UNCLASSIFIED

AD NUMBER
AD844713
LIMITATION CHANGES
TO: **Approved for public release; distribution is unlimited.**
FROM: **Distribution authorized to DoD only; Administrative/Operational Use; OCT 1968. Other requests shall be referred to Rome Air Development Center, Griffiss AFB, NY.**
AUTHORITY
RADC ltr 3 May 1979

THIS PAGE IS UNCLASSIFIED

THIS REPORT HAS BEEN DELI. ITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTICN STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

AD844713

RADC-TR-68-367
Final Report

AN INVESTIGATION OF ADVANCED PROGRAMMING TECHNIQUES

Richard M. Dobkin
et al

System Development Corporation

TECHNICAL REPORT NO. RADC-TR-68-367
October 1968

Each transmittal of this document
outside the Department of Defense
must have prior approval of RADC
(EMIIF), GAFB, N.Y.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

When US Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded, by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

AN INVESTIGATION OF ADVANCED PROGRAMMING TECHNIQUES

Richard M. Dobkin
et al

System Development Corporation

Each transmittal of this document
outside the Department of Defense
must have prior approval of RADC
(EMIIF), GAFB, N.Y. 13440.

FOREWORD

This technical report, consisting of two parts, was prepared by the Rome Operations Branch of the System Development Corp., Santa Monica, California, under contract F30602-67-C-0321, Project 5581. Participating in the study were Richard M. Dobkins (Project Leader), Nathan Adleman, and Guy Wiley of the Rome Operations Branch, and Leah Fine and Joseph H. Yott of the Technology Directorate. The work was performed from July 1967 to June 1968. The RADC Project Engineer is Mr. Richard M. Motto, EMIIF.

Part I is the study to investigate a JOVIAL compiler capable of interfacing with the MULTICS system and operating on the GE-645 computer, and to evaluate the advantages and disadvantages of incorporating certain features in such a compiler. Part II is the study to investigate the development of compiler techniques for generating code that operates effectively in the GE-645 Paging System.

The distribution of this document is limited under the U.S. Mutual Security Acts of 1949.

This report has been reviewed and is approved.

Approved: Francis M. Kelly
for JAMES G. MCGINNIS, Major, USAF
Chief, Info Processing Branch
Intel & Info Processing Division

Approved: A. E. Stoll
A. E. STOLL, Colonel, USAF
Chief, Intel and Info Processing Division

FOR THE COMMANDER Irving J. Gabelman
IRVING J. GABELMAN
Chief, Advanced Studies Group

ABSTRACT

The objective of Part I of the study described in this document was to perform two services. The first service was to investigate four existing JOVIAL compilers to determine which had the greatest potential for conversion to the GE-645 computer. The four compilers were the ones currently in operation on the CDC-1604B, the IBM 7090, the IBM 360, and the GE-635. The second service was to investigate and evaluate the advantages and disadvantages of incorporating certain features into a compiler which would operate on the GE-645 under the control of the MULTICS supervisor. These features included the production of programs with reentrant code, on-line compiling, partial compilation capabilities, string processing, advanced system and program compool features, on-line debugging aids, segmentation, and binary versus symbolic output. The study and the conclusions reached were made by comparing and evaluating the needs of the compiler with available system procedure and interface modules. In regard to an existing compiler, it was recommended that the CDC-1604B JOVIAL compiler be selected for conversion to the GE-645. As to the features to be incorporated, it was recommended that all the features be implemented with the following exceptions: that there be no batch compilation capability; only a limited partial compilation capability be made available; there should be no string processing in the initial version; only a small number of the on-line debugging aids be initially available; and that only binary output be produced.

The purpose of Part II of the study described in this document was to investigate the concept of Paging for the purpose of establishing techniques for generating code that operates effectively in the GE-645 Paging System. There were two major objectives of this investigation. The first was to determine if the code generation process for paging could be automatic (handled by software) or if present programming techniques should be altered to produce efficient code generation. The second objective was to define an implementation approach which will allow a rapid implementation of a Paged JOVIAL Compiler and the transfer of existing JOVIAL programs to the GE-645. The study was accomplished by making an investigation of which language types would be most efficient for paging, and how a program can be structured for most effective operation in a paged environment. The conclusions reached during this study are detailed in Sections II and III of Part II of this report.

PRECEDING PAGE BLANK-NOT FILMED

PART I - TABLE OF CONTENTS

			Page
Section	I.	Introduction	1
Section	II.	The Compiler to Transfer	3
Section	III.	Segmentation	7
Section	IV.	Reentrancy	18
Section	V.	System Compool Capability	22
Section	VI.	Program Compool	26
Section	VII.	Debugging Aids	28
Section	VIII.	On-Line Compiling	36
Section	IX.	Partial Compilation Capability	39
Section	X.	String Processing	41
Section	XI.	Binary Versus Symbolic Object Code	46
Appendix		J3 Language Forms Not Found In JB(Basic)	49

PART II - TABLE OF CONTENTS

		Page
Section I.	Introduction	51
Section II.	Paging Optimization Features for an Object Program	
1.0	Introduction	52
2.0	Organization of Object Code	52
2.1	Compiler Requirements	52
2.2	Compiler Restrictions	53
3.0	Structuring Techniques	53
3.1	Output Classification	53
3.2	High-Activity Area	53
3.3	Table Divisions by Data Classifications	54
3.4	Multiple Store of Invariant Data	54
3.5	Assign Invariant Data With Instructions	54
3.6	Block Invariant Data	54
3.7	Separable Set-Variables	54
3.8	Storage Overlays for Set-Variables	54
3.9	Named Temporary Set-Variables	54
3.10	Correlated Set-Variable Assignments	55
3.11	Parallel Subtables	55
3.12	Serial Tables	55
3.13	Instruction Assignment Order	55
3.14	Instruction Execution Order	55
3.15	Statement Elimination	55
3.16	Loop Data Assignment	56
3.17	Loop Instruction Alignment	56
3.18	Jump Simplification	56
3.19	Page Utilization	56
3.20	Advise Executive	56
4.0	Related Gains	56
5.0	Conclusions	57
Section III.	Compiler Processes to Produce Page-Oriented Programs	
1.0	Introduction	58
2.0	General Assumptions	58
2.1	Input Requirements	59
2.2	Execution Order of Instruction	59
2.3	Freedom in Albegraic Evaluations	59
2.4	Data Considerations	60
3.0	Optimizing Algorithm	61
3.1	Lists	61
3.1.1	Intermediate Language Program	61
3.1.2	Dictionary	61
3.1.3	Abridged Dictionary	62
3.1.4	Dictionary Trace	62
3.1.5	Segment List	62

PART II - TABLE OF CONTENTS (Continued)

3.1.6	Connector List	62
3.1.7	Accession List	63
3.1.8	Invariant Value Use List	63
3.1.9	Block Access List	63
3.2	Processing Steps	63
3.2.1	Process Dictionary	65
3.2.2	Process Intermediate Language Program	65
3.2.3	Process Accession Data	65
3.2.4	Consolidate Program Intelligence	65
3.2.5	Program Relocations	66
3.2.6	Dictionary Realignment	66
3.2.7	Generate Code	67
3.3	Compiler Outputs	67
3.3.1	Optimized Computer Programs	67
3.3.2	Compiler Printed Output	67
4.0	Proposed Procedures	68
4.1	Extract Dictionary Information	68
4.2	Extract Intermediate Language Information	69
4.3	Process Accession List	70
4.3.1	Establish Flow Connectors	70
4.3.2	Invariant Data	71
4.3.3	Extraneous Named Values	71
4.4	Consolidate Program Intelligence	71
4.4.1	Compile Time Executions	71
4.4.2	Consolidate Program Intelligence	72
4.4.3	Reported Intelligence	72
4.5	Construct Processing Blocks	73
4.6	Construct Decision Blocks	74
4.7	Move Program Segments	75
4.7.1	Restrictions on Moving Statements	76
4.7.2	Criteria for Moving Statements	76
4.7.3	Decision Block Intelligence	77
4.7.4	Decision Block Statement Moves	78
4.8	Factor Common Expressions	78
4.3.1	Processing Block Analysis	79
4.8.2	Decision Block Analysis	79
4.9	Dictionary Realignment	80
4.9.1	Divide Subscripted Data	80
4.9.2	Group Variable Value	81
4.9.3	Group Invariant Data	81
4.10	Computer Code Generation	81
5.0	Summary	82

PRECEDING PAGE BLANK-NOT FILMED

PART I - COMPILER ANALYSIS
SECTION I

INTRODUCTION

The purpose of this report is to present the results of a study which had two major objectives. The first objective was to investigate the possibilities of developing a JOVIAL compiler for the GE-645 computer which would interface with the MULTICS (Multiplexed Information and Computing Service) system. This service included an evaluation of the following items to determine which had the greatest potential:

1. Adapting the GE-635 JOVIAL Compiler to the GE-645.

2. Transferring the CDC-1604B JOVIAL Compiler to the GE-645.

3. Transferring the IBM 7090 JOVIAL Compiler to the GE-645.

4. Transferring the IBM 360 JOVIAL Compiler to the GE-645.

This objective was met by making an analysis of each compiler. The major areas considered for each compiler were: the reliability of the code-producing capability of the compiler, which took into account the age of the compiler, the amount of use since its inception, and the quantity and quality of the maintenance activity; the ease with which the compiler could be converted to the GE-645; the deviation of the compiler from official compiler specifications; and the amount of available documentation, such as program descriptions, maintenance manuals, and user guides.

The second objective was to evaluate the advantages and disadvantages of incorporating the following features into the compiler:

1. Reentrant Code

2. On-Line Compiling

3. Partial Compilation Capabilities

4. String Processing

5. Advanced Compool Features

6. Program Compool

7. Segmentation

8. Compiler Producing Reentrant Programs

9. On-Line Debugging Aids

10. Binary versus Symbolic Output

This second objective was accomplished by first making as complete a study as possible of the areas of MULTICS which were deemed most pertinent to the subject. It was then determined which features could and should be implemented, taking into consideration the costs and time required measured against the expected usefulness of the feature.

In regard to any question arising concerning compiler specifications, the basis used was AFM 100-24, "Standard Computer Programming Language for Air Force Command and Control Systems."

The following sections of the report present the analysis of each item considered during the study, the conclusions reached, and the recommendations made.

SECTION II

THE COMPILER TO TRANSFER

1.0 INTRODUCTION

This section attempts to outline the reasoning and justification behind our selection of the JOVIAL compiler most suitable for transfer to the GE-645 computer under the MULTICS system. What follows is a discussion of each of the compilers under surveillance with appropriate information concerning both their positive and negative facets.

2.0 GE-635 JOVIAL COMPILER

Supplied by the General Electric Company for the Rome Air Development Center, this compiler was first available for limited use in September 1966. While the full JOVIAL J3 language is not completely implemented, those forms missing are not of enough significance to allow their absence, of itself, to cause this compiler's elimination from consideration.

The 635 compiler is not written in JOVIAL, but in a macro-language called POPs (program operators). Thus, the compiler cannot compile itself. A variation of the POPs assembler operating on the 645 and producing code for the 645 which would operate under MULTICS would be an additional requirement for the transfer of this compiler. There is probably no need to elaborate on the advantages of maintaining or modifying a compiler written in JOVIAL as opposed to one written in an assembly or macro-language.

Compiler maintenance or program description documentation is not available with the 635 compiler.

The 635 and 645 instruction sets are very similar, and therefore, it might seem very easy to transfer the 635 compiler to the 645 machine. The differences, however, are of enough significance to have great effect on a compiler transfer operation. For one thing, there are a significant number of conventions that MULTICS demands an object program follow. Segmentation, linking, and base registers are features that are not accounted for in the 635 compiler. In addition, the compiler generates code which calculates addresses and dynamically sets them into instructions. This is not permissible if pure reentrant programs are to be produced. Investigation of the code generated by the 635 compiler has led us to believe that there is probably little point to trying to salvage any of its code-generating philosophy. There is, in fact, an amazing lack of similarity between a 635 running under GECOS and a 645 running under MULTICS.

Because of its very limited usage, the 635 compiler has not had the chance that many other compilers have had to have errors corrected. Compiler builders themselves can write test cases and discover just so many bugs in their program. It takes years of usage before some very significant errors may be corrected (see discussion of 1604B compiler below) in the standard JOVIAL compiler.

The implementation of a compool capability in the 635 compiler is not compatible with the accepted definition of a compool. Rather than a formatted, ordered set of descriptions, the compool is a set of data declarations which are essentially recompiled every time the compool is referenced. In addition, the programmer is burdened with the responsibility of supplying the compiler with a list of compool identifiers which his program will manipulate. There is no provision for specifying addresses in the compool. The compool approach recommended in Section V has so little in common with the approach used in the 635 compiler that nothing could be salvaged from that compiler in this area.

The error analysis achieved by the 635 compiler is found wanting in a number of respects. While formulated on a sound basis and extremely well-intentioned, the implementation has unfortunately resulted in non-self-explanatory messages and gaps of up to fifty lines between a statement in error and the corresponding messages. There are approximately fifty different error messages output by the 635 compiler, as compared to more than 225 output by the 1604B compiler.

3.0 IBM 360/50 JOVIAL (J5.2) COMPILER

Produced by SDC for the Defense Communications Agency, this compiler was adapted from the 360/50 Basic JOVIAL compiler written by SDC for its own use in 1965. The language processed by this compiler is a subset of JOVIAL (J3) known as Basic JOVIAL (or JB) with extensions (e.g., fixed-point arithmetic). The differences between JOVIAL (J3) and JOVIAL (JB) are outlined in the Appendix.

The 360 compiler is written in JOVIAL.

There exists a very good program description document for the generator portion of the compiler. The generator is that part of the compiler which is of primary interest as the translator is dependent on the machine for which it is generating code.

The 360 compiler has had a fairly extensive public exposure and is probably in reasonably good shape as far as JOVIAL compilers go. The errors that will appear with this generator will not only be infrequent, but of the sort which will be very easy to circumvent.

The major problems with the 360 compiler are its ability to accept only those language forms defined in JB and the intermediate language (IL) which it uses. The IL is very simple and straightforward, making it fairly easy for the translator to generate code. It is, however, very difficult for the translator to generate good, efficient code. The IL was just not designed with this in mind. The conversion of this JB generator into a J3 generator would involve changes and/or additions to the IL of a non-trivial nature.

4.0 IBM 7090 JOVIAL (J2) COMPILER

Produced by SDC for SACCS under System 465L in 1959, this was the first JOVIAL compiler. It processes a dialect of JOVIAL known as J2. Maintenance of this compiler was stopped in early 1965.

The 7090 compiler is written in JOVIAL.

There is no existing program description or maintenance document for this compiler.

While this compiler has had rather extensive exposure it has not suffered any radical changes to its basic operation. Thus, it could not take advantage of any new compiler techniques and processes developed since 1959. The cessation of maintenance in 1965 means, of course, that no errors discovered since that time have been corrected.

The IL generated by this compiler has been termed "horribly complex" by one of its former maintainers. As no documentation exists, it was not judged feasible to attempt to reinforce this opinion.

5.0 CDC 1604B JOVIAL (J3) COMPILER

Produced by SDC for the NAVCOSSACT CDC 1604A in 1965, this compiler has been successfully transferred to, and is presently operating on, the 1604B at RADC. Despite its designation as a JOVIAL (J3) compiler, it does not completely process some of the more exotic J3 language forms. The generator portion, however, does have the capability of handling the complete J3 language; the generator <u>is</u> the part we are interested in.

The 1604B compiler is written in JOVIAL.

There exists a very good program description document for most of the generator portion of the compiler.

This compiler has had constant maintenance, updating, and improvement since its inception. In addition to its use at NAVCOSSACT and RADC, it is being used at FOCCPAC, FOCCLANT and was transferred to the CDC 3600 for usage by the AFSC Space Systems Division. To illustrate the importance of widespread usage of a compiler in order to achieve proper checkout, it can be noted that since mid-1963, 190 errors have been officially documented as being corrected. It is quite probable that another 190 errors have been corrected without documentation. Any errors that appear in the future with this generator will not only be infrequent, but of the easily circumventable sort. It's time and users that produce good compilers.

The 1604B compiler contains what amounts to a separate program which produces its IL. This second "phase" of the generator produces an IL which is much more complex than that of the 360 JB compiler, but one which is much easier to handle, from the translator's point of view, in order to generate good, efficient code.

The ability of the 1604B compiler to handle compool references without a prior list of compool variables should make the implementation of the recommended compool capability in the 645 compiler an easier task than it would be using the 635 compiler.

One major disadvantage of the 1604B compiler is its size, i.e., it is significantly larger than any of the other compilers considered.

6.0 SUMMARY

We are recommending that a 645 JOVIAL (J3) compiler be produced by starting with the 1604B JOVIAL (J3) generator and adding a newly written 645 translator onto it for the following major reasons:

A. Of the compilers studied, the 1604B is the only one which contains a source language scanner and processor which will accept the full J3 language.

B. The 1604B produces an intermediate language which is the best suited for producing optimal code. The object code generated by the 635 compiler is incompatible enough with the 645 machine and the MULTICS system to eliminate its code generation phase from consideration. Hence, a new code generator has to be written in any case.

C. Adequate documentation is presently available for only the 1604B and 360 generators.

D. The constant usage and maintenance which the 1604B compiler has undergone makes it probably the best checked-out of those compilers studied.

E. The error analysis and detection done by the 1604B compiler is easily the best of these compilers.

F. The 1604B compiler was produced recently enough to take advantage of new compilation techniques, yet is old enough to have had as much or more usage than the other three compilers.

