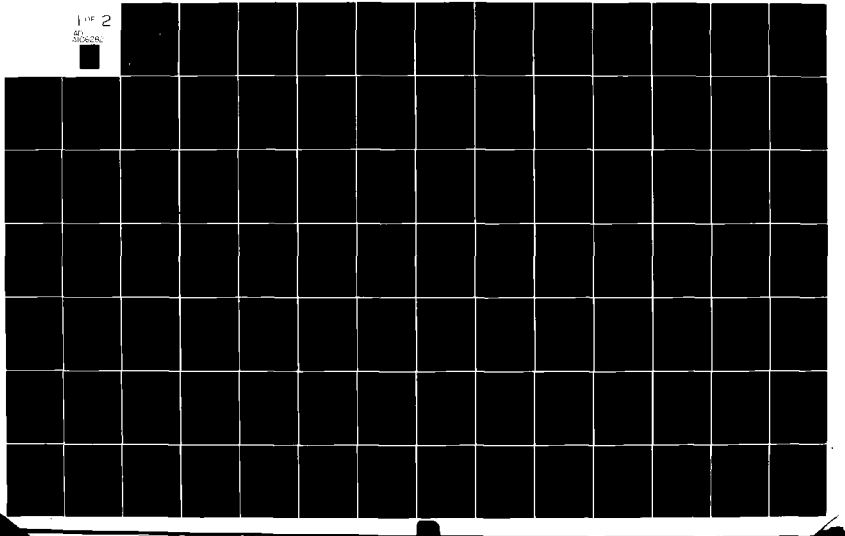
AD-A106 282 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2
THE SOURCE TO S2K CONVERSION SYSTEM.(U)
DEC 78 J L STEVENS
UNCLASSIFIED AFIT-CI-79-262T NL



THE SOURCE TO S2K CONVERSION SYSTEM

by

JONATHAN LEE STEVENS, B.S.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1978

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 79-262T	2. GOVT ACCESSION NO.: AD-A106 282	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): The Source to S2K Conversion System	5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Jonathan Lee Stevens	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: The University of Texas at Austin	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433	12. REPORT DATE: Dec 1978
	13. NUMBER OF PAGES: 107
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASS
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

OCT 1981

Fredric C. Lynch

FREDRIC C. LYNCH, Major, USAF
Director of Public Affairs
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

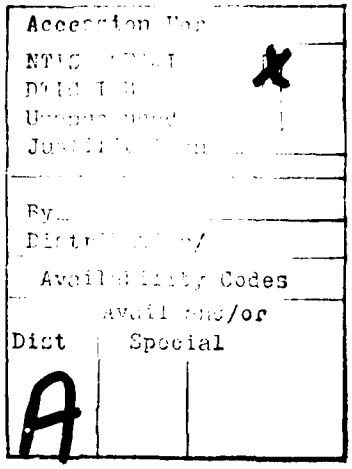
ATTACHED

81 10 26 222

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



ABSTRACT

The most common method of creating and loading a new database is to write a program using the host language macros the database management system provides. As for all software production, the cost of writing this program is high, particularily considering it may be executed only once. The "Source to S2K (System 2000) Conversion System" will generate a FORTRAN program which will load the described source file into the described System 2000 database. The user inputs these file descriptions and the source to target mapping transformations. The system's design is based on a common architecture developed through research of seven current conversion system implementations. This report will present this architecture, detail the design and languages of the "Source to S2K Conversion System" and comment on its implementation. Appendicies include a User's Manual, and examples of generated command files. The system has been implemented in PASCAL in a Control Data 6000 series computing environment.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

CHAPTER 1

Introduction

1.1 Statement Of Problem

An application which stores data under the control of a database management system (DMS) must initially "load" its data. The process of loading this data can be viewed as a data conversion problem -- how to convert the raw application data from its present form and format to one which is required by the DMS. Most established DMSs provide two initial load capabilities [10]. The first is an automatic load function. Typically this requires the raw data to be in a specific format, usually with delimiters surrounding each field value. In addition, the load function usually requires more computer processing time to load the same amount of data than the second conversion method -- a user written program. The user written program utilizes the DMS's file manipulation macrocommands in one of several host languages. A user written program also enables the user to include validation and conversion routines. These routines rarely are part of the DMS's load function. Even though

there are major advantages in processor time and flexibility, the user written program is expensive to produce, especially if it is executed only once. A method is needed, therefore, which will allow a simple, flexible, and cost effective means of converting initial load data into its underlying DMS data structure without requiring any special data formatting or high software production costs.

