DCIEM RESEARCH PAPER NO. 900

PROBLEMS IN SOFTWARE DEVELOPMENT

V.K. Taylor

DEFENCE AND CIVIL INSTITUTE OF ENVIRONMENTAL MEDICINE
INSTITUT MILITAIRE ET CIVIL DE MEDECINE DE L'ENVIRONNEMENT

DEFENCE RESEARCH BOARD, CANADA, CONSEIL DE RECHERCHES POUR LA DEFENSE

APRIL 1973

DCIEM RESEARCH PAPER NO. 900

PROBLEMS IN SOFTWARE DEVELOPMENT

V.K. TAYLOR

Computation and Analysis Group
DEFENCE AND CIVIL INSTITUTE OF ENVIRONMENTAL MEDICINE
1133 Sheppard Avenue West, P.O. Box 2000
DOWNSVIEW, Ontario.

DEFENCE RESEARCH BOARD — DEPARTMENT OF NATIONAL DEFENCE — CANADA

Keynote Address for Session (1) of the
Commonwealth Defence Science Organization
Symposium, April 1973.

PROBLEMS IN SOFTWARE DEVELOPMENT

The Early Days

Although special-purpose electronic hardware has been used for many years to perform logical operations, only in the past two decades has the programmable electronic calculator come to the fore. In the early days programs were written in a form which was very close to the bit structure of the machine, e.g., on the FERUT computer at the University of Toronto, program input was on five-level paper tape such that four successive tape characters gave the binary representation of one 20-bit word in the computer. Although much good work was done using this type of programming, only a small group of highly skilled practitioners was able to take advantage of this new tool and the majority of good work took place inside the confines of universities and research institutions. Programs were noted more for their intricacies and elegance than for their legibility. This was in part due to the small size of computer stores available and also to the nature of the people doing the work. Programs written by one person were not easy to develop or maintain by anybody else. The idea of an "identifier" or "label" and thence an instruction format was very important in the history of software development. Instead of putting an instruction into the computer in terms of four five-bit characters, an instruction could be entered in terms of its component constituents making the program easier to read and to develop.

A second major advance was the implementation of the first compilers. Representative was the University of Toronto "Transcode" which transformed the Ferranti FERUT computer from a one address, fixed point machine to a three address, floating point machine in which standard arithmetic operations were represented in one pseudo-machine instruction. Transcode was primitive and simple but did open up the field of computer programming to a far larger population.

Another advance made in this era was the implementation of the concept of "relocation" which enabled program sections to be created and tested independently with the knowledge that regardless of where the program sections were stored they should work. Changes in the interior structure of one program section would not interfere with the operation of other sections. This reduced the development period for a program by at least an order of magnitude. Now large programs could be split up and structured since program sections could be written and developed to fit together knowing only the "black box" behaviour of the different program sections.

Computer operating stores in this era were still quite small, e.g., the main store of the IBM 650 was a drum of 2,000 word capacity. Input and output were still very simple and could not take place concurrently with computation.

In 1954, a group headed by J.W. Backus brought out its famous report on the original Fortran language. The source machine for the Fortran language did not deal with instructions in the classic sense but rather with program statements, where a statement might involve the evaluation of an algebraic expression. Fortran was rather complex; the implementation of the original Fortran compiler on the IBM 704 took over 40,000 instructions, which was 10 times the core size. Nevertheless Fortran became so popular that by 1963 there were in existence compilers for 84 different dialects of the language. Fortran was supposed to be a machine-independent language, but due to the proliferation of dialects this was not the case. However, the major acceptance of such a language was a step in the right direction. Prior to this any program which outlasted the original computer for which it was written had to be regenerated for a new machine from the original flow diagrams, or more likely, from the original problem definition since the logic of the flow diagrams was liable to have a machine-dependent structure. In 1964, a standard for the Fortran language was produced upon which most modern Fortran compilers are based. Many manufacturers have enhanced the language which, although increasing its generality and therefore, supposedly, its usefulness, has meant that the average Fortran program is still not universally portable.