G. Despite its larger size and the extra time needed to compile it, the 1604B compiler is relatively as efficient in terms of its own operating speed as the other compilers considered.

SECTION III

SEGMENTATION

1.0 INTRODUCTION

Fundamental and essential to the operation of MULTICS is the division of memory space into segments. A MULTICS segment is a portion of the virtual memory space which may be addressed by name. It is of little use to ask whether programs which will operate under MULTICS should be segmented. Programs which ignore the MULTICS segmentation conventions will not work. However, it is of great use to consider what is the best way to use the segmentation facilities of MULTICS. The preliminary documentation of MULTICS on which this study is based indicates that minor variations in the use of segments will have substantial effects on the quality of the programs produced by the JOVIAL compiler.

The remainder of this section describes some of the characteristics of MULTICS segments and draws certain conclusions about how they should be used. In addition, a final paragraph offering a possible solution to the problem of the segmentation of the compiled object program is included. The objective of these segment usage policies is to combine JOVIAL and MULTICS in a way that will preserve the most important capabilities of both.

2.0 SEGMENT DESCRIPTION

The following subparagraphs describe things which must be specified about a segment in MULTICS and the facilities which the JOVIAL user will need for this specification. The following will have to be specified:

Name of the Segment

Access List

Parent Directory

Allocation Mode

Structure of the Segment

Contents of the Segment

The following discussion will concentrate on segments which contain data but there will be a few remarks about segments which contain instructions.

2.1 Segment Name

The syntax of names in MULTICS seems to be more complicated than that of JOVIAL. It would be possible to restrict segment names which are used only within the JOVIAL subsystem to the JOVIAL syntax but there is a need for JOVIAL users to refer to many of the MULTICS modules. Therefore, it will be

necessary to provide some capability to handle these names which seem to contain periods, dollar signs, and underline characters as well as the normal JOVIAL letters and numbers. One approach which can be used is to write the segment name as a JOVIAL Hollerith constant. This is a bit awkward but a neater method cannot be chosen until a complete description of the MULTICS name syntax is available.

2.2 Access List

The access list of a segment describes which processes or users may use the segment and how they may use it. It seems likely that the specification of the access list for a segment created by an explicit user command can be done outside of JOVIAL by system commands or by calls to system procedures. However, there will be some segments which are automatically created by the JOVIAL system. The conventions about these segments must be consistent with the rest of MULTICS and with the normal behavior of JOVIAL systems.

2.3 Parent Directory

In MULTICS each segment "belongs to" a directory (its parent directory) which determines the scope of the name of that segment. There may be many segments in the system with the same name as long as they each belong to a different directory. Specification of the parent directory will often be implicit but facilities for explicit specification should be provided for compool input and possibly for segment declarations within a program.

2.4 Segment Allocation Mode

The specification of allocation made determines when and by whom the creation and destruction of the segment will be triggered. A data base retrieval program provides an example of the requirements for flexibility and convenience of allocation. Suppose that we have two or more data bases with similar structure but different data. They all contain personnel data but each for a different department of the same organization. We also wish to allow simultaneous retrieval from each data base by more than one user. The combination of JOVIAL and MULTICS should be well suited to the production and operation of a single reentrant retrieval program which will satisfy this hypothetical requirement. The key to this achievement is providing the proper flexibility of control over data allocation to the JOVIAL user. It should be possible for the user to create segments for each of these data bases all with the same structure declarations (tables and items). The retrieval program must be programmed so that the segment names of the data bases may be variables whose values are not known at compile time. It must be possible to create segments which are private to the process of each user for scratch storage during the operation of his retrieval. Finally, these requirements should be satisfied with a considerable degree of convenience to the programmer and the retrieval user.

Two basic types of segment allocation control can provide the necessary flexibility and convenience:

A. The compiler generates code to automatically allocate space private to each process at the time that process first references the data. In addition, that space is automatically released when the process is destroyed. The only action required of the program writer or user is to specify that the segment has this allocation mode when he declares the segment.

B. The programmer includes a call in his program which causes the creation of the segment when his program is executed and a complementary call for the destruction of the segment. The compiler automatically generates code for the proper addressing of the data in this segment based on the structure specified in the segment declaration. By careful selection of the parent directory, the programmer may have several segments with the same structure and name and still program to access one of these segments without ambiguity. For example, if he makes the process directory of the process executing the create call the parent directory of the created segment, he will have a separate segment private to each process which contains the program with the create call. Similarly, if he uses a directory private to the personnel staff of department ABC as the parent directory of the created segment, then only processes which are run for members of that personnel staff will be able to access that segment.

The programmer should be able to specify the desired allocation mode in his segment declaration.

It is <u>not</u> recommended that allocation mode be separately specified for individual items and tables of JOVIAL data. It is recommended that all JOVIAL data take its allocation mode from the segment to which it belongs. This allows the programmer to declare a lot of minutely specified data within a segment while avoiding unnecessary expansion of the linkage segment. The loss of flexibility is only nominal since he can always put his different pieces of data in different segments if different allocation modes are desired.

2.5 Segment Structure

The usual JOVIAL declarations of items, tables, arrays and strings should be the main method of specifying the structure of a data segment. The structure of a segment of instructions may be partly specified by program or procedure declarations. It will be necessary to have facilities for the programmer to specify which elements are in which segment and in what order they occur in the segment. It is recommended that the JOVIAL overlay declaration be used for the specification of order. The specification of inclusion in a specific segment could be accomplished by an extension of the overlay declaration so that a segment name is a legitimate first term of the overlay's data sequence or by use of BEGIN END brackets to enclose the declarations of the data belonging to the segment. One advantage of the BEGIN END bracket method is that it could also be used to specify which segment a program or procedure belongs to for compool input. The programmer probably should be prohibited from specifying that formal parameters be placed in a specific segment. There probably should not be any facilities for the programmer to specify that some declared data is located at an explicit address of a segment because there are a few hints in the MULTICS documentation that some of the words of the segment

are used for system housekeeping but there does not seem to be any all inclusive statement of which words.

2.6 Segment Contents

It is recommended that the usual JOVIAL presetting facilities be available for program declared data. The actual operation of initialization must occur at the time of segment creation. This will not be the same as compilation time or compool assembly time. Therefore, special system routines for initialization will have to be separate from the compiler and compool assembler. Once the complete mechanism is set up for presetting program declared data the application of this mechanism to data declared in the compool may be quite simple.

3.0 EVENTS IN THE LIFE OF A SEGMENT

The events described below are fundamental to the use of segments in MULTICS:

Declaration

Creation

Attachment to a Process

Linkage to Another Segment

Detachment from a Process

Destruction

Although these are not the only things that can happen to a segment, they are the most significant things which happen to the segment as a whole and they form the basis of some segment classifications which are important to reentrancy. Definitions of these events follow.

3.1 Segment Declaration

The declaration of a segment is the action of filing away a set of descriptive information about the segment in such a way that certain system programs (such as the compiler) can refer to the description with little additional work on the part of the user. Some examples of such descriptions are the descriptions in a JOVIAL compool or the dictionary produced during a compilation. This set of descriptive information may include specifications of the attributes of the declared segment and declarations of the structure and coding of the segment contents. It does not include the body of the segment although it may include a list of initial values which are to be inserted when the segment is created. It is not necessary that the declaration specify all of the attributes of the segment. It is desirable to allow some of the potential descriptive information to remain unspecified until later events (especially until segment creation). The attributes: segment size, parent directory, and various details of the access list are most likely to be left unspecified until segment creation.

The description may be considered as a set of rules which are to be followed by the system when the segment is created, manipulated, or destroyed. This set of rules is similar in function to the set of rules which results from a JOVIAL item or table declaration. It is a command to the JOVIAL system that when manipulating element XYZ, the manipulation is to be consistent with the attributes filed in the declaration of XYZ.

It is recommended that new compool input language forms be designed for declaration of segments (see Section V). In addition, it would be desirable to provide similar language forms for including segment declarations in a JOVIAL program. This would allow a user who has a relatively simple inter-program communication problem to solve it without going through the extra steps of compool assembly.

The determination of what portions of a complete segment description should be included in segment declarations will depend on how the MULTICS system commands for segment creation and directory manipulation are organized. This information is not completely available yet. However, it is clear that some of the elements of the description should be optional and that it should be possible to specify some elements indirectly. For example, it should be possible to specify that the parent directory is to be the process directory of the process in which the create call is executed, even though that process and that process directory are not yet in existence at the time the declaration is processed.

3.2 Segment Creation

Segment creation is initiated by a call to MULTICS routines which build a directory entry describing the segment. This event includes several major operations:

1. Actual construction of the directory entry.

2. Allocation of some space for the body of the segment.

3. Creation of an associated linkage segment and segment symbol table if they are needed.

4. Initializing the value of the body of the segment if necessary.

The specifications necessary for this action (parent directory, size, linkage information, etc.) may have been previously declared and filed away (in a compool) or they may be specified as a part of the call to build the directory entry, or they may be split between the two sources. The JOVIAL compiler (and associated programs) will have to gather the information from declaration and call and provide a complete set of parameters to the MULTICS system routines.

Segments which contain program instructions should be created at compile time. If the programmer is controlling allocation, the segment creation event will occur as his call is executed. In the cases where the segment creation event is automatically controlled by compiler-generated code, it will be necessary to use the traps which MULTICS provides as a part of the linkage process.

3.3 Attaching a Segment to a Process

This includes giving the segment a segment number in this process and appending a copy of the associated linkage segment to the process' linkage segment. This event is handled automatically by the MULTICS routines which get called during linkage. The MULTICS documentation refers to this event as making the segment known to the process. It occurs when the segment is first referenced by the process.

3.4 Linkage to Another Segment

This event includes the modification of the reference-by-name which the compiler or assembler put in the original linkage segment to a reference-by-address. This event may cause the attachment event to occur if this is the first time the referenced segment has been used in this process.

3.5 Detachment of a Segment from a Process

This event includes the destruction of the linkage information which was built during attachment to the process and linkage events but does not include destruction of the body of the detached segment. It seems likely that this event would only be triggered by the death or destruction of the process.

3.6 Segment Destruction

This event is the converse of segment creation. It includes deallocation of space for the body of the segment, the associated linkage segment, and the segment symbol table, loss of the values in those spaces, and removal of the entry for this segment from its parent directory. Like segment creation, this event may be triggered automatically for some segments but there will be similar requirements for manual control for certain applications. In general we can look on certain segments as "belonging to" the process which triggered their creation in the sense that when this process ends its segments are destroyed. However, segments whose creation and destruction are under manual control should be considered to be only "associated with" the processes which triggered their creation or to which they are attached. Such a segment should not be affected by the end of an associated process and may be destroyed only by the explicit command of a user with the proper access authority. The mechanism by which this distinction is controlled is the specification of the proper parent directory for each segment. A segment whose parent directory is the process directory will be automatically destroyed by MULTICS system routines when the process itself is destroyed.

4.0 CONCLUSIONS

The effects of the MULTICS policies and conventions about segmentation enter into all aspects of program production for MULTICS. Successful implementation of any compiler requires an understanding of these policies to about the same extent that compiler production of a compiler for the IBM-7090 computer requires an understanding of its instruction set. The large volume of MULTICS documentation and its changes and growth have precluded the analysis of all

of the segmentation aspects of MULTICS and the GE-645 computer. In particular, the MULTICS system routines by which a user can manipulate directories require further study. The development of detailed plans for the use of these routines will be a necessary part of the compiler design.

The introduction of allocation modes and the addition of segment declarations to the JOVIAL language (at least for compool input) are significant extensions beyond previous JOVIAL implementations. Also, the division of labor between compile time and execute time is significantly different than previous JOVIAL implementations. All of these changes are considered to be important for one of two reasons:

1. Some are necessary to operate at all.

2. Some make it possible to accomplish things for which both JOVIAL and MULTICS are especially well suited, such as the reentrant data base retrieval program described in paragraph 2.4 above.

The relationship between JOVIAL and segmentation should also be considered in the user documentation for the compiler. Some tutorial material about segmentation is expected from the MULTICS implementation project. However, it will probably not be extensive enough or well enough oriented to JOVIAL to satisfy the needs of the JOVIAL users.

5.0 SEGMENTATION OF THE OBJECT PROGRAM

This paragraph is intended to answer some of the questions related to the segmentation of the compiler-produced object program. While specific recommendations are made, these recommendations should not be taken as gospel. There is much room for the compiler implementer to shuffle things around and perhaps evolve a more efficient and/or logical separation.

5.1 Available Segments

Available to the compiler for the deposition of data are the four segments generally associated with any program: the text; linkage; stack; and symbol segments. In addition, it is within the power of the compiler to create any other segment either it (the compiler) or the user discovers to be necessary. This creation may take place at either compile or execution time (via a trap routine).

5.2 The Obvious Choices

There is very little controversy in choosing the text (or procedure) segment to contain the object program's executable instructions. There seems little that can be gained from segmenting this object code any further (e.g., assigning each or a select few procedures to unique segments). Indeed, execution time will be impaired by attempting any such segmentation. Note that some general purpose procedures may be shared by multiple users by compiling them separately into their own segments (see Section IX).

Likewise, it is recommended that the program and compiler-generated constants also be allocated in the text segment. As they can be referenced only from within the text segment and since by their very nature, never change, there is absolutely no point in placing constants elsewhere.

If the user is provided the capability of specifying a unique segment for some of his data (this is the recommendation in Section V), then it is our belief that this data should, in fact, be allocated within that segment.

The symbol segment is a rather obvious choice for the program compool (if one is produced).

The natural place for the linkage pairs with fault type two's necessary to make intersegment references would seem to be the linkage segment associated with the text segment.

Finally, the system's mechanism for calling procedures in other segments (CALL/SAVE/RETURN macros) places the machine conditions on a CALL in the stack segment.

5.3 The No-So-Obvious Choices

Pertinent to the dilemma of where to allocate the object program's data which has not been specified as belonging to a particular segment, are the following facts:

A. To provide a reentrant capability, it is necessary that for each process which calls upon a reentrant program, that some or all of that program's variable data be allocated privately to that process.

B. The linkage segment associated with a particular text segment is automatically copied as an addition onto the process linkage segment for that process which referenced this particular text segment.

C. The invocation of the linker via indirect references to the linkage segment and fault type two's should for efficiency's sake, be held to a minimum.

D. The system requires that a specific base register pair (lb←lp) contain a pointer into the linkage segment itself; likewise, for the stack segment. These base register pairs allow for direct efficient references into the linkage and stack segments.

E. The segments containing the user's data private to this process must be "destroyed" at the completion of the process (see paragraph 3.6 above for discussion of segment destruction).

F. The linkage segment is automatically "destroyed" at process completion time.

The above-outlined facts point to the linkage segment associated with a text segment as the repository for the object program's data which was not

specified as belonging to a particular segment, and this is our recommendation...with one exception: data sets which are "too large" to fit into the linkage segment should be placed elsewhere. The linkage segment is finite in length (256K words) and care must be taken to prevent its overflow as the system presently makes no checks on its size. Since it is literally impossible at compile time to determine the eventual size of a process's linkage segment, it is suggested that some arbitrary size for a piece of data be selected as the cut-off point for allocation in the linkage segment. Any table, array, etc., larger than this number will be assigned to one (or more, if needed) unique segment(s). The determination of how large is large is left to the implementer. It should, of course, be a parameter which could easily be changed as more experience with the system yields more data to work with.

Thus, data which is not specifically assigned to a unique segment by the user and which is not "too big" should be allocated in the linkage segment associated with the object program's text segment.

The external symbol definitions may be allocated in either the text or linkage segment. Since the linkage segment is to contain the program's data (as described above) and since, with one exception, the symbol definitions are constant, we recommend that these definitions be placed in the text segment. The one exception mentioned above is the presence of a trap indicator within the definition. Since this indicator may be changed (i.e., cleared to zero), it is necessary that the symbol definition containing a non-zero trap indicator be placed in its entirety in the linkage segment. (Note: this is done in order to maintain the purity of the text segment, i.e., nothing within text gets modified.) There is provision within the linkage segment for distinct linkage blocks, one of which has pointers into the text segment for symbol definitions; the other of which has pointers into the linkage segment itself. It would seem to be possible for those definitions which contain non-zero trap indicators to be allocated in the linkage segment while all others would be in the text segment. While this is our recommendation, the extra time and effort necessary on the part of the implementer to adhere to it, as opposed to putting all symbol definitions in the same segment, should be considered before a final decision is made on which method to choose.

In order to maintain the integrity of all temporary registers used in the code generated by the compiler, it is necessary to place them in a segment which gets created, not only for each process which uses the particular reentrant program in question, but conceivably for an intersegment function call. To satisfy this requirement and to have maximum efficiency in the accessing of these registers, it is recommended that they be allocated in the stack segment. The sb←sp base register pair always contains the correct current pointer into the stack and provides for an efficient, direct reference to the data within. It is additionally recommended that the stack segment be used to hold the necessary save information during intrasegment procedure calls. This information would include the return address plus any index registers used within the procedure being called.

The last piece of controversial data is the set of preset data or initial values. As these cannot be set into the appropriate data segments until

execution time and as they are constant, it is recommended that they be allocated in either the text or symbol segment. Nothing we have discovered to this point, indicates that either is preferable over the other; thus, we recommend leaving this allocation to the discretion of the implementer.

5.4 Summary

The following chart summarizes our recommendations for the allocation of data in an object program.