1.2 Report Objectives

The objectives of this report are to document the design, implementation and correct usage of the "Source To S2K Conversion System". This system is a solution to the previously stated problem. It is a simple, flexible system which allows automatic generation of initial load programs for MRI's System 2000 (S2K) database management system. Because it generates a complete FORTRAN program, it has the advantages of efficient processor utilization, validation and conversion routine capabilities, and no special formatting of the input data. Because the program is generated from a small amount of user input, it is also cost effective, compared to writing the program by hand.

As all systems, however, the Source To S2K Conversion System has its limitations. The generated program can read in only one input source file at a time. Subsequent programs can be generated which will allow updating of the initial database, but from only one file per program. The target database must be an S2K defined database. Because S2K supports a hierarchical data model, the data transformations from the input source file to the target database are also based on a

hierarchical data model. As a consequence of this design, the input source file must be describable in a hierarchical manner. All validation and conversion routines must be written by the user in ANSI Standard FORTRAN. Within these limitations, however, lie a great number of source file to target S2K database conversion capabilities.

1.3 Design and Implementation Objectives

The design and implementation objectives of the Source To S2K Conversion System were not to develop new approaches or methodologies. The objectives were to study existing data conversion implementations, gleaning from them the required components and functions of a conversion system, design the source to S2K system based on this research, and finally, implement the system using disciplined, structured software engineering principles. In order to properly document how these objectives were accomplished, the remainder of Chapter 1 will discuss a common architecture for data conversion systems. This architecture was developed from the study of seven different data conversion system implementations. Based on this common architecture, the Source To S2K Conversion System was designed. Chapter 2 will report this design and details of its implementation. Appendix A is the system's User's Manual. It also contains execution instructions and a complete example. Appendix B is an example of generated UT-2D command files needed to execute the system.

1.4 A Common Architecture For Data Conversion Systems

Seven different data conversion implementations were studied to find their common functions and components. The implementations studied were: IBM's EXPRESS System, 1977 [19]; SDC's CODS System, 1975 [3,18]; University of Michigan's Data Translation Project, 1976 [6,11,12]; J.A. Ramirez's (University of Pennsylvania) Conversion System, 1974 [15,16]; CODASYL Stored Data Definition and Translation (SDDT) Task Group COBOL-TO-NIPS/360 Prototype, 1973 [7]; Honeywell's File Translator Prototype, 1975 [1]; and the ASAP-TO-REL System (University of Pennsylvania), 1975 [2,17]. Although the seven systems studied differ in purpose, basic approach and architecture, they all contained the same functional components. These components have been grouped into three sections: Definition, Logic, and Execution (see figure 1). The Definition section is composed of the definition languages the conversion system uses. Since most systems use the hierarchical data model, the Data Definition Language (DDL) used to describe the source and target files looks much like a COBOL Data Definition. The required steps of restructuring the source file to produce the target file are usually contained in the "conversion" language. Any special source translation and value conversions may appear in the DDL (seen in the Ramirez System) or in the conversion language (seen in EXPRESS).

A transition function between the Definition section and the Logic section is the Language Processor. The DDL and Conversion language statements must be parsed and checked for syntax. Some systems (CODS), have elegant semantic analysers which guard against ambiguous conversion statements and redundant or impossible constructs. All