The Evolution of Monitor Systems

By the late 1950's the relative increase of speed of central processor operations as compared to peripheral operations led to an ever-increasing imbalance in the roles of these two sections of the computer system. In order to help redress this imbalance, peripheral operations which previously had effectively stopped the computer now took place in parallel with the main computation process. This meant that the control of peripheral operations became highly complex and far less understood by the general rank of programmers. Today, the amount of store required to do reasonable peripheral operations is greater than the total operating stores available on the machines of the early 1950's.

A second method used to attempt to redress the imbalance between peripheral operations and computing, especially for large machines, was to insure that all input-output operations took place through high speed devices. Any slow speed operations such as reading or punching cards, or printing, would be relegated to a smaller, cheaper computer whose main purpose was to produce the high speed input form required or to accept the high speed output form from the faster and more expensive machine. In this way, the IBM 1401 first saw the light of day as an off-line input-output processor for the IBM 704, 709 series of machines.

The disassociation of input-output operations from the rest of the machine led to the generation of Input-Output monitor systems which controlled all program-initiated peripheral transfers. To the programmer the Input-Output monitor became part of a central computer in which the standard input-output instructions of the basic machine were replaced by special commands to the monitor system. Although the monitor system increased the efficiency of utilization of the peripheral devices an additional level of complexity had been added to the development of programs. Prior to this, errors could occur only in the program or the hardware. Now an error could occur at any of the three levels and it was in many cases more difficult to isolate.

At the same time as Input-Output monitors were being developed so were "Batch" monitors, the two usually being combined as one entity. In such a system programs were entered into the machine with a minumum of operator intervention, each program being controlled by a "job" description. The jobs would be entered into a "job stream" with each job being run sequentially. Operator action was reduced to the loading of peripheral devices and the investigation of anomalous behaviour. The throughput on the computer was certainly increased by this method but program development was again made more time consuming since the programmer himself was now disassociated from his computer and operated solely on paper. In such a system there could be no affinity between the programmer and the computer. The programmer entered his request into a queue of jobs. Eventually, he would get a result -- probably a day or two later. Thanks to such languages as Fortran the time for development of a new program did not go completely out of control. However, it was estimated that the normal time required to develop a new program was about the same as the time required to write it. The era 1960–65 was the acclaimed field day for Fortran in all its dialects. Throughout North America there were thousands of installations which ran nothing but Fortran programs. The Fortran compiler would be so imbedded in the monitor systems that the programmer could easily write programs thinking that the hardware itself assumed the Fortran structure.

It was realized by 1960 that although the disassociation of peripheral operations from the central processor substantially increased the efficiency of operation of the computer, it still did not create a match between peripheral and processor speeds. The result was that a considerable amount of central processor time was available but unusable during the running of a typical program. The development of the time-shared computer alleviated this problem but created another one, namely, that some of the computer time which was previously available for the running of a program had now to be used to control just which program in a series was to be run.

To implement the time-sharing concept safely, fundamental changes were made in the basic architecture of central computer systems to insure that no program could influence the operation of any other program. A fundamental philosophy in time-share operation is that a program must be able to run, regardless of where in the operating store that program is placed. This can be accomplished only by a very close integration of the hardware and the time-share monitor system. Complex algorithms are required to determine which of a set of programs should be running at any given time. Normally there is a trade-off between selecting the best program to run and the time taken to make the selection. Time-share monitors can become so complex that well over 70% of the actual operating time is spent in the monitor system performing housekeeping operations rather than in the running of the actual programs. When Dr. Stanley Gill was asked about the relative simplicity of the scheduling algorithms for the Atlas computer, he replied that he did not wish to spend 95% of his time determining how the other 5% should be used.

Until 1965 computers were highly individual. It was rare to find two computers made by the same manufacturer which had the same instruction set. Programs written for one computer would not run on a second. By 1965, manufacturers realized that this caused great inefficiency and duplication of effort in the generation of software for their different machines. Furthermore, the public demand

for computing power was leading to the generation of some very complex pieces of software. In 1964, IBM announced its 360 series of computers. In 1965, International Computers and Tabulators delivered their first 1904. These events marked the start of the "family era." Manufacturers produced a range of computers from the very small to the very large, all of which had the same instruction repertoire. "Upwards compatible" became the phrase of the day. This meant that any program written for a smaller machine would run on a larger machine of the same series. This never quite worked in practise, but it did enable a manufacturer to concentrate a major part of his effort on the generation of software for a system rather than for an individual machine. Minor effort was then required to tailor the software for the various different computers of the series.