Segmentation of the Object Program

Text	Symbol	Link	Stack	Other
Instructions	Program Compool	Linkage pairs (with fault type two)	Temporary Registers	User-specified a particular segment for his data
Constants				
Initial Values*	Initial Values*			
Symbol Definitions (without trap pointers)		User-declared data not specified as belonging to a particular segment and not too "large" to fit here	Saved Registers (at least two for each procedure)	Data area (table, array, etc.) too "large" to fit into link segment
		Symbol Definitions with non-zero trap pointers		

* Either of two segments is workable

SECTION IV

REENTRANCY

1.0 INTRODUCTION

A reentrant program is one that, as a single copy, can be called upon from one process, interrupted, called upon independently from another process, interrupted (and so forth), and independently resumed at each point of interruption. For example, if a compiler is reentrant, only a single copy of the compiler need exist, no matter how many programs are being compiled simultaneously. Large savings in the use of high speed storage and the volume of input/output traffic can be gained by compiling JOVIAL programs to be used with MULTICS as reentrant programs. An important side benefit of a JOVIAL compiler that produces reentrant programs is that the JOVIAL compiler itself will be reentrant since the JOVIAL compiler is used to compile itself.

The production of reentrant programs requires the capability to:

1) produce pure program elements
2) selectively allocate storage
3) efficiently select context

The following sections describe these capabilities and explain how they can be satisfied in the environment of JOVIAL, MULTICS, and the GE-645 computer.

2.0 PURE PROGRAM ELEMENTS

The essential characteristics of a pure program element is that it does not get modified during execution of the program. Thus, it can exist as a single copy in high speed storage even if it is being used by several processes. Furthermore, should storage requirements cause a pure portion of a program to be swapped out of high speed storage, it need not be written out since it has undergone no change and can be reloaded from its original source when it is brought back into core. The pure portion of a program includes instructions that do not get modified during program execution, constant data, and initial values.

By contrast, an impure program element is one that does get modified during execution of the program. Its usage dictates whether a single copy exists for all users or whether each user has his own individual copy. Whenever an impure portion of a program is swapped out of high speed storage, it must be written out so that a copy exists to bring back into core. The impure portion of a program includes variable data and modifiable instructions.

Effective use of reentrancy requires the compiler to generate pure code to the greatest extent possible and to isolate constant data and initial values from variable data and modifiable instructions. The different types of pure information must be concentrated in one or a few contiguous portions of the addressable space in order to minimize the housekeeping needed to distinguish the pure portions from the impure.

3.0 SELECTIVE STORAGE ALLOCATION

Effective use of reentrancy requires two types of storage allocation for program elements, single copy and multiple copy, and two modes of triggering that storage allocation, automatic and controlled. Some program elements, such as the pure portion of a program, must be allocated storage only once if we are to gain anything from reentrancy; whereas, other program elements, such as the compiler-generated temporary storage for intermediate results, loop variables, etc., must be allocated for each using process.

However, it is not always possible for the compiler to determine the type of storage allocation (single or multiple copy) for all data explicitly declared by the programmer. Most of the programmer-declared data should have multiple copy allocation but there are some important applications where there is a clear requirement for multiple users to share a single copy of data. For example, a reentrant data base retrieval program should be able to reference a single copy of the data base while being used by many users. Furthermore, the compiler cannot determine when to allocate storage for program elements it recognizes as single copy elements. Therefore, there must be two modes of triggering storage allocation: automatic, whereby the compiler provides the mechanism, via a linkage trap, to trigger storage allocation for each process that references the program element; and controlled, whereby the user, either within his program or from a console, provides calls to the system programs to trigger storage allocation.

It is necessary to provide the programmer some explicit control over the storage allocation mode used. One way the programmer can indicate whether to use controlled or automatic allocation is to include this information in the Compool. Another way would be to allow the programmer to control the allocation mode of program-defined data by extending the JOVIAL language. At the time storage is allocated, it must also be initialized if any initial values are given in the program. The initialization cannot occur at compile time since the number of copies of automatically allocated data is unknown.

Storage deallocation occurs in two modes as well; all storage that was automatically allocated is automatically deallocated at the death or destruction of the process and all storage that was manually allocated must be manually deallocated.

4.0 EFFICIENT SELECTION OF ENVIRONMENT

Whenever a single copy of a program is shared by different users and multiple copies of data storage are allocated, there must be some medium by which the connection between the single program and the proper environment is maintained. This medium of connection will be used every time an instruction in the program refers to an element which is separately allocated for each user. Efficient operation is so important here that reentrant programming is seldom worthwhile unless this connection can be made through hardware facilities such as index registers or base registers.

The GE-645 computer does offer hardware facilities to expedite efficient selection of environment. MULTICS has established conventions for usage of these hardware facilities and for some standard segment allocation. In particular, the JOVIAL compiler must produce code which references a linkage segment, a stack segment, and perhaps a symbol segment in addition to a text segment that contains the pure portion of the program and other needed segments. The compiler can also take advantage of the fact that one address base register pair always contains a pointer into the linkage segment while another contains a pointer into the stack segment by allocating data to these segments whenever possible (a discussion of data allocation can be found in Section III). The available address base registers should be used judiciously and the number of segments produced for a single program should be kept to a minimum.

5.0 RELATION BETWEEN REENTRANCY AND RECURSION

A recursive program is a program that can call itself or that can be called by a program that it has called or both. It is necessary for a recursive program to be reentrant to prevent a profusion of copies of the same program from overcrowding high speed storage.

However, a recursive program has even more stringent storage allocation requirements than a reentrant program. For a reentrant program, some storage must be automatically allocated for each process that uses the program and deallocated at the death of the process whereas for a recursive program, this storage must be automatically allocated each time the program is entered and deallocated each time the program is exited. When the recursive program calls itself, the previous allocation of multiple copy storage remains in effect but becomes inaccessible and a new generation of multiple copy storage is allocated. When the recursive program returns, the previous allocation again becomes the current one. Thus, in a recursive program, the mechanism for the selection of environment must distinguish between more than one generation of storage belonging to a single user and must connect the program with the youngest generation of storage.

In MULTICS, the natural place to allocate the multiple generations of storage subject to recursion is in the call stack segment. The standard MULTICS calling sequence and the base register pair that conventionally reference the call stack provide a reasonably efficient method of allowing the pure portion of the program to reference the youngest generation of the impure portion.

6.0 SUMMARY AND CONCLUSIONS

The special multiple allocations for storage, characteristic of reentrant and recursive programs, are costly. The design of the GE-645 computer and of MULTICS helps to moderate the extra costs due to reentrancy and recursion, but they do not completely eliminate the extra housekeeping. The extra housekeeping operations due to reentrancy are not too great and are balanced by the savings due to sharing a single copy of common programs. In fact, the savings due to a few heavily used large programs, such as the JOVIAL compiler, would probably justify providing reentrancy as a general capability of all JOVIAL

programs. But the extra housekeeping due to recursion is considerable and can be avoided by allowing the compiler to generate code that takes advantage of the fact that a JOVIAL program cannot be recursive.

SECTION V

A SYSTEM COMPOOL CAPABILITY UNDER MULTICS

1.0 INTRODUCTION

The following paragraphs describe how a system compool capability might be developed for use by a compiler operating under the control of MULTICS. They contain an explanation of the compool concept and list some of the advantages and disadvantages of having a compool capability. Also described is the design of the compool, how it could be constructed, and the methods which might be used in accessing the defined data.

2.0 THE COMPOOL CONCEPT

In building a system of programs coded in JOVIAL, one of the major problems which must be solved is how to provide a communication capability amongst the programs of the system. In JOVIAL, this communication usually means the sharing of data which is maintained at some location accessible by any program in the system. This collection of data descriptions is referred to as the compool.

Contained in the compool are complete descriptions of all data which may be used by more than one program in the system. During the compilation process, if a program references but does not internally define a piece of data, the compiler searches the compool for this data, and if found, generates code using the description and address as defined in the compool. If the data is not found, then it is assumed to be undefined.

The compool is usually built by a program referred to as a compool assembler. This program accepts as input data declarations which for convenience sake should be as near the standard JOVIAL formats as possible. Data which the assembler program should be capable of handling includes tables, table items, simple items, arrays, and constants. Also acceptable as input should be information concerning programs and procedures and their parameters.

A second program is normally included where there is a compool capability. The two major functions of this program are to disassemble the compool into some sort of symbolic output, and to perform an analysis of the compool in which it checks for such discrepancies as duplicate names and unintentional overlaying of data.

3.0 ADVANTAGES OF A COMPOOL CAPABILITY

If two or more programs are to communicate with each other using data as the means of communication, then it is not only mandatory that each program be able to access the data but that each program treat the data in exactly the same manner. Use of a compool provides this capability in that the data are structured in one place, the compool, instead of each individual program. It is therefore possible to change the whole data structure of the system by just the generation of a new compool and the recompilation of the system programs.

Also contained in the compool are the system-dependent constants which, especially in a mathematically-oriented system, will probably need changing during the course of the development of the system. A change in the compool eliminates the need for changing the constants in the possibly hundreds of places they are used throughout the system.

The use of a compool also serves one of the major purposes of MULTICS in that it provides for the sharing of a relatively small area of memory by a possibly large number of users.

Use of the disassembly function can prove to be a valuable tool in the areas of program and system documentation.

4.0 DISADVANTAGES OF A COMPOOL CAPABILITY

The only disadvantage in having a compool feature is the cost involved in building the compool assembler and disassembler, and in implementing the compool capability in the compiler. However, much of this cost can be offset as explained in the following paragraphs.

5.0 BUILDING THE COMPOOL

As a minimum requirement, the compool should contain descriptions and addresses for tables, table items, simple items, arrays, constants, programs, and procedures and their parameters. Data should be input to the assemble function in a symbolic form compatible with the JOVIAL language.

5.1 Compool Assemble Function

The recommended method of building the GE-645 compiler (see Section II) states that the compiler will have two major portions, the generator and the translator. It is recommended that the compool assemble function be incorporated in this generator portion of the compiler. The generator will operate in an assemble compool mode, at the optional request of the user, in which it performs only those operations needed to process a compool deck and to prepare the output which is needed during the compilation of a source program. Since the data descriptions in the compool deck will be very similar to those found in the source deck of a program, advantage could be taken of a great many of the processing features of the compiler, such as table packing, presetting, and overlaying of data. The cost of the assemble function would be greatly reduced and the user would be provided with capabilities, such as table packing and thorough error checking, that might not be found in an ordinary separate assemble program.

The input to the compool assemble function is a deck of data declarations and overlays headed by a control card which should give control information such as the compool name. The data declarations and overlays should be in a format as close to those described in the JOVIAL language specifications as possible. In addition to the data declaration cards, additional control cards may be input which enable the assemble function to organize the data into

groups, or segments. As a minimum requirement, these control cards could have the following format:

SEGMENT name, mode, access list $

The data which are to be segmented by a SEGMENT card should in some way be easily recognized as being controlled by that card. This could be done by enclosing the data within BEGIN-END brackets following the SEGMENT card.

The name field specifies the name of the segment. All data contained in this segment can be addressed by this name and the relative word number within the segment.

The mode field can be either a C for controlled mode or an A for automatic mode. The controlled mode means that the user is responsible for creating the data segment prior to the time he wants to reference it. In the automatic mode, the allocation and the creation of data segments is done for the user (via a trapping subroutine) and needs no extra activity on his part. This creation and allocation will take place at execution time on the occurrence of the initial reference to the data segment. As this automatic allocation will occur for each process using a particular program with a data segment declared to have an automatic mode, that program will satisfy one of the requirements of reentrancy (i.e., that of unique data for each of its users).

The access list is used to control the availability of the compool data to different users. It may contain the actual ID's of potential users or it may be a pointer to some table which contains these ID's.

In the absence of a SEGMENT control card, the assemble function should consider the data as all being in one segment, the name of which can be some derivative of the compool name.

It is recommended that the output of the assemble function be one segment. The name of this segment is the same as the compool name specified on the assemble function control card. Within this compool segment, each data description could be in a format corresponding to that of the Segment Symbol Table. This could possibly make the compool available for use by any MULTICS system procedure which is geared to handle this table. Since this is the same format recommended for the program compool (see Section VI), the system and program compools may, to some degree, be interchangeable.

5.2 Compool Disassemble Function

This is a feature of the compool capability that is recommended but which is not absolutely essential. The disassemble function may be handled by a program separate from the compiler. Its primary function is to translate the contents of the compool into a symbolic output. Concurrent with this activity is a check for possible error conditions in the compool (e.g., inconsistent overlays) which were not detectable during the assemble function.

6.0 ACCESSING COMPOOL-DEFINED DATA

In the compool mode of compilation, the compiler retrieves from the segment containing the compool the information it needs to produce code which will manipulate the data in the way specified by the source statement. Instead of an absolute or relative address, the address of the data is formed by using the segment name and word number within the segment.

The compool is accessed only if the program being compiled references data which is not declared internal to itself. This means that in the case of like names, the program-declared data are used.

7.0 RECOMMENDATIONS

The compool concept is entirely feasible in a compiler operating under MULTICS. It is recommended that, as a minimum requirement, the assemble function be incorporated in the generator portion of the compiler. It is also recommended that the disassemble function be considered as desirable but non-essential.

SECTION VI

PROGRAM COMPOOL

1.0 INTRODUCTION

The program compool contains complete descriptions of all data declared within a source program and all data in the system compool referenced by the program. It also contains information on all labels defined within the program. In addition, there are descriptions of data and labels contained in any procedure compiled into the program. The program compool is optionally produced by the compiler in response to a request by the user.

2.0 STRUCTURING

It is proposed that the program compool be structured in the format of the Segment Symbol Table and that it be placed in the symbol segment. This makes the program compool available for use not only by the debugging tools described in this document but by other MULTICS system procedures as well.

The program compool is built by the compiler during the actual compilation process. In addition to its normal function, the compiler, in response to an optional request by the user, operates in an assemble program compool mode during which time the table containing the program compool information is constructed.

3.0 ADVANTAGES OF A PROGRAM COMPOOL

The availability of a program compool allows a number of activities to be carried on which would not otherwise be available to a user. In the area of program debugging, several tools can be built around the use of a program compool (see Section VII). Test data generation can be accomplished by use of symbolic names (instead of any type of an address) in a language very similar to that used in the source program. Data recording can also be accomplished by specifying data names and locations symbolically. The data may be processed and appropriately formatted entirely dependent upon the compool definitions.

If the recommendations concerning the formats of the system and program compools are followed, then the two compools are compatible. The program compool may be used as input to the compool disassemble/analysis function described in Section V, thus saving the cost of an additional disassemble program. The output thus obtained is a useful tool in the preparation of system and program documentation as well as being an aid in program debugging.

Because of this compatibility, a program compool, once generated, may be used in lieu of a system compool for initial compilation purposes. The construction of system compools may also be assisted by the availability of numerous program compools.

4.0 DISADVANTAGES OF A PROGRAM COMPOOL

The only disadvantage of having a program compool is the implementation cost. However, this could just as easily be considered as an advantage if a system compool capability is to be built into the compiler. The same functions in the generator phase which are used to build the system compool may also be used to build the program compool, thus saving any need for extensive additions to the compiler.

5.0 CONCLUSIONS AND RECOMMENDATIONS

A program compool is an invaluable aid in the use of debugging tools. It can also prove useful in the area of system and program documentation and as a backup to the system compool. Its construction is not forced in unwanted cases due to the optional feature. The implementation cost is unprohibitive.

It is recommended that the construction of a program compool be incorporated in the compiler and that this construction be controlled by an optional request from the user.

SECTION VII

DEBUGGING AIDS

1.0 INTRODUCTION

Debugging aids can be divided into two categories: those which can be incorporated into the compiler and executed as part of the compilation process; and those which are normally operated during the execution of a user's job. Each category can be further divided into two modes of operation, on-line or off-line. The following sections describe these categories and the modes of operation.

2.0 DEBUGGING AIDS FOUND IN THE COMPILER

There are at least four debugging aids which should be incorporated in the compiler. The most important of these four aids is the list of program errors output during the course of the compilation. This list describes errors which are generated by the use of illegal language forms according to the syntax of the language. The other three aids are a set-used listing, in which a record is kept of all references to data, statement labels, procedures, loop variables, and switches; a listing of the compiler's dictionary, which is a symbolic description of each entry in the dictionary; and a listing of the object code produced by the compiler.

2.1 Program Error Listing

The program error list should be divided into two major sections. The first section contains a description of those errors which can be detected during the scanning of the source language statements. The description of the error is output at the time of its detection and as a result, the list of errors is interspersed with the listing of the source deck. In this way, the error can be more easily associated with the source statement containing the error.

The second section contains those errors which cannot be detected until the entire source deck has been scanned or until the object code is being generated. This list is output at the end of the source program listing or the object code listing, or both, depending upon the type of error.

The program error description may be one of two formats. It may contain just an error number which refers the user to a list of possible error conditions or it may be an actual description of the error with some sort of indicator as to where in the statement the error was encountered. In either case, the description should contain a reference to the number of the statement in error.

2.2 Set-Used Listing

The set-used listing is generated by a program operating as an optional phase of the compiler. It may accept as input either the same source program as that used by the compiler or the intermediate language output by

the compiler. The listing is a record of all the references the program has made to tables, table and simple items, statement labels, procedures, functions, closes, loop variables, switches, arrays, and strings. The listing is in a symbolic format and the references are to either a card sequence number on the source card or to a compiler-generated statement number. Also contained in the listing is information concerning: references to undefined data and sequence designators; items declared within a program but never used; ranges for loop variables; and the ranges for all BEGIN-END brackets. At the end of the listing are the totals showing the number of tables, statement labels, etc., declared in the source program and the number of references made.

During the course of its execution, the set-used program may build its own dictionary which is maintained and sorted by data type and which may be output for possible use by other debugging tools. Additional output could include the blocks of reference data which are also maintained by data type and sorted by card sequence number or by statement number.