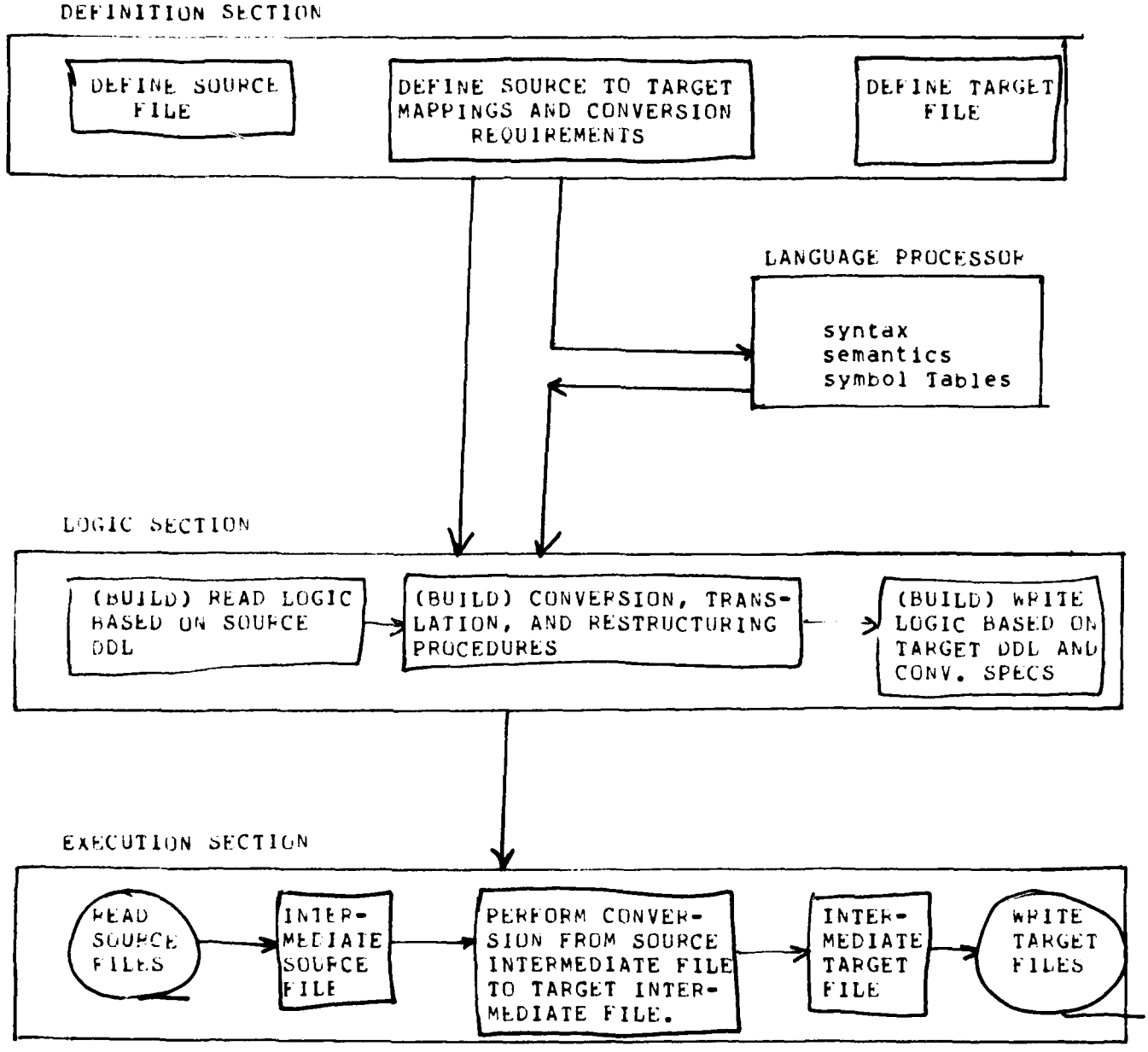


Figure 1. Basic Components of a Generalized Data Conversion System.

systems must combine the information gained from the definition and conversion statements and build a series of symbol tables. These symbol tables may be used in both the Logic and Execution sections.

The Logic section performs three functions: what format to read the source file, what is required to restructure, translate and convert each source record, and what format to write the target file. The read and write logic takes information from the symbol tables and file descriptions to determine the structure of the record and the location of specific fields. An example of this is the logic required to read a record containing a repeating group. Some type of WHILE LOOP would have to be executed (or generated, for non-interpretive systems) until a given delimiter had been received.

The Conversion logic is more difficult. If the conversion language is procedural (a small set of primitive conversion functions), the conversion logic is usually a set of generalized procedures corresponding to the conversion functions. This is the case for EXPRESS and CODS. Non-procedural conversion languages rely on deriving the conversion required by comparing the source and target DDLs. Any complex restructuring is accomplished by user written procedures (seen in Ramirez's System). For this approach, the conversion logic is reduced to simple mapping and proper procedure binding.

The Execution section contains the functions performed during the actual conversion. Before describing this, the interpretive vs generative approach must be explained. Generative systems generate programs which, when compiled, will perform the conversion desired.