Large Systems

We are now talking about very complicated, highly sophisticated operating systems with which any application program must interface if it is expecting to run on the modern computer. The computer is so far embedded in its operating environment that, to the general run of programmers, its architecture is almost completely lost from view. To illustrate the fantastic explosion in the production of system software over the past 15 years, consider that in 1955 the operational software on the IBM 650 was written in 10,000 lines of code whereas in 1966 the software requirement for the 360 series was 1,000,000 lines. This figure rose to 7,000,000 lines by 1969. This fantastic explosion is parallelled by the very rapid increase in the number of programmers who are doing the work. D'Agapeyeff notes that in 1958 a European general-purpose computer manufacturer often had fewer than 50 software programmers whereas in 1968 they probably number between 1,000 and 2,000 people. T.J. Watson, when Chairman of IBM, is reputed to have said that OS/360 cost IBM over $50,000,000 a year during its preparation and at least 5,000 man-years investment. It has been said that development costs for software equal the development costs for hardware in establishing a new machine line. These growth figures should be viewed more with alarm than with pride because it is obvious that expansion of this magnitude involves great technological leaps which are not necessarily crowned with success. This lack of success is especially important when it occurs in the underlying software of an operating system. It leads to the release of many issues and versions of the same system. This means that a programmer who is taking advantage of any of the detailed, non-publicized features of an operating system may find that his program will not run without modifications on a new release of the same operating system. The Canadian Dept. of National Defence runs an IBM 370/155 computer in Ottawa. There are 14 people at that installation whose major job is to keep up to date with the various releases of the IBM operating systems to insure that their own programs will continue to run.

OS/360 is the recognized giant in the field of operating systems. An early version released in 1966 consisted of over 1,000 separate modules containing nearly 400,000 source statements, a majority of which were in assembler language. It exhibits all the problems of large-scale software development, not the least of which being the number of errors which passed through the development stages of the

system. The release in 1968 of one version of OS/360 was intended to introduce 16 changes and to correct 1074 errors. The system was designed as an all-singing all-dancing operating system which would support all types of programming. Unfortunately, the design phase of OS/360 did not take into account feedback from the user community, with the result that the final product was not very satisfactory.

A project contemporary with the development of OS/360 was the development of the MULTICS System for the GE 645 at M.I.T. Nearly all of the MULTICS System was written in a subset of the higher level language PL/1. The advantages of coding in a high-level language lie in increased programmer productivity, fewer errors in the code, fewer programmers required, increased flexibility in the product, readability of the source code – in other words, self-documentation – and some degree of machine independence. However, there are inevitable penalties in size and performance of compiled code. The early versions of MULTICS were large and slow. Major improvements were made by improving the PL/1 compiler, by capitalizing on experienced programmers' ability to produce PL/1 code which compiles for fast execution, and by changing strategies in the system to optimize its performance. Some idea of the magnitude of the improvements achieved can be obtained from the fact that the overall system was at one time well over 1,000,000 words which was reduced to 300,000 by the methods described. The improvement in performance of modules in the system ranged between 4:1 and 50:1 at a cost of less than two man years work. These figures indicate that by the use of higher level languages major changes to software can be made without a massive personnel effort.

This flexibility is very important for software incorporating new concepts since initial versions must undergo evaluation to reach a satisfactory state. Although the experience with MULTICS is encouraging, very few large operating systems have been written using a higher level language. Project managers try to minixize the core store used and maximize the speed of operation, ignoring advantages such as portability and ease of recoding, which accrue from the use of standardized higher level languages.