2.3 Dictionary Listing

Each entry of the compiler's dictionary contains all the information gathered by the compiler about a single entity (e.g., an item, a procedure declaration). The dictionary listing is a symbolic representation of this information. For example, the listing should contain, for a table item, the item name, name of the parent table, item type, word number in the table, starting bit number, and number of bits or bytes.

This listing should be available to the user on an optional basis and should follow the source program listing or the object code listing in the output stream.

2.4 Object Code Listing

This is a listing of the actual machine instructions generated by the compiler. Its format should be the same as that required in the DIRECT-JOVIAL brackets of a source deck. In addition, the address and octal notation of the instruction should also be shown. This output should also be available on an optional basis.

3.0 EXECUTION-TIME DEBUGGING AIDS

The majority of program debugging aids available to a user are used to best advantage when used at the time of the program's execution. This means that if the aids are to be made available, especially in an on-line mode, they should be under the control of a program or a series of programs operating as an extension of the compiler or preferably, as a separate package. Since all programs should be able to take advantage of the execution-time debugging tools, any program operating under the control of the debugging package should be a copy of the actual program. This insures that at the end of the debugging activity there is still in existence an original copy of the program and that the policy of reentrancy has been maintained.

It should be pointed out that it is the responsibility of the user to make sure his program is available for testing by proper classification of his program segment (e.g., it would not be possible to use the debugging tools on a segment classified as execute-only).

The aids which fall into this category require the use by the debugging package of either a program compool or the output from the set-used program, depending upon the type of debug request. Therefore, a request can be considered as falling into one of two groups.

3.1 Requests Requiring a Program Compool

The majority of requests in this area require the debugging package, at specified points during the execution of the program being tested to: (1) print data according to the data descriptions found in the program compool; (2) give a dump of the area of core memory occupied by a user's job; (3) dump core memory from specified locations; (4) change data values; (5) alter the flow of the user's job; (6) delete previously generated debug requests; and (7) terminate both the debugging and the user's activity. Other requests permit the switching of debugging output between on-line and off-line units.

In order to accomplish this, the user's job is initiated and control is then passed to the debugging package. The user's requests are interpreted, his program is modified at the specified locations in order for it to relinquish control to the debugging package, and control is passed to the user's program. Upon encountering one of these modifications during the execution of the job, control is relinquished to the debugging package long enough for it to carry out the request or requests. This continues until the job has completed its execution.

3.2 Requests Requiring Set-Used Output

The purpose of these commands is to enable the user to find out where in his source program he has made references to certain data or types of statements. The output is similar to that produced by the set-used program except the user can request information on a single type, i.e., a table, table or simple item, statement label, procedure, function, close, loop variable, switch, array, or string. He can request information on all references or just on either the set references or used references. What he receives is actually a partial set/used listing. In addition to the reference information, the requester may also ask for information concerning the ranges for loop variables and BEGIN-END brackets by specifying the card sequence number (or statement number) of the loop variable or either the BEGIN or END card or statement number.

The requests utilizing the output of the set-used program are not used during the actual execution of a job. They can best be used either before or after the execution in order to better take advantage of the aids described in paragraph 3.1.

4.0 MODES OF OPERATION

The modes of operation refer to the location of the input to and the output from the debugging package. If it is to a primary device at which a user can take immediate action, then the package is operating in an on-line mode. If it is to some other type of device, then the mode is off-line.

The modes of operation in regard to output are controlled by the user through requests to the debugging package. He may request that all output is to be in a particular mode or he may request a mode for an individual request by prefacing that request with the mode indication. The output in either mode is in exactly the same format.

5.0 EXAMPLES OF DEBUGGING REQUESTS

The following examples illustrate possible formats for the debugging requests. Similar formats have been incorporated in existing debugging packages and have proven to be successful.

5.1 Requests Requiring a Program Compool

All the forms immediately following assume the availability of a program compool. The upper case letters denote the actual requests. The lower case letters denote the variable sections.

```
 1)  AT        loc     PRINT data $

 2)  AT        loc     DUMP FROM loc TO loc $

 3)  AT        loc     DUMP PROG $

 4)  AT        loc     SET statement $

 5)  AT        loc     GOTO loc $

 6)  AT        loc     DELETE $

 7)  AT        loc     RETURN $

 8)  AT        loc     QUIT $

 9)  AT        loc     IF relational statement THEN action ELSE action $

10)  ONLINE $

11)  OFFLINE $

12)  GOTEST $
```

where:

AT	causes a breakpoint to be inserted in the user's job at the location specified. Upon reaching the breakpoint, the debugging package is entered and the request executed. It is not necessary to repeat the AT for successive requests if the location is to remain the same.
PRINT	causes the data to be output in the format specified in the program compool (i.e., as the user declared the data in his program).
DUMP	causes data, or instructions, to be output in an octal format.
FROM	specifies beginning location of dump.
TO	specifies ending location of dump.
PROG	causes a dump of that area occupied by the instructions and data of the program being tested.
SET	directs the debugging package to alter the user's job as prescribed by the statement portion.
GOTO	causes the normal flow of the job to be altered.
DELETE	causes a breakpoint previously inserted in the user's job to be removed.
RETURN	allows the user to interrupt the execution of his job to return to the debugging package for the purpose of accepting more requests.
QUIT	causes the debugging package to terminate itself and the user's job.
IF..THEN..ELSE	provides the user with conditional operations.
ONLINE	causes the output of the debugging package to be directed to the primary unit.
OFFLINE	causes the output of the debugging package to be directed to a segment or file to be printed at some later point in time.

Operation in either the ONLINE or OFFLINE mode remains in effect until changed by the opposite request. If the user wishes a mode to be in effect for only one request, he may preface that request with either ONLINE or OFFLINE.

GOTEST	causes the debugging request to start execution of the user's job.
loc	label name (for express labels)
	procedure name.label name (for local labels)
	segment name.word number
data	table name (refers to all occurrences of all items in the table)
	table name ($entry number$)
	table name ($first entry ... last entry$)
	table item name (refers to all occurrences of the item)
	table item name ($entry number$)
	table item name ($first entry ... last entry$)
	simple item name
	nent (table name)
	procedure name.data
statement	nent (table name) = integer constant
	table item name = constant list (constants must match item descriptions)
	table item name ($first entry$) = same as above
	simple item = constant (constant must match item description)
relational statement	item EQ item or constant
	item NQ item or constant
	item LS item or constant
	item LQ item or constant
	item GR item or constant
	item GQ item or constant

item	table item name ($entry number$)
	simple item name
	nent (table name)
	procedure name.item
	If a constant is used in a relational statement, it must correspond to the item type. The exception to this rule is the use of octal constants.
action	any debugging request which was legal follows the "AT loc" term (other than IF).

5.2 Requests Requiring Output from Set-Used Program

The following examples use the output of the set-used program:

1)	REFS	data	name $
2)	REFS	data	name SET $
3)	REFS	data	name USED $
4)	REFS	BEGIN	card number/statement number $
5)	REFS	END	card number/statement number $
6)	REFS	lpvar	card number/statement number $

where:

REFS	signals the debugging package that the outputs of the set-used program are to be used instead of the program compool.
data	can be either TABLE, ITEM, LABEL, PROC, LOOP, SWITCH, ARRAY, or STRING. This tells what type of data to search for in the set-used output.
name	is the name of the data called for by data. If it is data local to a procedure, then the form is procedurename.name.
SET	indicates that only SET references are to be output.
USED	indicates that only USED references are to be output. Absence of SET or USED indicates that all references are to be output.

For number 4, the card number/statement number is the card sequence number or the compiler-generated statement number (whichever was used during the execution of the set-used program) on which a BEGIN is found. The output from the debugging package is the card number/statement number of the corresponding END card.

For number 5, it is reversed.

For number 6, a card number/statement number for either the beginning or ending range of the loop variable specified by lpvar is input. The output is the card number/statement number of the opposite range.

Some sort of octal patch capability should also be provided. This could be taken care of by the following request:

AT segment name.word number PUT octal constant $

In the octal constant section of the request is the actual content of the word which is to be placed at the memory location specified by the segment name.word number. There may be more than one octal constant per request and they need be separated only by a space. The constants are inserted in consecutive locations and are terminated by the dollar sign.

6.0 RECOMMENDATIONS

It is recommended that only those requests using the program compool be considered as an initial capability. The requests utilizing the output of the set-used program, as well as the program itself, could be considered as desirable but not essential.

SECTION VIII

ON-LINE COMPILING

1.0 INTRODUCTION

This section discusses the methods of on-line compiling under MULTICS. The possible types of printable compiler output are also analyzed.

2.0 ON-LINE COMPILING

The term "on-line compiling" implies that a user requests the compilation of a JOVIAL source program from a console-type terminal (e.g., IBM-1050, Teletype Model 37). The user then expects compilation results directly via his terminal, or at least, results which he can examine at his leisure via supporting software (e.g., print the contents of a segment).

2.1 Method of On-Line Compiling

A user can prepare a JOVIAL source program for on-line compiling in two possible ways:

A. Input a card deck representing the JOVIAL source program into the MULTICS Basic File System via on-line, on-site techniques (e.g., card to tape to MULTICS, etc.).

B. Input a JOVIAL source program into the MULTICS Basic File System by typing at a console terminal via the MULTICS Editor program.

These are only two methods of inputting a program. These methods do not preclude others, as long as the JOVIAL source program resides in the MULTICS system in some segment which is available to the JOVIAL compiler.

After the user has created the file containing the JOVIAL source program, a command line of the following form is typed:

jovial alpha

The term "jovial" requests the MULTICS Operating System to invoke the JOVIAL compiler with the argument alpha. The term "alpha" represents a pointer to the directory in the MULTICS Basic File System which, in turn, points to the location of the user's source program, which is assumed to have been previously created by the user within a segment (e.g., beta.jovial).

2.1.1 Options

The various possible options to the compiler (e.g., SET-USED ON, COMPOOL_ASSEMBLE ON, BRIEF OFF) can be input in a number of different ways.

One method is by a direct use of the MULTICS option command. This command permits a user to establish a permanent record of the types of options that he desires. The format for the option command is:

option optionname setting

For each compilation, the user need not re-type all the options. The "jovial" command will result in the calling of the read-opt procedure which determines the status of the various options whether they be peculiar to the JOVIAL compiler (e.g., COMPOOL_ASSEMBLE) or to all translators in the MULTICS system (e.g., BRIEF ON).

If a user wishes to override his current option settings for a particular compilation, he can include an interjected option command within his JOVIAL request. The following command line is an example of an interjected option command:

jovial alpha (option brief off)

In the above example the option command is invoked to set the brief option to off before the compilation of the contents of the segment pointed to by alpha. In this case, the system does all the work.

Another method of inputting options to the compiler would be a command line of the following form:

jovial alpha (list brief no_setuse)

This example shows the JOVIAL command to have two input arguments. One, the term "alpha" and, two, a list (array) of character strings that the compiler must scan and interpret itself. In this example, the illustrated options to JOVIAL are not known to the MULTICS system unless the JOVIAL compiler calls the system routine which records option settings.

In summary, the option command has been provided so that the user need not re-type all his options for each compilation regardless of the language involved. These options are available to any MULTICS subroutine and, for efficiency sake and MULTICS standardization reasons, all language translators are expected but not required to interface with the option facility. As long as a user has created an option setting, any language translator may access it via system subroutines.

2.1.2 Output

The JOVIAL compiler processes the source program pointed to by alpha. All output from the compiler is directed into new files in the same location pointed to by alpha. Examples of these compiler created files are:

beta - contains the binary equivalent of the JOVIAL source program that was contained in beta.jovial.

beta.error - contains diagnostic messages from the compiler.

beta.link - contains the linkage segment for the binary program in beta.

2.2 Advantages of On-Line Compiling

The user can compile his JOVIAL source program and receive immediate output. Job through-put will be as expedient as possible.

3.0 USER OUTPUT

As the source program is being compiled, syntactical errors may occur. These errors should be reported by the compiler immediately as they occur, on-line, so that the user may either correct his errors and reinitiate the compilation or completely abort the compilation from his terminal. The error messages may possibly also be written into a segment (e.g., beta.error) which the user may print or retain as a continuous history of the compilations of beta.

The compiler should also produce a segment in which will be included error messages interleaved with the source program, set-used listing, source program dictionary, etc. This segment could, at a later point in time, be printed via off-line software or directly on the user's terminal. Whichever method of printing the user selects, the compilation will not have been restrained by the relatively slow speed of the user terminal.

4.0 BATCH-PROCESSING COMPILING

A batch-processing compiler serially processes a number of JOVIAL source programs and produces binary output for them. This type of compiler normally operates under the complete control of an operating system such as GECOS. Another alternative would be a free-standing compiler which handles all its hardware interfaces by itself.

It is felt that this type of compiling would be neither useful nor feasible in the MULTICS environment and should not be considered at this time.

SECTION IX

PARTIAL COMPILATION CAPABILITY

1.0 INTRODUCTION

Partial compilation provides the capability to compile a small portion of a program without the need to compile the entire program. More specifically, partial compilation provides a means to modify a program symbolically by compiling additions of new symbolic code or deletions of existing symbolic code without recompiling the entire program. Since only a small portion of a program is recompiled for each modification, partial compilation is quicker and less expensive than recompilation of an entire program. Since the modifications are made symbolically, partial compilation is simpler and more reliable than machine language changes such as octal patches.

The extent of the partial compilation capability implemented is reflected in the complexity of the resultant compiler. A compiler that allows the compilation of individual program statements or small groups of statements would differ considerably in complexity from a compiler that merely allows the compilation of distinct program parts such as individual procedures. Many new techniques would have to be developed for the implementation of an extensive partial compilation capability.

One experimental JOVIAL compiler with a partial compilation capability is currently being studied at SDC as part of the Interactive Programming Support System (IPSS), which was supported in part by RADC. A tentative conclusion of the IPSS project is that it is not feasible to compile individual program statements. They are now investigating the feasibility of compiling small groups of statements. We feel that duplication of the IPSS effort is unwarranted and that the final decision about implementation of an extensive partial compilation capability be reserved until the IPSS study is completed.

2.0 RECOMMENDATION

We recommend implementation of a somewhat limited partial compilation capability which allows the compilation of individual procedures separate from the compilation of the main program or any other procedure. We use the term "external procedure" to refer to a procedure that is compiled all by itself and hence is external to the program or procedure calling it. Whenever an external procedure is compiled, the control information to the compiler would identify the procedure as an external one, in response to which the compiler would generate the instructions and storage required for subsequent linkage with the calling program. Whenever a program or external procedure that calls an external procedure is compiled, some information about the called external procedure must be supplied to the compiler. The reason for this is twofold: it signals the compiler to generate the instructions and storage required for linkage with the external procedure and it supplies the compiler with information about the formal input/output parameters of the external procedure. One way to provide the compiler with the necessary information about an external procedure is to define the external procedure in the Compool. Another

way, which is not absolutely necessary but is desirable, is to allow for a prototype procedure declaration of an external procedure within the program and hence within the JOVIAL language. The prototype procedure declaration would consist of the procedure heading followed by a procedure body that contains no statement list. For example, the prototype procedure declaration of external procedure XYZ might be:

```
PROC   XYZ(AA,BB = CC) $
       ITEM AA F $
       ITEM BB F $
       ITEM CC F $
BEGIN
END
```

3.0 ADVANTAGES OF LIMITED PARTIAL COMPILATION

A partial compilation capability that allows for the compilation of individual procedures can provide most of the benefits of a more extensive partial compilation capability. Moreover, the implementation cost for such a capability would be quite low since the linkage mechanism of the MULTICS system can be used to combine the external procedures and the rest of the program into a single unit. While such linkage does introduce some inefficiency, in many cases the inefficiency will be negligible. In other cases, it might be desirable to reduce the linkage time by using the planned binding mechanism of the MULTICS system. The amount of inefficiency introduced would be considerably less than that introduced by the more extensive partial compilation. Furthermore, since partial compilation is of primary value during program checkout, the program can be compiled as a single unit after the completion of its checkout, thus eliminating any unnecessary linkage.

4.0 DISADVANTAGES OF LIMITED PARTIAL COMPILATION

The limited partial compilation is neither as flexible nor as convenient to use as the more extensive partial compilation capability. It necessitates that a program be physically divided into parts. In addition to any procedure the programmer chooses to compile as an external procedure, each procedure called by an external procedure must also be an external procedure and compiled separately. Several restrictions are imposed on the external procedure: it will no longer be allowed to reference data declared in the main program, requiring such data to be defined in the Compool; it will no longer be allowed to branch to a main program statement but will be required to execute the normal return processing. The amount of data defined in the Compool will be increased.

SECTION X

STRING PROCESSING

1.0 INTRODUCTION

This section outlines the advantages and disadvantages of implementing JOVIAL STRING items in a GE-645 JOVIAL compiler. In addition to a possible implementation approach, a tentative conclusion is reached regarding the feasibility of the inclusion of STRINGS in the compiler. For formal definitions of STRING and bead, the reader is referred to AFM-100-24.

2.0 ADVANTAGES

A. <u>More than one occurrence of the same item may be packed into one computer word.</u>

For example, if a program requires 900 entries for a 12-bit item on a machine whose word size is 36 bits, a STRING item may be declared such that only 300, in lieu of 900, computer words are necessary.

Instead of declaring a table with 900 entries and one item 12-bits long, a table with 300 entries and one STRING item could be declared as follows:

```
TABLE TB V 300 S  1  $
BEGIN
  STRING SI I 12 S Ø Ø D 1 3 $
END
```

The table TB would have the following format:

Ø — 11	12 — 23	24 — 35
SI($Ø,Ø$)	SI($1,Ø$)	SI($2,Ø$)
SI($3,Ø$)		
SI($897,Ø$)	SI($898,Ø$)	SI($899,Ø$)

with a resultant saving of close to 600 words of core storage.

