SSGREN
    DCL GREN    ENTRY (EPOCHP)             /* FUNCTION FOR LONGITUDE     */
                RETURNS(FLT);              /*  OF GREENWICH              */

MEMBER SSWDG
    DCL WDGI ENTRY(FILE,CHAR(*),CHAR(*));  /*  WRITE INITIAL             */
    DCL WDG  ENTRY(FILE,RDDP);             /*  WRITE NEXT POINT          */
    DCL WDGT ENTRY(FILE,CHAR(*));          /*  TERMINATE WRITE           */

MEMBER SSMODEP
    DCL MCDEP ENTRY (EPOCHP);              /* NORMALIZE EPOCH            */
```

Fig. 2—Compile time text included from a common source insures uniform definitions between subroutines.

1. Revise atmospheric drag model to reflect the work of Eisner and Yionoulis.
2. Revise radiation pressure model.
3. Investigate use of fill words in the message to convey a compact ephemeris and solar index number.
4. Write summary data for automatic backup.
5. Change the Post-Editor Fit to punch a run summary card. Change the PEF to print the summary data in meters rather than earth radii. Change the breakout criteria in PEF.
6. Change subroutine GGRD to remove a preprocessor error.
7. Apply the station equipment delay time to the fiducial time points.
8. Change the Kepler fitting program for the case when eccentricity goes negative. Adjust perigee by one-half the period.
9. Correct an error in the Siftor for the case when TRANET-2 passes are deleted.

Fig. 4—Program change log.

ADENB	EDRDMSRH	FADL
ADENC	EDRDSPHA	GETSKA
CHALWA	EDRDSPHB	GETSKB
CHALWB	EDRDSPHC	GETSKC
EDRDDG	EGPA	GVPC
EDRDDH	EGPB	GVPD
EDRDDI	EGPC	GVPE
EDRDMSRF	FADJ	GVPF
EDRDMSRG	FADK	

Fig. 5—A sequence code appended to the name of each program component indicates evolutionary changes.

for internal data formats. The change log (Fig. 4) is a tool for continuing maintenance and is continuing even now after the program has been operational for two years .

PROGRAM ACCOUNTING

Some means of maintaining an orderly accounting of program components is necessary in the development of a large program. The method established with the Orbit Determination Program took advantage of the file handling utilities of the IBM operating system. Central libraries were established for source code (PL/1 statements), compiled modules, and executable versions of the program. Checked-out subroutines were stored as source code in the source library and compiled versions were stored in the load module library. The completed components were used to generate a production program. All hard-copy listings were kept in a central file for reference.

Confidence in new blocks of code increases in direct proportion with the time that the code has been in use. Accordingly, whenever a subroutine had to be upgraded or revised, the new version did not replace its predecessor in the library but became an addition to the library[1] (p. 149). It was always possible to revert to the previous version of a program if revisions inadvertently produced bugs. A naming convention was adopted whereby every subroutine had a unique name corresponding to its major entry point. These names were used for filing listings and as member names in the program libraries. When a successor was generated the entry name was unchanged, but the library member names were constructed by appending an A to the first successor, a B to the second successor, and so on. Most of our program components are in the A or B stage, but there are a few especially troublesome and critical components that are in the K, L, and M stages. We were particularly careful to avoid changing calling sequences (argument lists) so that revised subroutines would be compatible with their predecessors (Fig. 5).

PERIODIC REVIEWS

As programming progressed, there was a natural evolution from the component stage to the system integration stage. We conducted internal reviews of our progress about every month and we supplied external

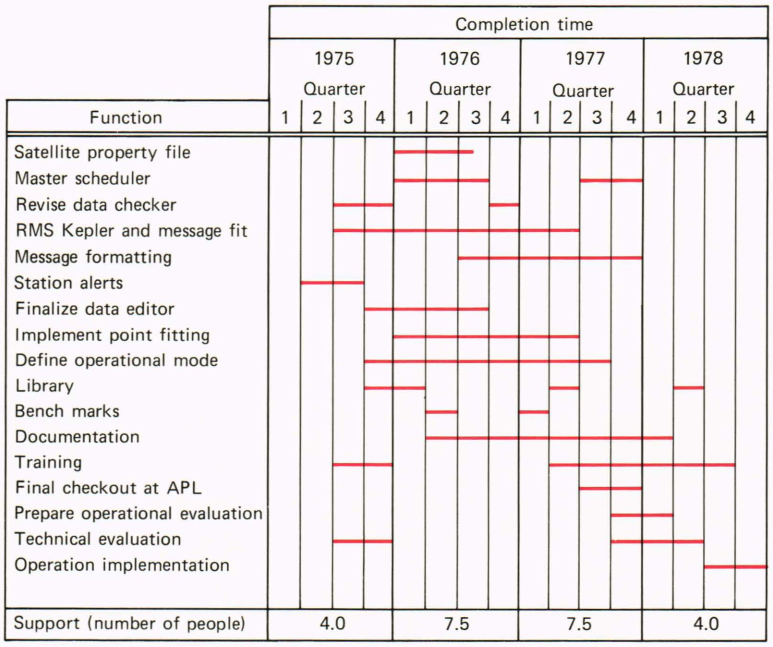

Fig. 3—Sample completion schedule.