Applications Software

The design goal for an operating system normally does not relate to any particular program which is run on that system but rather to the overall operating efficiency of the total installation. Thus, an operating system often falls into the category "Jack of all programs, master of none." In trying to satisfy too many criteria the system falls short of satisfying any of them. Iterations of the design of the system lead to new releases which can effectively change characteristics of the computer. Applications programs written to run on the computer are effectively being written for a soft hardware interface which can change during the life of the machine. This is a disconcerting experience for anyone programming in assembly language or trying to maintain an assembly language program on such a system.

In many ways the writer of operating system software has an easier task than the programmer who has to produce a solution to a real-world problem. He is writing software between two interfaces – one which is given to him and is relatively fixed, i.e., the hardware, and one which he has some leeway in specifying, i.e., the interface with the application programs. If, for some technological reason, he cannot reach his main design goal then subgoals are usually satisfactory. On the other hand, the applications programmer is dealing with the man-machine interface and his work epitomizes the tip of the computing iceberg. His programs are visible and must offset all the vagaries of the underlying hardware and operating system to produce the desired result. Usually it is in his programs that all radical changes take place. It is here that the truth of the design philosophy of his particular applications program or system is shown.

In systems of this type, although the interface between the applications program and the computer is "soft," for most effective results it should normally be considered "hard." In other words, application programmers should not interfere with the operating system even though a modification might favourably affect their programs. The boundary layer between the operating system and an applications program should never become turbulent. Despite this it has become axiomatic that the operating system supplied by a manufacturer will not be satisfactory and will have to be modified by the user. This leads to situations in which major programming effort is required just to keep up with the various issues of an operating system so that the user-defined modifications remain effective. Unless an installation is dedicated to one particular type of applications programming system, such modifications are hard to justify.

Programmers working on a computer which is not dedicated to a single applications area normally have little say in the effective life of the computer. They cannot assume that they will be programming indefinitely for the same computer or the same computer type. Programs for long-term projects may end up being run on various machines during the lifetime of the project. In such situations re-programming becomes a very definite problem. Programs written in lower-level languages will almost certainly have to be re-written for a new machine. Hopefully, programs written in a higher-level language will be able to run with little modification, providing that the new computer supports that particular language. Even if it does not, it is often easier to write a compiler for a language than to make all the changes to all the application programs so that they may continue to be used.

Time Sharing Services

With the large-scale introduction of time-sharing services in Canada, computing at the end of a telephone line has become a reality. Within the Canadian Defence Research Board a terminal language APL is gaining fast acceptance, especially at the Defence Research Establishment Valcartier which runs an APL service on its in-house Sigma-7, servicing 25 terminals throughout the establishment. This service is also used, via telephone lines, by other areas in the Board. Unlike Fortran or Cobol programs, an APL

program is not compiled but is executed line by line from the source language statements. However, the operations allowed in the language are so powerful that for a large class of problems this is not considered a difficulty. Moreover, the fact that statements can be executed as they are written enables one to test algorithms before they are embedded in a final program. The introduction and acceptance of APL not only gives the programmer a language which is good for a particular class of problems but also gives him a major diagnostic tool for other program development.

The advent of time-sharing and of terminal operation has been of great psychological benefit to the programmer. For years he had been forced off the machine and had been programming only on paper. He would post the paper to a "letter box" and eventually receive a reply which would be the result of a development run. He would be effectively shielded from the computer and have no rapport with it. Terminals changed all this. A programmer can now sit at a terminal and do a development run, modify his program then continue to do development. There has been much discussion as to whether this is an effective way to use computers, but there is no doubt that psychologically it is a more effective of using people. The programmer who is working on a terminal has come full circle from the early 1950's and is again tangibly associated with his computer. However, whereas in the early days the computer would support one programmer using a language such as Transcode, today on an APL machine there may be as many as one hundred different people independently operating at the same time.

Dedicated Computer System

Up to now we have been talking mostly about the general purpose computer in which the central hardware is immersed in an ocean of operating software to produce a viable system for the production of different categories of application programs and systems. We have yet to discuss the dedicated computer system in which a computer is used for one purpose only and as such is equivalent to special purpose hardware. In a defence environment such a system would be used to control or monitor the operation of some external equipment such as a rocket launcher, radar or digital sonar. In such systems the computer loses its identity and becomes a control module in some operational environment. Although software for such a system parallels the software for a general purpose systems, and the same design decisions have to be made, the integration of the operating and applications levels of software is far greater. In fact, in some systems both levels merge into a specialized operating system.