B. <u>The words of a long literal item may be individually accessed.</u>

This can be accomplished by declaring a STRING item to be literal, one word in length, and overlaying a long literal item.

For Example:

```
TABLE TB V 1Ø S 2Ø $
BEGIN
  ITEM LONGL H 12Ø Ø Ø $
  STRING OVERL H 6 Ø Ø 1 1 $
END
```

With the above declaration, any word of item LONGL may be accessed with the expression OVERL(w,e) where w is the word number desired (Ø ≤ w ≤ 19) and e is the entry number desired (Ø ≤ e ≤ 9).

Without STRING items being implemented, the above outlined objective could be achieved in one of two other ways. Fifteen different table items could be added to this table, each of which overlays a different word of LONGL. A much more preferable technique would be to use the BYTE modifier. For example: "BYTE($6,6$)(LONGL($Ø$))" refers to the second word of the first entry of item LONGL. With STRING items, the programmer could write an expression one-half the number of characters needed in the BYTE expression (i.e., OVERL($1,Ø$)) to reference the same data area. In addition, in the case that the word number (thus, first bead of the BYTE expression) is variable, a compiler would be able to generate significantly better code for the STRING item than it could for the BYTE modifier. This is because it does not know if the expression "BYTE($X,6$)(LONGL($Ø$))" refers to a data area within one word or extending over two consecutive words. It would therefore, be forced into always generating code to handle the worst case. The bead of a STRING item, on the other hand, is known to the compiler to be within one word, and the compiler can always generate the best possible code when it is used in this manner.

C. <u>The individual bytes of a literal item may be individually accessed.</u>

For example, by adding the declaration:

```
STRING OVERB H  1  Ø  Ø  1  6 $
```

to the table used in Section B above, it is possible to access any one byte of item LONGL with the expression OVERB(b,e) where b is the byte number desired (Ø ≤ b ≤ 119) and e is the entry number desired (Ø ≤ e ≤ 9).

The STRING item, in this case, has the same two advantages over the BYTE modifier as it had in Section B; i.e., the programmer can write half as much JOVIAL to accomplish his objective and the compiler can produce significantly better object code in the event of a variable byte number.

D. <u>The ability to preset a STRING item gives the programmer a limited capability to set certain individual bytes of a character string and only those certain bytes.</u>

By presetting the first word of STRING OVERL in Section B above, the first word of ITEM LONGL would be preset without touching any of the

other words of LONGL. Likewise, succeeding words of LONGL could be preset without changing those words of LONGL which it is desired to leave unset.

E. The presence of a control item allows the programmer to work with a variable number of beads in each entry of a table.

Thus, a table containing data about an organization's personnel might contain a STRING item which has some indication of each of the different professional associations to which a person belonged. The number of beads used in each occurrence of the STRING item could be variable depending upon the number of associations to which that person belonged. The table would, therefore, use only as much space for each entry as it necessarily had to in order to contain its data. Extra space need not be allocated for every entry in order to allow for the maximum case in every case.

F. The inclusion of the interval parameter in the STRING item's declaration allows one to skip a specified number of words before subsequent beads are accessed.

For example, STRING SI of Section A above, with an interval of two would look as follows:

SI($Ø,Ø$)	SI($1,Ø$)	SI($2,Ø$)
SI($3,Ø$)	SI($4,Ø$)	SI($5,Ø$)
SI($6,Ø$)	etc.	

The words between the occurrences of SI could be used for other data.

G. In a character-oriented machine, the usage of STRING could allow a programmer to conveniently address any one single character in core memory.

Likewise, in a machine with powerful character manipulation instructions, the usage of STRING would allow one to reference different bead lengths (up to one word) throughout core knowing that the compiler has these instructions at its disposal.

3.0 DISADVANTAGES

The sole disadvantage of STRING items, albeit a consequential one, is the cost of implementation. The primary problem area would be in the translator (code generator) phase of the compiler. Since it is legal to use a STRING item anywhere any other variable is legal, the translator would have to be prepared for STRING items used as subscripts, as beads, as input and output parameters, as switch items, etc. There are a significant number of different areas of the translator which would have to be aware of the possibility of a STRING item being used. Upon discovering the use of a STRING item, a different and, in

some cases, unique set of code would have to be generated. This is no trivial task. As an illustration, the following is, in general, the kind of code the 645 compiler must generate for the retrieval of a STRING item declared:

```
STRING SI description c  d  e  f  $
```

and used:

```
SI($a,b$):

LDQ     a
DIV     f
STA     Temp1
MPY     e
STQ     Temp2          }
LDQ     b              }   Only needed if # wds/entry ≠ 1
MPY     # wds/entry    }   Otherwise,  ADQ     b
ASQ     Temp2          }                STQ     Temp2
LDQ     Temp1
MPY     (bead separation)
ADQ     c + (bead size)
STQ     Temp1
LDA     Temp2
LDQ     SI, AL
QLR     Temp1,*
ANQ     Mask = (bits=1 for bead size)
               (i.e., clear to zero all but meaningful bits)
```

Note that the compiler has to create and keep track of two temporary registers; likewise, it has to create registers containing the constants "f" and "e" and make them a part of the user's program. Dependent upon the packing specification, "bead separation" must be calculated by the compiler and saved in a register which becomes part of the object program. The "mask" used in the final instruction is yet another variable which must be determined by the compiler. It should be noted that the code shown here is meant to handle the general case, i.e., one in which the STRING item's subscripts are both variables. In the event that either or both are constants, a good portion of this code can be eliminated as the compiler could and should do the arithmetic calculations itself and not generate code to do so (e.g., eight divided by two is always four; there is never any need to generate code to calculate this within the actual object program). With constant subscripts, the retrieval should be possible in three instructions. This, of course, means that much more work for the compiler, with the justification being a more efficient object program. One further note; if the bead is literal, extra code must be generated to prefix it with literal blanks rather than arithmetic zeros.

4.0 CONCLUSIONS

STRING items can be used to good advantage by JOVIAL programmers. They answer a number of different programming needs. However, STRINGs must still be considered a "frill" within the JOVIAL language. While they offer shorthand and convenient methods of manipulating certain kinds of data, there is nothing that the use of STRINGs accomplishes that can't be accomplished using other JOVIAL forms such as long literals and the BIT or BYTE modifiers. It is therefore recommended that STRINGs be implemented on the 645 JOVIAL compiler, but only if it is done so as not to exclude any other language form. In other words, STRINGs should be included only after everything else has been done and sufficient time and talent remains for the implementation. Since this is essentially a positive recommendation (even though it may not read as such, it is intended to be), it is consequently imperative that no portion of the compiler be initially coded so as to preclude, either by design or accident, the eventual inclusion of STRING items.

SECTION XI

BINARY VERSUS SYMBOLIC OBJECT CODE

1.0 INTRODUCTION

This section addresses itself to the problem of whether a GE-645 JOVIAL compiler should produce binary or symbolic object code. Closely related to this problem is the unique paging environment of MULTICS.

2.0 BINARY CODE

Binary code is that code produced by the compiler which is immediately executable. This is to say that no subsequent processing of the compiler's output is required to produce a usable computer program.

2.1 Advantages of Producing Binary Code

An advantage of a compiler that directly produces binary object code is the fact that the compiler can generate more efficient code for the paged environment of MULTICS. In its processing, the compiler constructs large internal tables of descriptive information. It is through an analysis of these tables that the object code is produced. In the MULTICS environment, the compiler should be cognizant of not just efficient code, but efficient object code that will be paged as well. It is the presence and usage of those internal descriptive tables that would help in the achievement of this goal.

Production of binary code causes a minimal amount of computer time to be expended for language translation, thereby reducing the expense of producing operational computer programs.

Not to be overlooked is the fact that official JOVIAL specifications direct the compiler to produce binary object code and not interface with a separate assembler.

2.2 Disadvantage of Producing Binary Code

The sole disadvantage of producing binary code is that the cost of developing a compiler with this type of output would be higher than the cost of a similar compiler which produces symbolic code.

3.0 SYMBOLIC CODE

The problem of binary versus symbolic object code is more than a problem of feasibility in view of compiler development cost. Other considerations are, in the case of symbolic code, the availability of a reliable assembler and the completeness of the instruction set which that assembler supports.

3.1 Advantage of Producing Symbolic Code

In terms of cost, a JOVIAL compiler that produces symbolic machine code would be less expensive to develop than a compiler that produces binary object code.

3.2 Disadvantages of Producing Symbolic Code

The symbolic object code would have to be input to an assembler to produce executable object code. Program production time would be increased due to the separate compilation and assembly processes. This presents probably the major obstacle to the production of symbolic object code: the availability of a reliable assembler.

At the time of the release of the MULTICS Operating System, there will be only two assemblers which will produce binary output suitable for running on the GE-645. These are the EPLBSA and FL/1 assemblers. There is no plan for EPLBSA to be released with the system. The FL/1 assembler will be relatively new, and therefore its reliability would be insufficient to support a compiler producing symbolic code. Although the EPLBSA assembler has been used to produce the operating system, it has a limited macro capability and it is questionable if its operating speed is sufficient to prove satisfactory in the time-shared environment of MULTICS.

4.0 CONCLUSIONS AND RECOMMENDATIONS

Although the cost of a compiler producing binary object code is higher than a compiler which produces symbolic object code, the time required to produce the operational object program via the binary object code route is less.

The paged environment of MULTICS should not be ignored by language translators if they are to generate the best possible object code. To do this, the compiler should fully utilize all possible information from its internal tables which can only be done by directly producing binary object code.

The availability of a complete and reliable GE-645 assembler is questionable at this time.

Official JOVIAL specifications call for the output of binary object code.

In view of these conclusions, it is recommended that the compiler produce binary code. However, this recommendation is qualified to some extent. To achieve a usable JOVIAL capability under MULTICS at the earliest possible time, and if the EPLBSA assembler is available, it may be desirable to at least consider the production of symbolic object code and a temporary interface with the assembler. Using this initial JOVIAL-to-symbolic object code capability as a base, it may be possible to develop a more advanced and efficient compiler by concentrating on the production of binary object code with matured MULTICS subroutines for standard system interface requirements.

APPENDIX

The following language forms are defined in the J3 language specification, but are omitted from or altered in the JB (Basic) language specification:

ALL modifier

ARRAY declarator

boolean:item

CHAR modifier

dual:item, constant, etc.

exchange:statement

fixed:point:item, constant, etc.

IFEITH-ORIF

like:table

MANT modifier

MODE declarator

ODD modifier

STRING declarator

relation:lists

transmission code:item, constant, etc.

close, statement:name, table, array as procedure parameters

rounding as part of item:description,

range values

implied definition of simple:items via presetting

protected switches and closes

switch or close as switch branch point

unnamed table

the two operands of literal assignment and comparison statements must contain an equal number of characters in JB

the two operands involved in entry manipulations must be of equal entry size in JB

names are limited in length to six characters in JB (J3 defines no limit)

names may not contain embedded primes in JB

FILE declarator

INPUT/OUTPUT operations

DEFINE may indicate a string of symbols (JB limits second operand to a constant)

more than one OVERLAY may contain the same variable

OVERLAYS may include environment with preset data

ENT or ENTRY is acceptable (in JB, only ENT is accepted)

PART II - PRODUCTION OF OBJECT PROGRAMS FOR A PAGE-ORIENTED COMPUTER SYSTEM

SECTION I

INTRODUCTION

The purpose of this report is to present the results of an investigation into the concept of Paging for the purpose of establishing techniques for the generation of code that operates effectively on a computer with paged hardware features. The objective of this investigation was twofold:

1. To determine if the code generation process for paging can be automatic (handled by software) or if present programming techniques should be altered to produce efficient code generation.

2. To define an implementation approach which will allow rapid implementation of a Paged JOVIAL compiler and the transfer of existing JOVIAL programs to a paged environment.

Paging is a relatively new technique which allows division of programs into equal blocks of information and which permits an easier allocation of physical memory. Through this technique, it is possible to select any or all divisions of a stored program to obtain the information desired. A paged memory allows flexible techniques for dynamic storage management without the overhead of moving programs back and forth in the primary memory. This reduced overhead is important in responsive time-shared systems where there is heavy traffic between primary and secondary memories.

The mechanism of paging, when properly implemented, allows the operation of incompletely loaded programs. A supervisor need only retain in main memory the more active pages, thus making effective use of high-speed storage. Whenever a reference to a missing page is made, the supervisor need only interrupt the program, retrieve the missing page, and reinitiate the program without loss of information.

The balance of this report is divided into two main topics. The first is a description of features that can be inserted into an object program by a compiler to enable the program to operate more efficiently in a hardware-paged environment. The second is a description of a compiling process that produces object code optimized for computers operating within a page-oriented system.

SECTION II

PAGING OPTIMIZATION FEATURES FOR AN OBJECT PROGRAM

1.0 INTRODUCTION

With the development of computers with paged hardware features, it has become evident that a reevaluation of program structure is in order. A compiler which properly organizes the code of an object program can substantially reduce the cost of running this program under an interactive time-sharing system that utilizes hardware paging features. In order to accomplish this, new compiler techniques must be developed and used in compiling program statements into this more favorable program structure. Consequently, two major areas should be investigated: the optimum organization of code; and the techniques to be used in compiling into this form.

2.0 ORGANIZATION OF OBJECT CODE

In a typical JOVIAL program, the program statements control the order of execution of the compiled code. Therefore, the flow of the resultant code is fixed. However, this code may be ordered within core in various ways, requiring the addition and deletion of control jumps and thereby altering the normal flow.

Variables should be examined before being assigned to a storage location. Those that are never "set" but "preset" should be handled in a manner similar to that of handling constants. It is profitable to locate some constants in more than one location, and will be profitable to dynamically relocate some variables during execution.

In the case of tables where the format is not specified by the user, the compiler should determine whether the table is to be structured in serial or parallel, according to its usage by the object program. In fact, a combination of serial and parallel constructions may be desirable for some applications. That is, a serial table may be divided into several parallel tables.

2.1 Compiler Requirements

In order that a compiler may properly structure a program, it is necessary to collect enough information concerning the program to enable the compiler to make certain decisions. A reasonable approach is to include this information collecting in the generator phase of a generator-translator compiler and to maintain this information in the compiler's dictionary.

This information should aid in the assignment of storage for data and, in addition, will guide the compiler in selecting sequences of intermediate language steps for translation to computer code. Before the translation process, an analysis will be made to select sequences of the intermediate language to be translated into computer code.

2.2 Compiler Restrictions

A few considerations and adjustments are required before the analysis can proceed. Most important of these considerations are the aids provided by the programmer in specifying the construction of data. Specifically, OVERLAY data declarations restrict all the references to any of the constituent items to references to an ordered group. If all the items in the overlay are invariant, then the group is considered as invariant. If any item in the OVERLAY group is set or changed during execution, the entire group must be considered as a single set-variable group. Completely defined tables or the use of an OVERLAY declaration for table items will similarly group the items.

The use of the ENTRY variable also imposes restrictions. If it is set by any assignment statement, all items within the table must be considered as set-variables. If the ENTRY variable is used by a statement and never set, they are not all necessarily set-variables. Thus, in some cases, references to certain variables will be treated as a reference to a group in the analysis that follows.

3.0 STRUCTURING TECHNIQUES

Before evolving compiler techniques for code optimization, it is necessary to define the characteristics required of efficient object code. The following descriptions set forth some of the characteristics that lead to efficient "page" utilization by a program. Compiler techniques can be evolved to incorporate all of these ideas into object programs. In some cases, the characteristic is explained in terms of the compiler techniques used.

3.1 Output Classification

For this analysis, the compiler output is classified into three sections. The first section includes all the data variables that are set or altered during the execution of the object program. The second section contains the data and constants that remain invariant throughout the execution of the resultant code. The third section consists of the computer instructions. These instructions are said to be "pure," that is, they will not be altered during their execution. Any code changes that are required should be stored with the variables.

3.2 High-Activity Area

A high-activity area for set-variables is required in addition to the more voluminous area reserved for all set-variables. The high-activity area can be used as a temporary storage area for data during their periods of reference activity and can also serve as the only storage area for short-lived data. In general, the data within this area are from one of three groups: (1) the unnamed temporary intermediate values for which no general storage space is provided; (2) the named set-variables that have a short local scope; and (3) data for temporary holding. The data of the third group can be either preaccessed variables or set-variables that are to be moved into their permanent assigned area at a more auspicious time.

3.3 Table Divisions by Data Classification

Each table that is declared can be subdivided into the items that are invariant and the items that are set or altered during the execution of the compiled code. Two parallel tables are produced. One contains set-variables and the other, invariant data. Optimum storage assignment is then possible, consistent with each type of data. Of course, some invariant data items are tied to set-variable items. This results from completely defined tables or OVERLAY declarations containing table items.

3.4 Multiple Store of Invariant Data

In some cases, it may be efficient to assign the same data to more than one location. This would be especially effective if the references to the data are isolated.

3.5 Assign Invariant Data With Instructions

Invariant data assignments that are utilized in a segment of the object program can be physically located on the pages together with the instructions that access that data.

3.6 Block Invariant Data

Invariant data that are used throughout the object program can be blocked in parallel to the segments in which they are used. These blocks are useful in assigning storage so that the invariant data "page" circulation is minimized. Two methods may be used to assign data to the same storage area: (1) the data from blocks that are essentially identical, and (2) the data from blocks that are used sequentially.