progress reports in a semiformal management meeting for the program sponsor and the user group. Those reviews provided an opportunity to revise our completion schedule and update our change log. The reviews also showed some of our initial design features to be bad ideas and convinced us to abandon them. One feature was "multi-tasking" (using the same code to process several independent data streams). We thought the program should process several satellites simultaneously. That feature was available, but we did not need to provide multi-tasking within the program since the vendor operating system has multiprogramming. Another feature was an input processor to scan and interpret the control data. We elected to use PL/1 "data-directed input" in lieu of implementing our own input processor.

SECURITY

Program security—preventing the loss or destruction of our work through some local disaster—was a concern. We routinely unloaded all files from our central computer from disk to tape and stored our tapes locally, but remote from our computer center. As we reached the final stages of completion, we issued advance copies of the program to the user group in California for their review and storage. In that way, we satisfied our security problem and also provided a close liaison with the future users.

INSTALLATION

The program installation at the user site consisted of an intensive two week program staffed by six people. The first week was mainly concerned with the tedious problem of compiling all components and running test cases on the user computer system. The week included classroom lectures on the physics and mathematics theory. The second week continued classroom lectures on the analysis behind the program, on the program itself, and on data file organization. The classroom work was enhanced by laboratory sessions where the user staff submitted computer runs using the new program. There were about 20 people participating from the user staff. After initial installation, the user group performed experimental runs to test the program and later performed parallel operational runs. That program test period proceeded for about a year with continuing feedback of questions and problems to the development team.

I offer the following checklist for managing any large-scale programming effort:

1. Establish the design objectives early and have them written down. It is important to realize that software, if it is to be worthy of the name, should be flexible and amenable to change. Otherwise the logic would have been implemented in hardware.
2. Ask the program staff to establish the standards and conventions they can agree to follow. I think it is important for the staff to establish their own standards. Neither management-dictated nor very elaborate standards will be effective.
3. Postulate a completion schedule and revise it periodically to match reality.
4. Maintain a change log in addition to the completion schedule so that completed components that need modification will not be overlooked.
5. Conduct a periodic review at intervals frequent enough to keep everyone informed of progress but not so frequent as to interfere with the ongoing work. (Weekly or monthly seems about the proper frequency for review.) If the reviews are not helpful to the staff cut their frequency and change their emphasis.
6. Have a central file for storage of completed programs including listings, source code, and documents. Have a procedure for maintaining this central file. It is imperative that you have all related source codes no matter how slapdash and impermanent they may seem.
7. Have a security plan to protect your files from local disaster or inadvertent destruction.
8. Solicit continuing feedback from the ultimate users of the system so that what you construct can be incorporated into their operation with as little impact as possible. (It is important that you work on the problem, not on what some specifications says should be done.) It is best to plan for three or four preliminary versions of the system before the final operational version is introduced[1] (p. 150).
9. Maintain a distinct line between production or operational software and research or developmental software. Only developmental software that has been thoroughly checked should be promoted to operational status. If the project is large enough, a similar line should be maintained for personnel assignments. Those people engaged in operational computing should offer continuous feedback to those engaged in software maintenance and development.

Technology frequently advances on parallel fronts. More discerning people have defined and published concepts such as "top-down design," "structured programming," and "programming team"[5] that we discovered empirically to be very helpful in developing the orbit determination programs.

REFERENCES and NOTES

[1]F. P. Brooks, Jr., *The Mythical Man Month*, Addison Wesley, 1978.

[2]American National Standards Institute, *American National Standard FORTRAN*, ANSI X3.9-1966.

[3]American National Standards Institute, *American National Standard FORTRAN*, ANSI X3.9-1978.

[4]American National Standards Institute, *American National Standard PL/1*, ANSI X3.53-1976.

[5]G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand-Reinhold, 1971.

[6]It is very important that the programmer's name appear here. If changes are required later, we want to know with whom to talk. The best documentation is really no substitute for a person's knowledge.