Interpretive systems determine how to convert each record and perform the conversion all in the same step. Thus, interpretive systems have their Logic and Execution sections combined. A generative system would have a compiler between the Logic and Execute sections.

The Execute section performs the actual conversion. This includes reading the file, writing it in an intermediate source format, converting the file to an intermediate target format, and finally, writing the target file. Let us now look at the details of several implementations and see how they map to the common architecture just presented. The specific details studied will be the purpose of each system, the number of files it can handle, the data model it uses, and whether it generates code or is interpretive. These attributes, for the seven systems studied, are summarized in figure 2.

1.5 Comparison Of Implementations

The number and type of files a conversion system will support is greatly influenced by its purpose. For example, the purpose of CODS is to convert a source database to a target database, using the source and target database management systems to do the storage and physical level conversion. The Ramirez System converts a source sequential file to a target sequential file. His system can not use a database file, nor can CODS convert a file without the DMS. The ASAP-TO-REL System converts flat files produced by ASAP (a sequential file management system) to relational database load strings in REL (Relational English) format. Thus, its purpose is to allow a subset of a very large sequential file

SYSTEM NAME	PURPOSE	FILES	DATA MODEL	APPROACH
Rameriz,J (U. of Penn)	file to file	1 input 1 output	Hierarchical	Generate PL/I program
CODS (SDC)	DMS to DMS	1 DMS to 1 DMS	Hierarchical	Interpretive
EXPRESS (IBM)	file to file and DMS to DMS	multiple inputs and outputs	Hierarchical	Generate PL/I procedures
Data Translation proj. (U. of Micn.)	file to file	Multiple inputs and outputs	Relational	Interpretive
CODASYL (SDDTGG)	file to file	1 input 1 output	CODASYL	Interpretive
Honeywell	file to file	1 input 1 output	Hierarchical	Generate Assembly lang.
ASAP-TO-REL (U. of Penn.)	ASAP file to REL load string	1 input 1 output	Relational	Interpretive

Figure 2. Comparison of Studied Implementations.

to be converted into a relational database and queried against by the system REL [2]. Although the purposes and files of these data conversion systems vary greatly, they functionally are the same. Each system reduces the source files to one consolidated intermediate file. The format of this file is known. The target file's format is known, and a single target file is produced. IBM's EXPRESS further illustrates this process. EXPRESS can convert multiple input files, database supported or not, into multiple output files. It utilizes a "Reader" step to read all source files into a single intermediate file. The conversion step converts the source intermediate file into a single intermediate target file. The "writer" step then writes out this intermediate target file into the size and format the user wants. Thus, although the purpose, number and type of files differ among systems, they functionally execute the same.

The data model used by a system will affect its Data Description Language (DDL) and restructuring language more than its functional architecture. As previously mentioned, most systems use a hierarchical model. The rationale is that this model is familiar to the user (COBOL programmer), the restructuring dynamics are well known [13], and the major commercial database management systems support the model. Michigan University's Data Translation Project, however, uses a relational data model. The relational model allows for a normal form of data for its intermediate source and data file. This allows for more general and efficient conversion [6]. If, for instance, a relational source file needs converting to a hierarchical target file, the conversion would require a total reading of the source file before the conversion began. However, if the majority of the time the system is

converting hierarchical to hierarchical files, it probably would not be efficient to use the Michigan approach. In summary, the hierarchical model is the most popular model supported. If non-hierarchical files need converting, either a complete read step is required (as done by the Michigan System and EXPRESS) or software support outside the conversion system (as in CODS) is required.

The final feature to discuss is whether the conversion system is interpretive, or whether it generates code to be compiled. Although the approaches are clearly different, the logic required in the systems are the same. With the interpretive approach, the system programs are generalized, whereas for the generator approach, the method of constructing programs is generalized. In both cases the logic, for instance, to read all of the occurrences of a particular record vs generating the code to do the same function requires the same amount of knowledge about the record and its structure. Thus, the read step for an interpretive system reads the source record, while the generative system produces the code to read the source record. Functionally, and logically, they are the same. There are, however, some run time differences between the two approaches. Interpretive systems must execute logic using the DDL to determine how to read, convert, and write each record. Generative systems perform the reads, conversion, and writes directly since a specific program has been generated and compiled to do such. Better run time efficiency can be expected from the generative system due to the direct execution. There is, however, the system overhead of creating and compiling the generated program. Literature on actual performance comparisons is not known.