3.7 Separable Set-Variables

If the scope of a set-variable can be divided into nonoverlapping sections, the variable can be treated as if each scope applied to a distinct variable. Hence, each section can be assigned to a distinct storage cell. In fact, if one section is of short local scope, it may only appear in the high-activity area.

3.8 Storage Overlay for Set-Variables

The data storage assignment for set-variables is considered to be binding only from the time they are set to the time they are last used. Hence, it is feasible to assign a storage location to more than one variable if their scopes do not overlay, and the page is active during the scope of the variables.

3.9 Named Temporary Set-Variables

Set-variables that are local to only a short segment of program flow need not be assigned a permanent storage area. The result may be held in a

general register for its subsequent use. More frequently, it could be held within the high-activity area during the time it is needed.

3.10 Correlated Set-Variable Assignments

The sequences of access to the various set-variables may be compared for similarities. Any similarities should be reflected in the storage assignments of set-variables. Permutations of these patterns resulting from a preaccess move to the high-activity area as well as a delayed storage should be included in the comparing.

3.11 Parallel Subtables

Table items that occur in distinct program segments can be assigned to separate parallel subtables. The items that are used in several segments may be divided according to the reference pattern. This division should be performed for the invariant item tables as well as for the set-variable tables.

3.12 Serial Tables

If, in the analysis of segments that form loops, it is determined that the subscripts are indexed systematically, it may be desirable to order the indexed table items serially. In the case of a search where one item of an entry is used as a key until a match is obtained, it would be more efficient to provide a table for these items as a separate table parallel to the other items of the entry.

3.13 Instruction Assignment Order

The statements entered into the compiler specify the algorithm of solution to be used by the object program. As such, this specifies the order of execution of the pertinent statements. This does not place a restriction on the order of location for the object program. Unconditional jumps can be added and deleted as needed. In addition, the conditional branches can have their relational operations inverted to rearrange the object code.

3.14 Instruction Execution Order

A pair of adjoining assignment statements can be interchanged if either does not access its companion's set-variable and if any function involved does not affect any of the used variables. An interchange of statements may be desirable to improve the reference pattern with respect to data.

3.15 Statement Elimination

Statements that are not executed should not be translated. In addition, the assignment statements, together with set-variables that are not used or that are reset before use, should be eliminated. In either case, error messages would be supplied by the compiler. (This technique is primarily aimed at code improvement although it may also aid paging.)

3.16 Loop Data Assignment

The reduction of page turning is especially important in loops. Three optimizing techniques are available to improve the data referencing of a loop. First, statements that are independent of the loop can be removed from the loop. Those that set data to be used within the loop should precede the loop. Those that set variables not to be used within the loop can follow the loop exit. Care must be exercised if alternate exits exist. Loop-invariant data can be precalculated before the loop is activated, and can be moved to the high-activity area before entering the loop. The primary concern is to reduce the loop references to storage areas outside the high-activity area. A bonus effect should be the reduced computation within the loop.

3.17 Loop Instruction Alignment

The instructions that execute the loop can be assigned to a minimum number of pages. This may involve skipping some storage registers. This skipped storage can be used for invariant data.

3.18 Jump Simplification

A jump to a jump may be simplified by inserting the second jump for the first. In principle, excess jumping is reduced as much as possible. If the jumps involved are conditional, inserting the code in more than one place to evaluate the conditions should be considered.

3.19 Page Utilization

A program's organization is calculated to minimize page jumping. If frequency data relative to the branches of a conditional jump is available, the assignment of instructions to pages should be adjusted according to this information. In some cases, a set of instructions may be repeated if this eliminates page turning without an appreciable increase in the program size.

3.20 Advise Executive

A final requirement is to advise the executive routine in advance when additional pages will be required. To prevent overriding storage that is still required, it is also necessary to advise the executive when a "page" is not needed for further use. On a nonpaging computer, these executive calls can be used for secondary storage assignment. The specified calls can be altered to call special functions that load the requested data.

4.0 RELATED GAINS

In addition to the advantages gained for a "paging" computer, there are related achievable gains that are not considered to be strictly "paging" optimization. One gain is that only the set-variables need be saved before they are over-ridden. Code can be reentrant, and computation for unused set-variables can be eliminated. The resulting object code is easily made self-initializing if the first access of the set-variables is to the preset area and if their subsequent saving and restoring is never done on top of this preset area.

Another feature is that no data assignment is absolute. Each address is set as a base address plus a fixed displacement. The base address is a variable and need not be assigned until the program is loaded. If care is exercised in generating the object code, it should prove easy to dynamically reassign this base address value whenever the page is reloaded.

5.0 CONCLUSIONS

This section has proposed several characteristics for "page" oriented object program code. The compiler, in which the techniques to produce such an object program are implemented, should collect information in the generator phase. The optimizing routine will juggle the data and program statement sequences and assign core locations. It will also direct the translator in its translation of intermediate language into computer code. A sequence may be translated more than once. The amount of improvement in program structure can only be a matter of conjecture until the compiler using these techniques is designed and implemented.

SECTION III

COMPILER PROCESSES TO PRODUCE PAGE-ORIENTED PROGRAMS

1.0 INTRODUCTION

Following the definition of paging optimization features for an object program is the design and development of compiler techniques which can best implement these features, and the determination of a compiler algorithm that optimizes object programs for efficient operation. Standard code improvements increase processing efficiency within page-oriented systems. Some conventional code improvements provide an extra bonus in efficiency for a paging environment.

A program optimized for paging may not be seriously degraded if executed on a non-page hardware computer with a page-oriented software system. Conversely, a program optimized without considering paging is superior to a non-optimized program if executed on a page-oriented computer system. As stated previously, one of the main objectives of this study is to determine the extent of additional gain a compiler can achieve through structuring and rearranging of problem solutions for a paging system. Programs produced by such a compiler will be more efficient on a page-oriented computer system than if optimization was not attempted.

The optimizing procedure is proposed as one algorithm, to take advantage of data collection and data processing similarities, and to find the extent of optimization feasible. Each optimization feature or code improvement does not have to stand by itself. The important criterion is placed on the totality of code improvement. Individually the code improvements are to be measured against this total improvement. Although code optimization is feasible and can be profitable, it is doubtful if any particular code improvement feature is profitable by itself. Therefore, to establish the proper basis for an evaluation of the individual page optimizing features, a comprehensive code-improving compiling method is required.

The optimizing features and techniques that comprise the proposed algorithm are presented in the following paragraphs. Contained in the paragraphs are the general assumptions for this compiler optimization, an overview of the optimizing algorithms, and a detailed description of the individual procedures within the analysis.

2.0 GENERAL ASSUMPTIONS

Before proceeding with the description of the optimizing process within the compiler, it is necessary to describe some of the assumptions fundamental to the processing procedure and its final result--an object program. The following subparagraphs describe the form, nature, and content of the input, the liberties permitted in permuting the order of execution in evaluations, and data considerations.

2.1 Input Requirements

The input for this compiler process consists of an intermediate language program with an associated dictionary. It is assumed that some previous compiler action, such as the generator phase, has transformed a program from the source language into an appropriate intermediate language and has produced a corresponding dictionary of the program symbology. The first processing of the input data sets reduces the dependence of subsequent processing on a specific source language. Only the first processes depend upon this input. The initial optimizing task is to extract information from the input data sets and to transform the input into a new intermediate language program and a new dictionary.

Each instruction in the intermediate language consists of a function or operation code plus pointers to the cells that represent operands. The cells that represent operands are either dictionary entries or other intermediate language instructions. Thus, source language statements can be interpreted as small trees in the intermediate language. These small trees represent segments of programming such as would appear in processing boxes of a flow chart. The source-language-dependent process mentioned above creates this structure such that the intermediate language program is essentially source-language-independent.

2.2 Execution Order of Instruction

It is assumed that the intermediate language instructions occur in the order in which they are to be executed in solving the source language problem. This implied order of execution is not absolute. Liberties are possible with this order of execution without altering the problem solution. An intermediate language instruction is eligible for execution as soon as the values of its operands are available. This condition can occur during compilation if the operands are all invariant. If an operand is the result of a prior instruction, then the prior instruction must be executed before the current instruction. That is, if an operand has a variable value, then the proper value must be established before an instruction using the operand can be executed.

In changing the order of execution, the flow indicated by conditional branching and the consequent rejoining must be considered. An instruction that is advanced past a branch point is included on all flow paths in which it is redundant. An instruction cannot be advanced past a collection point if it alters the result for any path. This move collects common instructions after the flow paths merge. In moving an instruction backwards, the reverse requirements are imposed. For a collection point, the instruction must be inserted on all flow paths. For a branch point, if moving the instruction alters one of the other paths, it cannot be moved. This move normally collects common instruction that can be executed before branching.

2.3 Freedom in Algebraic Evaluations

An algebraic expression imposes an interdependence of execution sequence on operations. Normal evaluation is from left to right for operations of equal priority. For instance, the evaluation of "A+B+C" is as if it were written "(A+B)+C."

The evaluation of "A+BxC-D/(E+F)" is offered as an example of the normal freedom allowed. In the evaluation, the product of B and C is formed before the product is added to A. The sum of E and F is obtained before dividing D by that sum. The sum of A and BxC, as well as the quotient, are formed before the subtraction can be executed. Using parenthesis, the expression is evaluated as

"((A+(BxC))-(D/(E+F)))"

The accession of operands can be at any time between the point at which they are established and the point at which they are required in the evaluation. In fact, if B and C in the preceding example are both preset constants, the compiler could perform the multiplication instead of the object program.

If the value of a function is required, some variable values may be changed during the evaluation of the function. The necessary intelligence to identify changed variables is available if the function evaluation is compiled with the program. The function evaluation is able to change only the data defined in common for the precompiled function evaluations. Procedures indicate the established values within their actual parameters. In any case, the data that are subject to change by a function evaluation are accessed before the function is evaluated if the operand occurs before the function within the problem statement. Otherwise, the data are accessed after the function is evaluated.

2.4 Data Considerations

The principle data handling facility that the compiler provides is the assurance that the proper values of the operands are used in the execution of the program. An absolute storage location is not required for the value of each variable.

Values of simple variables that are invariant throughout the problem solution can be treated as if they were constants. The values of constants are located in any storage location convenient for their use as an operand. If the same constant value is required as an operand in more than one instruction, the value may be available in several storage locations for instructions isolated from each other.

Values of simple variables that change during the program flow are considered temporary. They exist only as long as they are needed as operands. Some storage locations contain different variable values in different portions of the program solution. Some values are dynamically relocated to avoid page turning. Thus, some values occupy more than one storage location. Some values exist only temporarily in an arithmetic register and never occur in a storage location.

Values of subscripted variables are necessarily more closely tied to storage locations. The primary reason is the problem of accessing the proper value. Specifically, this information exists as a block of information. As such, the flexibility of its handling is more restricted than simple variables. A block is treated as a simple variable that has extra storage requirements that impose added restrictions.

3.0 OPTIMIZING ALGORITHM

An efficient manipulation algorithm is required for the profitable compiler production of optimized object program code for a page-oriented operating system. The analysis must be complete in order to maximize the gains. The various procedures must be integrated to avoid unnecessary compiler processing. The proposed method integrates the techniques into one composite algorithm. The program intelligence lists are designed for maximum information with minimum processing.

This paragraph describes how the techniques are integrated. The description covers the data lists and indicates the various processing steps with their appropriate input and output data.

3.1 Lists

The processing algorithm is concerned with data manipulation. These data are contained in various lists, depending upon content and intended use. In order to comprehend the algorithm, it is necessary to know the content and uses of these lists. Therefore, a description of these lists is supplied before the algorithm is presented.

Within these lists a pointer to another entry, whether in the same list or in some other list, is an index of the entry within its list.

3.1.1 Intermediate Language Program

This list is one of the initial inputs to the algorithm. It consists of the intermediate language instructions that represent the source language program. Each source language statement consists of a recognizable chunk of these instructions. With each chunk is the identifying line number of the source code. During the first modification, the program is subdivided into logical units designed as program segments. Most program segments will coincide with source program statements. Additional segments are added if functions are used or if multiple conditions are imposed in decision-making statements.

Each time the algorithm uses the intermediate language program as input, a modified intermediate language program is produced. The latest version is used as the next input in all cases. When there is a need to identify the particular intermediate language program, the terms "input intermediate language program," "segmented program," "reduced program," "realigned program," and finally "computer program" will be used.

3.1.2 Dictionary

This list is an initial input to the algorithm, and is scanned only once. Information is extracted for three lists: one list contains all the information necessary for a symbolic diagnostic capability. Another list is an abridged dictionary used for the internal processing. The last list is a dictionary trace that is used only during the first intermediate language

modification. This list provides the facilities for changing the operand references from input dictionary entries to the internal abridged dictionary entries.

3.1.3 The Abridged Dictionary

The abridged dictionary is produced during the initial processing of the dictionary. The entries correspond to the named values within the source language program. The primary information within these entries consists of internal information, such as type of data and storage requirements for the data. After the accession lists are processed, there is a pointer in each entry to the portion of the invariant value use list that pertains to the pointing entry. This dictionary is updated every time the intermediate language program is modified.

3.1.4 The Dictionary Trace

The dictionary trace exists only for the original translation of the input intermediate language program into a new form that reflects the new abridged dictionary. This list is generated when the dictionary is processed and is not needed after the input intermediate language program is processed. For each program flow entry, there is sufficient information to alter the intermediate program to point to program segments instead of the dictionary entries. The data entries provide the data type such that data conversion conventions can be inserted within the intermediate language. In effect, this list is the initial dictionary, with pointers to the abridged dictionary entries, so that the segmented intermediate language program references that dictionary.

3.1.5 The Segment List

The segment list is designed to provide the linkage between the analysis and the intermediate language program. A segment entry is entered in this list for each source program statement that is represented in the segmented program. Some source program statements are divided into more than one program segment. With each segment of the segmented program that represents a source language segment, there is a pointer to the segment list entry, and there is a literal containing the source language line identifier. Within each segment list entry there are pointers to the first and last entries within the segmented program that pertain to this list entry. Each list entry has a flag value. The next succeeding segments in the program flow are indicated by the pointers to other segment list entries. Another pointer in each list entry points to the connector list entry group that have this segment as a destination. Whenever the intermediate language program is modified, this list provides a guide to the modification, and is modified to reflect the program changes.

3.1.6 The Connector List

The connector list is used to represent the Boolean connection matrix. The list has an entry for each flow connection, from program

segment to program segment. Program segments that are never reached are not included.

Each connector list entry consists of a pointer to a segment list entry. The pointer designates the source and destination segments as entries in the segment list. The connector list is ordered by the destination segment since the list is used for backtracking the program flow of execution. Each entry of the segment list has a pointer to the group of connector list entries that have that segment as a destination. This list exists for two reasons: (1) the full Boolean matrix requires excessive storage, and (2) the list is easier to access than the matrix.

3.1.7 Accession List

The accession list is generated twice; while the intermediate language program is segmented and after each program segment is reduced. The list provides the information for the invariant value use list. Each entry consists of a pointer to an abridged dictionary entry, a pointer to a segment list entry, and a flag indicating whether the access represented is a fetch, a store, or both. There is one entry in this list for each intermediate-language-instruction access of a dictionary entry data operand.

3.1.8 Invariant Value Use List

The named values for which there is no value established during the program flow are assumed to be invariant for the program. A list of all the segments using each invariant value is constructed. A pointer to the appropriate entries within the invariant value use list is added to the abridged dictionary. The preset values are also located. An error condition exists if the values are not preset.

3.1.9 Block Access List

This list provides the order of data access within each program processing block after the relocated program is constructed. This list provides the intelligence for blocking data preliminary to assigning storage for the data values.

3.2 Processing Steps

The proposed optimizing algorithm consists of a series of related tasks. These tasks are executed in processing steps. The order of the steps and the tasks assigned to each step are dependent upon the input required and the output desired for the individual tasks of a step. The input and output requirements for the optimizing algorithm steps are presented in Table I. This subparagraph presents an overview of the steps of the composite algorithm.

TABLE I

OPTIMIZING ALGORITHM STEPS

PROCESS	STEP	INPUT	OUTPUT
Process Dictionary	1	Dictionary	Abridged Dictionary Dictionary Trace
Process Intermediate Language	2	Input Intermediate Language Dictionary Trace	Segmented Program Segment List Accession List
Process Accession Data	3	Segment List Connector List Accession List	Connector List Invariant Value Use List
Consolidate Program Intelligence	4	Segmented Program Segment List Abridged Dictionary Invariant Value Use List	Reduced Program Updated Segment List Updated Abridged Dictionary Updated Invariant Value Use List
Program Relocations	5	Reduced Program Abridged Dictionary Invariant Value Use List	Relocated Program Updated Abridged Dictionary Updated Invariant Value Use List Block Access List
Dictionary Realignment	6	Abridged Dictionary Block Access List	Updated Abridged Dictionary
Generate Code	7	Relocated Program Abridged Dictionary	Computer Program

3.2.1 Process Dictionary

The purpose of this step is to consolidate the source program dictionary information. The input for this processing step consists of the dictionary that was generated during the processing of the source language program. The output information consists of two lists: (1) an abridged dictionary, and (2) a dictionary trace.

The abridged dictionary is generated to produce a condensed dictionary version suitable for the subsequent optimizing process. The dictionary trace permits the alteration of the intermediate language program in the next processing step, at which time the intermediate language program is modified to reference the new abridged dictionary.