Often the design, development and production stages of a system will overlap, not only due to the pressure of a completion schedule, but also due to the actual introduction of the computer itself. If a computer does not influence the operating environment into which it is placed the computer is not required. However, the effect of the computer cannot be gauged accurately prior to its insertion. Thus, the initial program design will probably not represent the final system. The programs written must be flexible and capable of being changed according to new conditions. Here again is a situation

where the use of higher-level languages is of great benefit. Despite this, most such systems are programmed in an assembler language regardless of the relative rigidity which this imposed. The standard complaint is that higher-level languages do not get close enough to the machine itself for control situations, e.g., Fortran is not very good at manipulating bit registers. The real complaint should be that the computer when bought did not come with the proper kind of higher-level language for the problem at hand. There have been very few good higher-level control languages designed. This is not because higher-level languages are bad for control but rather that the solution of digital control problems has been an individual affair. Project managers have tended to be parochial in their outlook, considering that their computer would be the only computer in the control system for the life of their systems, and generating their programs accordingly, usually in machine language. They never considered that the operations they were doing might have to be repeated at some later date, or that perhaps the problem they were solving was not the correct one, or that the programs written might have to be responsive to differing requirements based on experience with the system. Surely, such a problem does not have to be very large before it becomes worthwhile to spend time in designing and developing one's own higher-level language in which to write the control programs; but it is not often done.

Future Developments

Computer hardware is developed in a multi-billion dollar industrial complex which is based on fundamental research in all the physical sciences. A major part of its technology is not dedicated to the computer alone but is general to all the fields of electronics. The synthesis of a computer is based on the components industry, which industry supplies elements for all types of electronic systems. Because of the broad industrial base and particularly because of the components industry, the development of computer hardware can take place at a relatively great pace. Moreover, since the cost of research and development can be spread over many computers and since there is a relatively high capital investment in each computer, a considerable amount of money can be made available for research. Complex hardware systems can be synthesized with relative ease. However, a hardware system is a dead system. It requires software to bring it to life. Software research and development is not founded on a multi-billion dollar broadly-based industry. There are very few components that one can buy off the shelf to put into an operating system. There are a few tools such as higher-level languages, diagnostic programmes, etc., but very few components. Subsystems developed at IBM are highly unlikely to be of use to ICL or to any other manufacturer. This is not the case with hardware where integrated circuits of the right characteristics are available to any manufacturer who can find the correct page in the appropriate catalogue. Further, software technology is not based on fundamental physical laws but rather on a very constrained type of applied mathematics, which does not land itself to revolutionary discoveries.

There is general agreement that software engineering is in a very rudimentary stage of development as compared with the established branches of engineering. In fact, it has been stated that software production today appears in the scale of industralization somewhere below the more backward construction industries. Despite this, there are extremely strong economic or political pressures to produce complex systems which on the face of it are not necessarily revolutionary but when brought down to the software level involve major efforts in research without any guarantee of success. Professor Stanley Gill from the Imperial College of London summarized these points very well when he stated:

"It is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of the technology unless the very considerable risks involved can be tolerated. We can see no swift and sure way to improve the technology and would view any claims to achieve this with extreme caution. We believe that the only way ahead lies through the steady development of the best existing techniques."

Over the years, we have developed many tools for the design and manufacture of software. These tools range from the concept of the flow diagram, which is nothing more than a graphical statement of the logic of the program, to higher level languages which permit that flow diagram to be transposed onto the computer to form an operating program. Programmers are loathe to build on the works of others, preferring to "re-invent the wheel" to build up components with which to synthesize their programs. There is no concept of a components industry in which subsystems with given terminal behaviour could be acquired through a catalogue. There are user groups for various manufacturers' equipment from which specialized software packages can be acquired. However, these will work only with specific configurations of equipment and could not in any way be considered universal.