1.6 Common Architecture Details

It has been shown that the common architecture is a valid representation of the required functions and components of a data conversion system. This section will examine each functional component in more detail. Examples from the studied implementations will be used to illustrate functional component specifics.

1.6.1 The Definition Section Functions

A data description language (DDL) must be capable of describing the structure of the source and target files, (a hierarchical model will be assumed for this discussion). Shoshani [18] describes three levels of data structure description: logical, storage, and physical. The logical description itemizes the entities of the record, the relations among them, and the size and type of the fields. The storage level describes such things as file indexing organizations, access paths, and fixed or varible length records. Finally, physical level descriptions indicate how and where data is to be read and written, such as device type, blocksize and label information. If the conversion system converts the storage and physical level as well as the logical level, the DDL must have the capability of describing all three levels. This is the type of DDL the Ramirez and Michigan systems use. CODS, on the other hand, uses the source and target database management system facilities to perform the storage and physical conversion. The CODS DDL is therefore much smaller and simpler. If the source and target storage and physical levels are fixed (but not necessarily the same), the DDL,

again, would not have to describe all three levels. This is the case with the ASAP-TO-REL system where the input is always an ASAP file and the output is always Relational English Language strings.

There are two basic approaches for describing conversion specifications -- procedural and non-procedural. The non-procedural approach requires the user to describe the source file, the desired target file and the translation rules. Michigan's Translation Definition Language (TDL) is an example of the non-procedural approach. The procedural approach requires the user to specify, in terms of the conversion language primitives, the specific steps required to enact the conversion and the order in which to execute them. Examples of procedural conversion languages are EXPRESS' "CONVERT", and CODS' "CDTL". Proponents of the non-procedural approach believe it is less restrictive and easier to use [15]. The procedural languages proponents believe it is more powerful, efficient and direct [18].

The CODS Common Data Translation Language (CDTL) is representative of the procedural conversion languages studied. It consists of eleven primitive operations (EXPRESS' "CONVERT" has nine primitives). The primitives describe the basic data transformations required to restructure hierarchical data model structures, plus varied validation and conversion capabilities. The data transformation operations are of three types: 1.) moving values across on the same level, 2.) moving data values down and repeating them in each of its members, and 3.) performing an operation on a set of lower level values and moving this new single value up, or moving a specific occurrence of a lower level value up. Details on the meanings of these

transformations are in Appendix A -- User's Manual.

The final function in the Description section is the parsing, syntax checking and symbol table building. The literature gives little detail on these implementations. It is assumed that basic compiler principles are used.

1.6.2 The Logic Section Functions

When discussing the next two sections the reader is reminded that the interpretive and generative systems will differ slightly. The interpretive system will **execute** the code corresponding to the logic it just performed. The generative system will **output** high level code corresponding to the logic it just performed.

The read function is usually implemented by traversing the data descriptions and previously built symbol tables. As each field is parsed, a position in the input buffer is filled. For systems requiring storage conversion, the read function must have a subroutine corresponding to each possible access method. For physical level conversion most systems take advantage of the operating system they are executed on by merely setting appropriate file attributes. This may be done dynamically for interpretive systems, or in the generated Job Control Language (JCL) for generative systems.

The conversion logic is implemented differently based on the procedural/non-procedural characteristic of its conversion language, as

previously discussed. CODS uses the CDTL statements and the CDDL symbol tables to build a conversion table. Each table entry consists of the primitive's ID number and the relative address of the source and target fields. During execution (CODS is interpretive) each conversion table entry is executed by a CASE statement using the primitive number as the key. The Ramirez System uses a non-procedural conversion language (DML). It is a generative (non-interpretive) system. Its implementation requires the user to specify the maximum number of occurrences any repeating group may have. The strategy is to build the source and target record buffers large enough to hold the largest possible source/target record. During execution the read function expands the source record into a large fixed format record. The conversion function will then execute the data transformation operations from the source input buffer to the target output buffer. Input validations or special conversions must be written by the user in PL/I procedures and submitted as part of the conversion statements.