3.2.2 Process Intermediate Language Program

This step completes the task of extracting information from the source data. The input to the algorithm step to process the intermediate language program consists of the initial intermediate language program and the dictionary trace. The dictionary trace provides the data necessary for translating the intermediate language program to include the dictionary changes. Data operands that are represented by dictionary entries are changed to the abridged dictionary. Source language data conventions are eliminated by inserting conversion instructions in the segmented program. Program flow control is changed to segment references. In addition, the extracted lists reference the new abridged dictionary. These lists indicate the breakdown of the source program into program segments and the accession of dictionary information within these segments. Thus, the principle outputs are the segmented intermediate language program that reflects the abridged dictionary, the segment list, and the accession list.

3.2.3 Process Accession Data

This step in the algorithm determines the role of the individual named values of the source program. The input data for the processing step consist of the segment list and accession list that are generated from the intermediate language program.

A connector list is generated as an intermediate table in this process. This table is produced exclusively for this processing step. (Another connector list with the same format is generated later in the program.) Another output from this processing step is the invariant value use list. Pointers are added within the abridged dictionary to the corresponding entries within these lists. An additional task for this process is the retrieval of preset values from the source program. Appropriate error messages are initiated as needed.

3.2.4 Consolidate Program Intelligence

This processing step of the proposed algorithm is designed to execute the intermediate language instructions that involve invariant data. In the process, various lists are changed. A new intermediate language program is produced. The segment list is modified to reflect the new program.

New invariant values are added to the abridged dictionary and some values are deleted. Some of the variable values will become invariant. The entries within the abridged dictionary and the invariant value use list are modified to reflect the changes.

The primary purpose of this algorithm step is to eliminate unnecessary intermediate language instructions. Some instructions are deleted, since their operands are invariant. In this case, the instructions are replaced by their calculated results. Some statements are deleted because they are never executed. The deletion of unnecessary intermediate language instructions and unnecessary invariant data aids in program optimization. Less computer time is required to execute the resulting object program, and less storage is required to contain the object program; therefore, the storage requirement reduction increase the chances for rearranging the program to reduce page turning in the program execution.

3.2.5 Program Relocations

This step is the algorithm process produces program reorganization based on program flow. The intermediate language instructions are accessed by program segment. Changes are then made in the intermediate language instructions. The processing modifies all the entrant tables as well as the intermediate language program. A block access list is generated to specify the order of access for data within each of the resultant program processing blocks.

This program relocation moves program segments and intermediate language instructions from their original positions to locations in the program flow with lower activity. Thus, it is obvious that the program execution time is reduced. If the movement is from inside a program loop to outside the program loop, then the loop requires less computer instructions. Thus, less page turning is required for instructions. In addition, the execution of these program portions just prior to the loop tends to collect the data required for the loop execution into a block. Thus, if these collected data are properly allocated to storage, the execution of the loop will require less page turning for data.

3.2.6 Dictionary Realignment

The input data for this processing step consist of the abridged dictionary and the block access list. The process assigns the data values to storage blocks.

The use of subscripted items is analyzed to determine how these items are to appear in parallel and serial tables. The variable values are analyzed to determine groupings and to determine the values to store within the high-activity area. The invariant values are investigated to establish the type of storage for each value. This analysis is designed to initiate storage grouping for subsequent storage assignments, such that optimum assignments can be made for a page-oriented computer system.

3.2.7 Generate Code

This is the final step in the algorithm. The process selects the individual intermediate language instructions for translation to computer code. Program relocation is included as allowed by parallel operation considerations.

During this processing step, the analysis is performed to determine the final execution order for the elements of the processing block. This order is designed to minimize page turning. The instruction sequences are generated as if they started with a displacement of zero from a base location. After these sequences are generated, they are assigned to page areas. Intermediate instructions are then provided to supply the link between these program sequences, and invariant data blocks are assigned to unused page areas. Loops are aligned to occupy a minimum number of pages. Some instruction sequences will be repeated to avoid unnecessary page jumping. Calls to advise the executive routine in advance of upcoming requirements are inserted.

3.3 Compiler Outputs

The outputs generated by the compiler process is produced as two groups.

The first group consists of the object code. This output is the translated program ready for execution as computer code. The mode has been optimized for execution on a page-oriented computer system.

The second group of outputs consists of the program information gleaned from the program during the compilation. This output consists of printed lists that provide program documentation for program maintenance.

3.3.1 Optimized Computer Programs

The compiled computer code is produced in two sections: (1) a directory, and (2) the computer code.

The computer code is optimized for execution within a page-oriented computer system. This code exists as chunks of executable computer code. Each chunk is relocatable at load time. If a code chunk is reloaded, the second load location need not be the same as the first load location.

The directory maintains a record of the status of the computer code during execution. This directory contains some page-oriented executive function subroutines. The computer software system loads the directory, and the directory loads the required computer code chunks as needed and supplies the primary base location values. The advice-to-executive subroutines are contained in the directory.

3.3.2 Compiler Printed Output

The printed output of the compiler is selective. The specifics of the pertinent printed lists are contained within the sections of this

paper concerning the information within the individual lists. The different lists include:

1. The input program.

2. The invariant value use lists.

3. The statement label use list.

4. The program code.

5. Miscellaneous program exceptions.

4.0 PROPOSED PROCEDURES

The proposed compiling algorithm produces computer programs that are optimized for page-oriented computer systems. Extensive data collecting and processing is required prior to any optimization. An integrated scheme of data collection is used to avoid duplication of effort in the process. This section supplies details concerning the individual procedures. The input to the algorithm consists of the intermediate language program that has been parsed from the source language program and a dictionary that contains entries for the named values.

4.1 Extract Dictionary Information

The dictionary supplies valuable problem intelligence for the optimizing algorithm. The specific entries were added in a haphazard manner when the dictionary was constructed. This initial task introduces order into the dictionary functions. This order is incorporated within the intermediate language program during the processing step that extracts information from the intermediate language program. The extraction of dictionary information is the first algorithm step, because the dictionary rearrangement is independent of the intermediate language program, and the converse is not true. The dictionary information is reproduced into standard formats. Two lists are generated during this process: (1) an abridged dictionary, and (2) a dictionary trace.

The dictionary entries are partitioned into blocks according to type of entry. Three partitioned types are recognized: (1) flow entries, (2) data entries, and (3) structure entries. The information contained within each block is added to the lists in a format dictated by the type of entry in the block.

One partition feature of the dictionary contains the program flow entries. These flow entries contain information concerning statement labels, procedures, functions, and switches. Assigned to these flow entries are segment numbers, which identify the intermediate language instructions that follow the flow identifier. The segment numbers replace the flow branch operands for the segmented intermediate language instructions. This flow information is not required in the abridged dictionary. The remaining flow information is incorporated in the segmented intermediate language program during the next

processing step. Thus, this information plus the segment number is included in the dictionary trace.

The dictionary entries concerning data and data structures form the remaining dictionary partitions. Simple items are separated from subscripted items. Aside from grouping the entries pertaining to each structure, there is little processing that can be done during this step. Overlays and fully defined structures are labeled within the abridged dictionary for future use. An inspection of the intermediate language is required to separate the variable data from the invariant data.

Additional information available in all dictionary data entries is the specification of data type. The dictionary trace includes this information such that the data conversions conventions can be inserted in the intermediate language program. The conversions are added to the segmented intermediate language program during the next processing step at the same time as information is extracted from the intermediate language. Thus, the source language data conversion conventions are quickly assimilated with no further concern. The identification of each entry has been essential during the parsing.

4.2 Extracting Intermediate Language Information

The intermediate language program provides vital problem intelligence for guiding an optimizing process. A primary purpose of this processing step is to establish program segments. These program segments represent sequences of intermediate language instructions. The program flow analysis is primarily in terms of these program segments. This processing step produces a list relating the data accesses with these program segments. A segmented intermediate language program is produced to reflect the abridged dictionary and the program segments. Data conversion instructions required by source language conventions are included within the segmented intermediate language program.

The problem flow is reduced to program segments, and none of these program segments includes more than one source language statement. Each conditional comparison of compound conditional statements becomes a program segment. Functions within assignment statements become special program segments that split the source statements into the program segments that precede the function and those that follow the function. Each program segment has an assigned number. The segment number assigned during the dictionary processing is used for labeled source statements. The segmented intermediate language instructions refer to program flow by these segment numbers. For each segment list entry, four values are obtained. Two of these values specify the first and last instructions of the program segment within the segmented intermediate language program. The other values specify the normal continuing segment and the alternative segment if it exists.

The intermediate language program is altered to form the segmented intermediate language program, and individual instructions are processed in groups by source statement. Operand pointers to dictionary entries are transformed to reference the abridged dictionary. Operand pointers to intermediate language results are modified to agree with the segmented intermediate language program sequence.

Data conversion instructions are inserted as required by source language data conventions into the segmented intermediate language instructions.

The operand pointers to dictionary entries initiate the generation of accession list entries. This accession list relates the abridged dictionary data pointer with the segment number. If the accessed data items are part of a completely defined structure, the dictionary pointer will indicate the structure. If an overlay declaration is involved, the lower dictionary index is used. A coded flag indicates the type of access--whether fetch, store, or both.

4.3 Process Accession List

The accession list contains an entry for every access to the values represented by dictionary entries. Each accession list entry consists of a pointer to the abridged dictionary entry, the number of the program segment containing the access, and a coded type-of-access flag. This list supplies the intelligence necessary to determine if a value is variable or invariant within the program execution. The accession list is ordered by dictionary pointer in order to group the accesses by dictionary entry. If an access is a store within a dictionary entry group, the value is assumed invariant. Source program overlay statements combine dictionary entry groups before this determination. The accesses from statements that are never executed are eliminated. The invariant values are tabled and reported as a list.

4.3.1 Establish Flow Connectors

An initial task is to establish a connector list which is extracted from the segment list as directed segment pairs. These connectors represent all the possible paths of execution of a problem solution. Extraneous paths are not included.

The connector list is established by tracing the problem execution flow as designated in the program segment list. The problem segment list entries are marked as they are used. Two pointers are used to trace the flow; one designates the next available connector entry cell, while the other designates the next connector path to trace. The program segment list entry designated as the destination in the connector list is investigated. If the entry is not marked, the one or more exits from the program segment generate the corresponding connector list entries to be added. The generation terminates when the two pointers are to the same entry, that is, when all paths to be traced have been traced.

The extraneous program segments are the program segments that correspond to the unmarked program segment list entries. The data accesses from these segments are eliminated from the accession list. Thus, the accesses from the extraneous statements do not unduly influence the optimizing process. After these entries are deleted, the accession list is ordered by dictionary pointer. Thus, the accession list entries are grouped according to the dictionary entries.

4.3.2 Invariant Data

If no store exists for a value group, the data are invariant within the program flow. A list of the invariant value uses is maintained for subsequent listing. A test determines if preset values exist for all the invariant data.

4.3.3 Extraneous Named Values

Occasionally, some data dictionary entries exist for values that are never accessed. This condition occurs either for truly extraneous data declarations or for named values accessed only from within extraneous segments. The dictionary entries that pertain to these values are deleted from the dictionary in the subsequent optimization processing.

4.4 Consolidate Program Intelligence

Before a detailed analysis is made of a program execution flow, the collected intelligence is analyzed. The information is adjusted to reflect the interdependence of the information. Program instructions that can be executed at compile time are replaced by their results. The invariant value use list is expanded to include the results of executed instructions and contracted to reflect the elimination of instructions and their operands. The program execution flow is compacted to reflect the elimination of the executed instructions. Finally, extensive reports are generated to reflect the compiler's knowledge concerning the object program.

4.4.1 Compile Time Executions

As a first step of the program consolidation, the intermediate language program is simplified. The intermediate language instructions that can be executed at this time are replaced by their results, and each program segment is processed independently. Extraneous statements are ignored.

The intermediate language instructions are executed by the interpretive routines. Each instruction execution depends upon the specific operation involved. As an example, an arithmetic operation involving two invariant values would result in a constant value. This simplification is also used to reduce evaluations where one of the operands is some special value such as zero or one. For example, an instruction to multiply a variable by one is replaced by an access of the variable. Each program statement is simplified independently, since no cross reference for intermediate results occurs between program statements. After a statement is simplified, a reduced intermediate language instruction sequence is generated for the prior instruction sequence. An accession list is generated that designates all the variable values accessed from the reduced intermediate language program by segment. As each use of an invariant value is eliminated, the corresponding entry in the invariant value use list for the value should be deleted. If all uses of an invariant value are deleted, the value is eliminated from further considerations.

Additional invariant values are added to the dictionary by this evaluation, and corresponding entries are made in the invariant value use list.

If an instruction assigns a constant value to a variable, additional simplification is possible in other program segments that use the variable. If a fetch of the variable is dependent only upon this particular store, then the segment containing the fetch is subject to additional execution simplification. The fetches that occur of preset variable values only are treated as fetches of invariant values.

During this process, a list of the null program segments is generated. In some cases, program segments become null. Additional null segments exist in the original problem. All unconditional transfers of control are null segments since they consist of an exit only. Additional null segments are generated when conditional branching segments involving invariant data become unconditional transfers during the compile time executions. The connector that is to be deleted is marked for subsequent attention. New extraneous statements can be generated by this process.

4.4.2 Consolidate Program Intelligence

Before proceeding with the optimizing process, the segment list is adjusted by the deletion of null segments. Each non-null program segment is checked to determine if an exit is to a null program segment. Each exit to a null program segment is extended through all null program segments until the exit points to a non-null program segment.

Another task is the creation of a new connector list. Beginning with the starting program segment, a connector is generated for each segment exit. Segments that occur as destinations in these connectors are used in generating connectors for each of their exits. A number is attached to each connector to indicate the number of alternative exits from the origin segment of each connector. When this connector list is completed, it includes all the execution paths in the problem solution. This newly created connector list is sorted by destination to meet the subsequent optimization processing requirements.

At this time, all extraneous program segments are known. This information is translated into statement numbers and from there into input line numbers. Thus a printed listing of all extraneous source problem statements is readily available. The null statements in this list are marked as null.

4.4.3 Reported Intelligence

At this time, the compiler has collected extensive intelligence that should prove useful to the programmer. The following is reported as a minimum:

a. An invariant value use list of the original program accesses for each of the invariant values.

b. The extraneous named values.

c. The compiler executed instructions, with their results. This list is ordered by source language statement.

d. The statements that exit to each of the labeled source program statements.

e. The statements that are extraneous to the execution of the problem solution.

4.5 Construct Processing Blocks

The initial stage in establishing processing blocks collects program segment into strings for the initial processing blocks. Later, these processing blocks are extended by appending decision blocks as elements within the processing blocks. If a decision block can be appended on two processing blocks, then the processing blocks are concatenated with the decision block as an element that separates the elements from the two original processing blocks.

The connector list and the segment list are used in the initial assignment of program segments to processing blocks. During this process, a program blocking list is started. The first list entries indicate the first and last elements within processing blocks. A processing block is formed by identifying a starting element and appending all the succeeding segments that form a linear string. That is, segments are added that have only one entrance path and only one exit path. The exit path condition is established from the segment list. The entrance path condition is established from the connector list. The connectors that lead into or out of the processing block are altered to indicate the program blocking list entry rather than the first and last program segment list entries.

A starting program segment of a processing block is characterized as having one exit and being either a collection point or a first element following a branch point. Therefore, two methods of identification of first segments are required. The first method checks for program segments that have multiple path entries in the connector list and only one exit in its segment list entry. The second method checks the program segment list for multiple exits. Each exit leads to a first element of a processing block or to another program segment with multiple exits.

When this process is completed, the remaining program elements are the processing block elements and the program segment elements with multiple exits. All the program segment elements with single exits have been incorporated into processing blocks. These processing block elements are designated by program blocking list entries.

These remaining elements form a massive decision block that encompasses the entire problem solution algorithm and is divided into minimum decision blocks. As each minimum decision block is formed, it becomes the sole element of a new processing block. This new processing block is concatenated with the adjoining processing blocks if no alternative entrance exists within the

interior of the potential processing block. Processing blocks are concatenated by setting the exit of the last element contained within the first processing block to point to the first element of the second processing block. Then the last element indication of the first processing block is changed to the actual last element from the second processing block. The second processing block is dropped as a program blocking list entry.

4.6 Construct Decision Blocks

Decision blocks are designed to contain a minimum number of elements. A decision block has one common entry segment within the block and one common exit segment outside the block. Each decision block becomes an element of a processing block, possibly the only element of that processing block. The decision block elements are program decision segments and processing blocks.

The search for the elements that comprise the individual decision blocks uses the connector list, the segment list, and the program blocking list. The connector list is used in determining the predecessors of an element, while the segment list and the program blocking list are used in determining the successors of the decision segments and the processing blocks respectively.

When a decision block is isolated, a decision block list entry is created within the program blocking list. This list element contains pointers to the first element of the decision block and to the exit element of the decision block. The connector list locates all the elements that exit from the decision block. These elements are flagged as decision block exits. The connector list entries are marked as decision block exits and the destination pointers within these list entries are changed to reference the decision block list entry. This decision block becomes the sole element of a new processing block. A new processing block entry is added to the program blocking list. This processing block entry points to the decision block entry as the first and last elements of the processing block. The destination pointers of all the connectors that lead into the first segment are changed to pointers to the new processing block list entry. The source pointers of these connectors locate the elements for which the exits are changed from the first element of the decision block to the new processing block. The resulting processing block is concatenated with its neighbors, if possible. The resulting element replaces its constituents before the search for decision blocks is continued.

Thus, the connector list entries, segment list entries, and program blocking list entries that designate a processing block are isolated within the lists. These list elements are accessed by the decision block list entry in the program blocking list. The designated decision block becomes an element of a processing block that is an element of the remaining program flow.