For the successful implementation of a software components industry, programming must become less of an art and more of an engineering discipline. Some work has taken place along these lines but there is considerable work remaining. One of the most important areas is that of "Standardization." This involves (among other things), standardization of flow diagram techniques, standardization of documentation. Without documentation there can be no communication, and without communication we are right back in the days of the cottage industry. The challenge is there; it is up to us to meet it.

4101 U 420
4102 U
4103 U

0201 DCIEM
0202 406986
0203 (RP)-900

UNCLASSIFIED

Security Classification

PRECEDING PAGE BLANK

DOCUMENT CONTROL DATA – R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATING ACTIVITY
0204a
Defence and Civil Institute of Environmental Medicine
0204b Downsview ONT (CAN)

2a. DOCUMENT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. DOCUMENT TITLE
04a
Problems in software development

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
08a Research paper

5. AUTHOR(S) (Last name, first name, middle initial)
1101
TAYLOR, V.K.

07 CAN

6. DOCUMENT DATE
46 APRIL 1973

7a. TOTAL NO. OF PAGES
0901 9

7b. NO. OF REFS
0902 nil 0

8a. PROJECT OR GRANT NO.

8b. CONTRACT NO.

9a. ORIGINATOR'S DOCUMENT NUMBER(S)

9b. OTHER DOCUMENT NO.(S) (Any other numbers that may be assigned this document)

10. DISTRIBUTION STATEMENT

11. SUPPLEMENTARY NOTES

12. SPONSORING ACTIVITY

13. ABSTRACT

50
In the early days of electronic digital computers, development of software was confined to the development of individual applications programs written in machine code. Since then, as computers have become faster and more powerful, the central computer has been viewed through ever increasing layers of systems software to such an extent that now the architecture of the central computer is often lost from view. The investment in software for a new system may be as great as the investment in hardware. However, the problems involved in the software development may be considerably greater. To develop larger software systems, better tools are required. This paper attempts to highlight tools which are available and to indicate problems inherent in their use. Some speculative ideas are presented concerning the direction in which software research might be oriented to enable us to overcome current difficulties.

DSIS
70-070

DC

UNCLASSIFIED

Security Classification

KEY WORDS
Computing history Software Monitor systems Compilers

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the organization issuing the document.

2a. DOCUMENT SECURITY CLASSIFICATION: Enter the overall security classification of the document including special warning terms whenever applicable.

2b. GROUP: Enter security reclassification group number. The three groups are defined in Appendix 'M' of the DRB Security Regulations.

3. DOCUMENT TITLE: Enter the complete document title in all capital letters. Titles in all cases should be unclassified. If a sufficiently descriptive title cannot be selected without classification, show title classification with the usual one-capital-letter abbreviation in parentheses immediately following the title.

4. DESCRIPTIVE NOTES: Enter the category of document, e.g. technical report, technical note or technical letter. If appropriate, enter the type of document, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the document. Enter last name, first name, middle initial. If military, show rank. The name of the principal author is an absolute minimum requirement.

6. DOCUMENT DATE: Enter the date (month, year) of Establishment approval for publication of the document.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the document.

8a. PROJECT OR GRANT NUMBER: If appropriate, enter the applicable research and development project or grant number under which the document was written.

8b. CONTRACT NUMBER: If appropriate, enter the applicable number under which the document was written.

9a. ORIGINATOR'S DOCUMENT NUMBER(S): Enter the official document number by which the document will be identified and controlled by the originating activity. This number must be unique to this document.

9b. OTHER DOCUMENT NUMBER(S): If the document has been assigned any other document numbers (either by the originator or by the sponsor), also enter this number(s).

10. DISTRIBUTION STATEMENT: Enter any limitations on further dissemination of the document, other than those imposed by security classification, using standard statements such as:

    (1) "Qualified requesters may obtain copies of this document from their defence documentation center."

    (2) "Announcement and dissemination of this document is not authorized without prior approval from originating activity."

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document, even though it may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall end with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (TS), (S), (C), (R), or (U).

    The length of the abstract should be limited to 20 single-spaced standard typewritten lines; 7½ inches long.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a document and could be helpful in cataloging the document. Key words should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context.