Some systems (ASAP-TO-REL) use the operating system to perform "value" conversions, such as Hollerith to EBCDIC code conversion. Other systems (EXPRESS and the Michigan systems) perform the conversions themselves. CODS has a separate language, Common Format Definition Language (CFDL), and a separate functional component which performs the "value" conversions. Most systems support table look-up value translations, but, obviously, the user is required to fill the table (for interpretive systems) or write the translation subroutine (for generative systems).

1.6.3 The Execution Section Functions

The first function during execution is to read the source file. If the system can handle several files, most implementations read all of the files and combine them into a single intermediate source file. This is done by the Michigan, EXPRESS, and CODS systems. It is not necessary to read the entire source file before converting. The Ramirez and ASAP-TO-REL systems read a source record, convert it, and write the new target record out one at a time. These systems usually can handle only one input source file and are guaranteed it will be in a specific storage format, (i.e. sequential file with variable length records).

The implementation of the conversion step is usually motivated by efficiency factors. The number of I/O operations must be kept to a minimum as well as memory to memory data moves. EXPRESS implements a "pipelining" technique to increase its efficiency. The Michigan system has been making efforts towards bypassing the conversion step for records which do not require conversion, (aggregate schema facility). Most of the "minor" implementations have not introduced any significant efficiency features and execute the conversion step quite straight forwardly.

A final comment should be made on execution flexibility. Flexibility in this sense means: 1.) the ability to handle the hard to describe, very unusual conversion requirement, and 2.) the ability to execute the conversion in incremental steps. The generative systems usually allow more flexibility in regard to handling the unusual conversion case. This is because the generated code can usually be

accessed and modified prior to its execution. EXPRESS produces separate read, conversion and write PL/I procedures for each job. During execution the EXPRESS system calls these procedures based on the conversion phase it is in and the data being operated on. The Ramirez system produces a complete, self-contained PL/I program. The execution step is conducted completely free of any conversion system support. The EXPRESS system could be difficult to alter, particularly if the desired change was in the control portion of the program. The Ramirez system, however, would be much easier to modify since it is a complete, self-contained program. The advantage of the Ramirez self-contained program is also a disadvantage in terms of incremental step execution. The only way to break-up the Ramirez conversion is to stop its execution and rely on some "restart" mechanism to start it at a later time. Other systems, such as EXPRESS, Michigan and CODS, allow separate reading, converting and writing of the files to be converted. With this flexibility, the conversion can run even though a large block of computer time is not available.

1.7 Summary of Common Architecture

Based on the examination of seven data conversion implementations, the common functions of a data conversion system have been identified. These include a DDL to describe the source and target files, a conversion language to describe the source to target field mappings, and read, conversion and write modules. Differences in DDLs were found to be based on the data model the language used, and how many data structure levels it converted (logical, storage, and physical).

Conversion language differences arose depending on whether the language approach was procedural or non-procedural. Read and write module differences were based on how many source/target files the system could handle. Conversion module implementations differed due to the procedural/non-procedural language approach, and efficiency factors introduced. Finally, whether the system took an interpretive or generative approach appeared to affect its output (converted records or a conversion program) more than its architecture. Based on this common architecture, the Source to S2K Conversion System was designed.

CHAPTER 2

DESIGN AND IMPLEMENTATION

This chapter will discuss the design of the Source to S2K Conversion System and document its software implementation. The design discussion will follow the organization of the common conversion system architecture, as presented in Chapter 1. The implementation discussion will present the general software organization, major data structures, and itemize the main procedures, their functions, inputs and outputs.

2.1 System Description

The Source to S2K Conversion System design lent itself well to a "top-down" development. The system's purpose, to convert source files to S2K databases, was well defined. Because the S2K system provides a conversion facility through execution of a Program Language Interface (PLI) program, generating a new program for each conversion job appeared

to be the best approach. Using the hierarchical data model also was a natural choice since the target file would always be an S2K database. In order to simplify the implementation, the number of source files was restricted to one, as was the number of different target databases. Generating a PLI FORTRAN program was decided over generating a PLI COBOL program due to local support. Thus, starting with the purpose of the system and some basic decisions, the design of the system developed. It would take as input a description of the source file, S2K database, and conversion mappings, and produce a PLI FORTRAN program which, when executed, would perform the actual conversion. Figure 3 shows this design. Design details of the system's Definition, Logic, and Execution sections are now presented.