The decision block determination begins with a forward trace from the initial program element on all paths of problem flow to the first program elements that have alternative entrance paths. If two or more paths are traced to one program element, the paths are backtracked to determine if a decision block precedes the collection point. The connector list is used to backtrack to the next prior decision segment. If two or more paths are traced back to a common

decision segment with no intermediate dangling paths, then a decision block is isolated. If there are intermediate dangling paths, then the flow is traced backwards from this decision segment to the next prior decision segments before repeating the testing. Backtracking is terminated on decision segments that are not reached on all their exit paths.

If the replacement of a decision block element eliminates a collection point, the program-flow forward trace is extended to the next collection points. The forward trace is also continued if no decision block is found and if all the entrance paths are traced. Eventually, an impasse occurs if any loops exist within the program flow. The program execution flow forward trace from these elements is the same as above. Backtracking terminates at these program elements. If the program flow track reaches a collection point that has been traced beyond, then this collection point may complete a program loop. After the normal decision block testing is completed, additional tests determine if a loop exists. The flow is backtracked from the suspected element. This reverse flow trace can be past decision segments that exist as potential loop exits. Backtracking is terminated as the collection point elements for which all entrance paths have not been traced. If the program loop has only one exit, then the entrance paths are checked to determine if a common starting program element exists with no alternative dangling side paths. If so, the loop and its entrance paths back to the common starting element form a decision block to be isolated. If the program loop has multiple exits, then these multiple exits must reach a collection point before the existence of a decision block is determined.

At this point in the optimizing process, the individual elements are in a sequence that reflect an acceptable order of execution for the program. This order can be altered within limits without changing the resultant values of the stated problem solution. Data dependencies establish the limits of reordering the execution with no change in the problem solution. The remaining analysis determines the specific changes within these limits that optimize the resultant computer program.

4.7 Move Program Segments

The initial analysis that determines the favorable program movements concerns decision blocks only. A pushdown table is employed to maintain positions of postponed analysis. The analysis starts with the all-encompassing processing block that represents the entire problem solution. As sub-blocks are found, the current block analysis is tabled and remains tabled until all the sub-blocks are processed. Thus, the inner block analysis is completed first. The most inner blocks are processing blocks that contain program segments and no decision blocks as elements. The analysis relies on the fact that the decision blocks are elements of processing blocks and contain processing blocks as elements.

The techniques of the initial analysis depend upon the program execution flow within decision blocks. Attempts at gross improvements precede finer adjustments. The first optimizations move program segments to processing blocks of lower activity than the original processing blocks. Examples include moving

statements from loops and moving statements from absolutely executed to conditionally executed processing blocks. A tentative processing block is established for the moved statements. If possible, this block is concatenated with other processing blocks. The gross improvement analysis does not involve any program division smaller than a segment. Thus, the individual intermediate language instructions are not accessed for this analysis level.

The analysis depends upon the form of the decision block being analyzed. The movements are limited by the processing blocks that are elements of the decision block and by the processing block that contains the decision block as an element. The analysis of a decision block with no loop differs from the analysis with a loop. Data dependencies determine the program segments that can be moved.

4.7.1 Restrictions on Moving Statements

The range of possible positions for a program segment depends upon the processing blocks. The possible positions within this range for the program segment within decision blocks depends upon the program flow. Each element of a processing block is executed if the processing block is executed. The order of execution of the individual elements within a processing block is independent of the program flow. The primary reordering restriction within processing blocks is the obvious restriction that a value must be established before it can be used.

Within the decision blocks, the movement of program segments is complicated by the conditional execution of some elements and the repeated execution of elements in a loop. The normal movement within a decision block transfers a program element from one processing block to another processing block within the same decision block. If a program element is moved to a position where its execution is unconditional whenever the decision block is executed, that program element is transferred to the processing block that contains the decision block as an element. This program element is eliminated from the decision block and becomes an element of the processing block containing the decision block in parallel with the decision block.

Data dependencies and program flow restrictions are considered before a program segment is moved from one processing block to some other processing block. An advanced calculation must follow the evaluation of any of its constituent operands. A delayed evaluation must precede any new setting of values for the variable value operands. Established values must be calculated before they are used. An evaluation in a new position must not adversely affect calculations on any program flow path. This condition is verified by the execution flow within the decision block.

4.7.2 Criteria for Moving Statements

A problem statement is moved from one processing block to another processing block to create a more efficient object program. An obvious gain occurs in moving a program element to a processing block with a lower frequency of execution than the originating processing block. Various levels of activity are readily apparent for the elements within the program

flow of a decision block. The more active elements are the loop elements. The less active elements are the conditionally executed elements that occur after decision segments. Additional levels of activity occur for combinations of these conditions.

Since all the elements of processing blocks are executed, all processing block elements will have the same level of activity. The interchange of these elements reduces temporary storage requirements and in consequence reduces page turning. This optimization is achieved only after additional dictionary intelligence is available.

4.7.3 Decision Block Intelligence

Before program segments are selected for movement, the available program intelligence concerning the applicable decision block is processed. The initial program intelligence consists of the accession lists for the variable values that are accessed during the execution of the decision program segments and of each element for all the processing blocks that are contained within the decision block.

The processing of accession data is equivalent to the method described in section 4.3 (Process Accession List). The major difference is that the frame of reference becomes the decision block instead of the entire program. Invariant values specify that the variable values are invariant with respect to the decision block. The data details are carried to the depth of the elements of the processing block contained within the decision block. Hence, this analysis produces an accession list of its data requirements for use in outer level block analysis. Data used only interior to this block are not included in the list. The type-of-access flag is expanded to include changed and conditionally changed values.

The approach to flow analysis is direct. Each element is analyzed in turn to determine its role in the program flow within the decision block. The entry element is the first element inspected.

Three operation modes exist in the flow analysis: (1) a forward trace finds non-loop elements; (2) backtracking isolates a loop; and (3) a second backtracking identifies the loop elements. The initial entry element is used to determine which mode is first used. If the first element is a loop element, all the elements of the loop are isolated before continuing. If the first element is not a loop element, all the initial non-loop elements are analyzed first.

The forward trace finds the non-loop elements. The exits of the traced elements locate succeeding elements, elements with alternative exit paths to trace are tabled, and the traced connector is flagged. If the entered element is the destination in more than one connector, and if all the applicable connectors are not marked, the tabled elements supply another eligible path. The element is tabled as a merge element the first time this occurs. If all the pertinent connectors are flagged, then the element is removed from the table, flagged as a non-loop element, and used to continue tracing. When all paths

are traced, this mode of operation is terminated. If elements with unmarked entrances remain in the connector list, then a loop exists in the decision block.

The backtracking that isolates a loop is required if any elements remain from the proceeding analysis. Since the connectors that lead outside the loop in a backward direction are marked by the prior analysis, they are easily avoided. The backward trace starts from all the elements remaining from the above analysis and continues until all paths reach another of these elements. An investigation is required to isolate all elements that mutually precede and follow each other. This operation mode isolates one loop. The loop elements are flagged during a second backward trace. The exit paths from the loop are tabled at this time. The forward trace continues from these exits to find additional non-loop elements.

4.7.4 Decision Block Statement Moves

Each decision block element is classified in two ways: (1) loop or non-loop and (2) decision program segment or processing block. The individual components that enter into this analysis are (1) the decision program segments, (2) the individual elements of the processing blocks contained within the decision block, and (3) the elements that precede the decision block within the processing block that contains the decision block. Optimization at this level moves these components to positions of lower execution activity than their original positions. These moves include the removal of program elements from loops.

Some components are moved from a position before a decision program segment to a position after the decision program segment. This move is possible if the element's execution is not required on all exit paths of the decision program segment. An element may be removed from a loop past the exit test in this process. Program elements contained within loops are removed from the loop if their repeated execution is redundant.

4.8 Factor Common Expressions

The next task analyzes the intermediate language instructions. This analysis results in an instruction reordering with the insertion and deletion of some instructions. The task is executed for each decision block immediately after the statement decision block moves are completed. The first steps in the analysis apply to the instructions of each processing block within the decision block.

The elements being analyzed consist of the intermediate language instructions within the program segments of the processing block. Later, the decision block is analyzed and the elements then consist of the intermediate language instructions within the decision segments and the instructions that result from analyzing the processing blocks that are contained within the decision block. A realigned intermediate language program is produced at this time.

4.8.1 Processing Block Analysis

The decision blocks that are elements of the processing block under consideration are analyzed and optimized before the processing block is processed. The process manipulates those decision blocks that are elements of the processing block as inflexible sequences of intermediate language instructions. These elements are repositioned within the flow of the processing block as units. Their movement is limited by data restrictions.

The program segment elements of a processing block are analyzed after the decision block containing the processing block is analyzed for statement moves. The intermediate language instructions within these program segments are collected into one sequence. Special intermediate language instructions designate the decision block elements. The major optimizing processing depends upon where the processing block occurs within a decision block. The initial processing is independent of the covering decision block. Pairwise comparisons are made on the intermediate language instructions of the block to determine the instructions that are identical and require the same operand values. Excess instructions are deleted, and the remaining instructions are modified to accommodate the deletion.

4.8.2 Decision Block Analysis

The preliminary processing block analysis is completed for all the processing block elements of a decision block before the decision block is analyzed. In contrast to the preliminary processing of the processing blocks, which is an intra-block analysis within one processing block, the initial decision block processing is an interblock analysis between the processing blocks that are the constituent elements of the decision block under consideration.

This initial process factors common expression sequences from two parallel processing blocks. The factoring method depends upon the location or position for the sequence. The sequences are inserted before the decision segment at which the program flow diverges or after a point at which the program flow merges. If more than two parallel processing blocks occur, then the factoring is accomplished in a pairwise manner. In this factoring process, the instructions within the computation portion of a decision segment are also included. The actual decision instruction remains unless all the alternatives on both paths are identical.

If a program loop is involved within the decision block, an attempt is made to remove all loop invariant calculations. Tentative intermediate language instructions are generated for the pre-loop access of these values. If the loop has multiple entrance paths, the instructions that are removed are inserted such that these intermediate language instructions are executed regardless of entrance path used.

4.9 Dictionary Realignment

The next step in the processing realigns the dictionary according to the classification of the values that are represented. Within the classifications, the dictionary entries are blocked according to the order in which the values are used within the program execution. One classification consists of the program constants and the invariant values that are accessed by nonsubscripted names. Dictionary entries that represent values with duplicate representations are combined in this process. Another classification consists of the variable values that are accessed by nonsubscripted names. For ease of symbolic reference, these dictionary entries are ordered alphabetically. The remaining two classifications of dictionary entries represent the values called by subscripted names. These entries are classified according to whether the represented data values are variable or invariant. The structure values for dictionary entries cannot be completely divided if completely defined tables exist or if data overlays are declared.

The subscripted named values allow the least processing; hence, they are processed first. The nonsubscripted variable values affect the problem flow and are stored with variable tables. The invariant data occupy whatever storage space is available. Since invariant data have the most flexible storage requirements, their analysis is last.

4.9.1 Divide Subscripted Data

The preliminary processing of subscripted data divides these data into two groups according to whether the data values are invariant or variable within the program flow. Data involved in overlay declarations are all placed together in the same group. This group is placed in the invariant data group only when all the values are invariant. The same requirement exists when the data are from a completely defined table. This first division produces two structure groups: (1) the subscripted named values that are invariant throughout the program, and (2) the subscripted named values that are changed during the program execution. Structures that have subscripted named values in both groups are considered as two separate structures in the remaining analysis.

Separation for subscripted named values is also done according to the use in distinct loops where the subscripts are loop-dependent. These items are grouped during the loop analysis. Parallel substructures are constructed to represent this grouping. Thus, when the loops are executed, the required values will be in close proximity to each other. All subscripted named values that are accessed from a single loop are tabled as serial entries within one structure. The remaining values are in other parallel structures. Two loops using some common subscripted named values complicate the situation. In the case of invariant data, parallel tables can be used. Common values can be duplicated if the storage requirements are not increased substantially. If excess storage is required or if the data are variable, one structure is used to include all the values. Subscripted data not referenced within loops are maintained as serial entries within common structures according to the source program declarations.

4.9.2 Group Variable Value

The manipulation of variable values is designed to group these data such that the fetching and storing of data is a flow from group to group that parallels the program execution. The accession lists that are generated when the intermediate language program is realigned provide the initial intelligence concerning accesses and order of access. The variable value subscripted items are included within this analysis. A structure is treated as a unit with extensive storage requirements. Values that are in structures are accessed from these structures. Some values are closely related to these structures and are stored with the structures.

The analysis begins from the innermost loops of the program and proceeds outward. The goal is to locate the variable values in storage such that the values are readily accessible during the execution of the loop. Various storage methods are utilized to satisfy the differing access requirements. Some values are accessed from fixed storages, which others will be accessed before the loop is executed. This prior access involves precalculations for instructions that use loop-independent operands. The values are grouped such that the type of access is optimized. Variable values that are used only within one loop or section of code are either assigned to the high-activity temporary area or to the fixed storage area. The storage method depends upon whether the value is saved from one loop execution to the next loop execution.

4.9.3 Group Invariant Data

The process that groups the invariant data begins with a check for values with duplicate representations. This first step compares the values pairwise to determine if duplicate value representations can be consolidated. Thus, there is some reduction in the storage requirements for invariant data.

Subsequent steps are designed to arrange the invariant data for an orderly access of the values in parallel with the flow of program execution. These remaining tasks reduce computation time by properly assigning the data to storage. Counts are accumulated to determine the number of program segments that access each value. The values that are most frequently accessed are grouped. The values that are used very seldom are included with the program sections that access that data. In some cases, the invariant data are duplicated, if the data value is randomly accessed or if the data value belongs in two logical program sections.

4.10 Computer Code Generation

At the time computer code is generated, the actual storage assignments are unknown. The registers that will contain the base location values are assigned at this time. The displacements of data from the base locations are known for the computer code being generated. Computer code is compiled relative to base locations. The location values are contained in assigned registers. Again, the displacement is the only compiling requirement. In

fact, an entire compilation can be made with the base location values considered as variables that are assigned at execution load time. It is easy to generate the object code before determining where this code is located. As an example, a loop is compiled before determining the loop's position on a page. After the compilation, the loop is positioned to fit on a minimum number of pages. If there is an excess number of cells remaining on a computer page, some of the invariant data are located with the instructions.

The sequence of selecting the intermediate language instructions for translation to computer code is from the innermost program sections outward to the entire program. This order is the same as the order that the program sections were selected to factor common expressions. Thus, the process to factor common expressions has rearranged the intermediate language instructions into this order.

After the sequences of computer code are generated, the various sequences are combined into page segments. Between these page segments additional instructions are inserted to adjust registers for base locations. In addition, calling sequences are inserted to the subroutines that advise the executive routine in advance of the program requirements. Counting instructions are also inserted. These counts are available to the executive advising subroutines to provide the capability for a dynamic determination of alternative page loading.

5.0 SUMMARY

This section has presented a means of manipulating the program intelligence available from an intermediate language program with an associated dictionary. The procedure as outlined will produce more efficient code than is produced by most existing JOVIAL compilers. An object program produced by this process should prove to be effective and efficient when operating in a page-oriented operating system.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

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

1. ORIGINATING ACTIVITY *(Corporate author)* System Development Corporation Black River Blvd Rome, New York 13440	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE
An Investigation of Advanced Programming Techniques

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final - July 1967 - June 1968

5. AUTHOR(S) *(First name, middle initial, last name)*
Richard M. Dobkin

6. REPORT DATE October 1968	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO. F30602-67-C-0321 b. PROJECT NO. 5581 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* RADC-TR-68-367	

10. DISTRIBUTION STATEMENT
Each transmittal of this document outside the Department of Defense must have prior approval of RADC (EMIIF), GAFB, NY 13440.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center Griffiss Air Force Base, New York

13. ABSTRACT

The objective of Part I of the study described in this document was to perform two services. The first service was to investigate four existing JOVIAL compilers to determine which had the greatest potential for conversion to the GE-645 computer. The four compilers were the ones currently in operation on the CDC-1604B, the IBM 7090, the IBM 360, and the GE-635. The second service was to investigate and evaluate the advantages and disadvantages of incorporating certain features into a compiler which would operate on the GE-645 under the control of the MULTICS supervisor. These features included the production of programs with reentrant code, on-line compiling, partial compilation capabilities string processing, advanced system and program compool features, on-line debugging aids, segmentation, and binary versus symbolic output. The study and the conclusions reached were made by comparing and evaluating the needs of the compiler with available system procedure and interface modules. In regard to an existing compiler, it was recommended that the CDC-1604B JOVIAL compiler be selected for conversion to the GE-645. As to the features to be incorporated, it was recommended that all the features be implemented with the following exceptions: that there be no batch compilation capability; only a limited partial compilation capability be made available; there should be no string processing in the initial version; only a small number of the on-line debugging aids be initially available; and that only binary output be produced. (Over)

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14. KEY WORD	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
Computer Programming,						
Compilers						
Paging						
Segmentation						

(Continuation of Abstract)

The purpose of Part II of the study described in this document was to investigate the concept of Paging for the purpose of establishing techniques for generating code that operates effectively in the GE-645 Paging System. There were two major objectives of this investigation. The first was to determine if the code generation process for paging could be automatic (handled by software) or if present programming techniques should be altered to produce efficient code generation. The second objective was to define an implementation approach which will allow a rapid implementation of a Paged JOVIAL Compiler and the transfer of existing JOVIAL programs to the GE-645. The study was accomplished by making an investigation of which language types would be most efficient for paging, and how a program can be structured for most effective operation in a paged environment. The conclusions reached during this study are detailed in Sections II and III of Part II of this report.

UNCLASSIFIED
Security Classification