2.2 Definition Section Design

Languages had to be designed which allowed the user to input the necessary information needed to generate the FORTRAN program. These languages included one to describe the source file, one to describe the mappings between the source and target files, and a third for miscellaneous system input. A special target description language was not necessary as the required S2K database description input could be used.

2.2.1 Source File Definition Language Design

As discussed in Chapter 1, there are three levels of data structure that must be described: logical, storage, and physical. Since the Source to S2K system has a limited scope, extensive

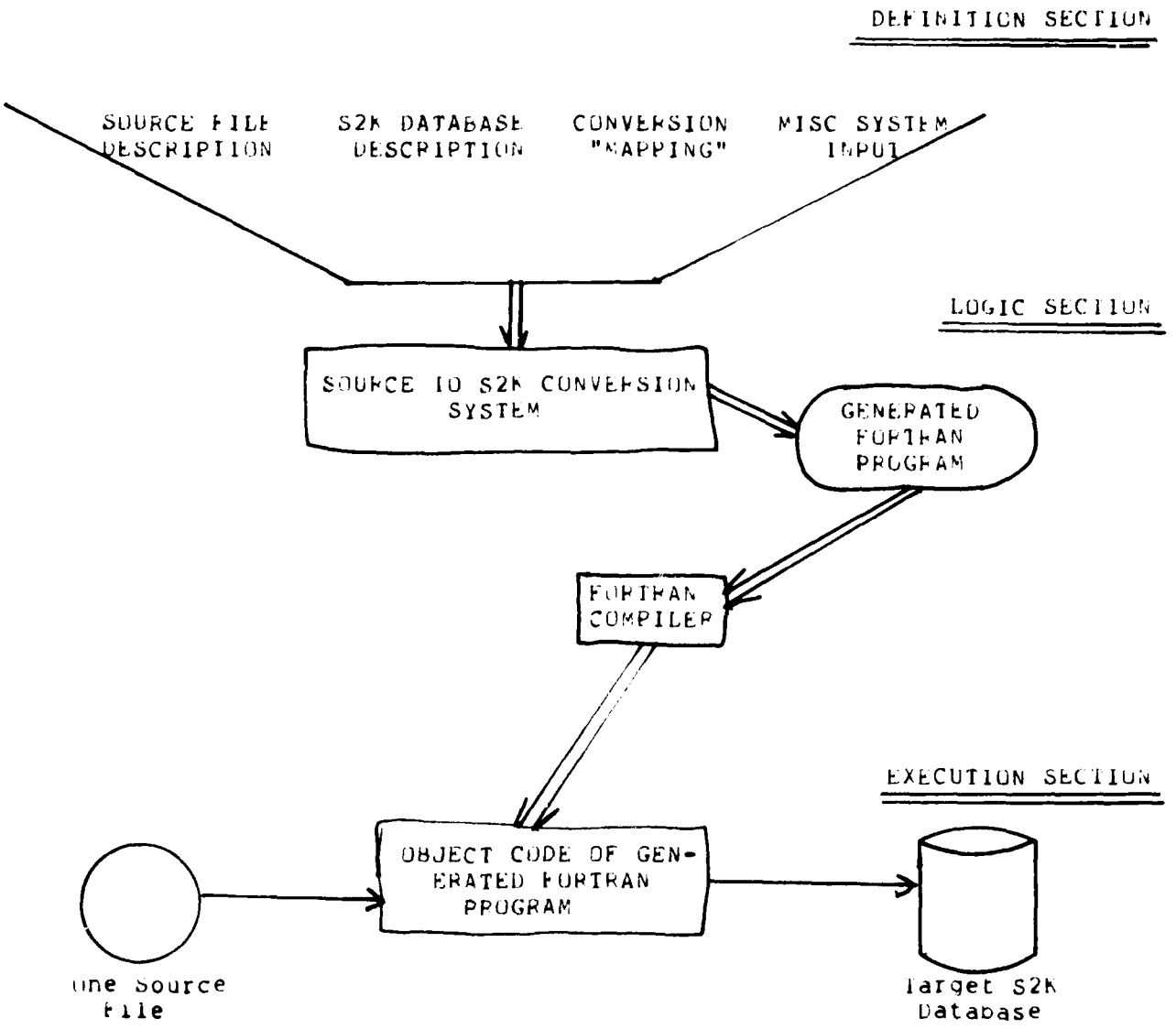


Figure 3. Data flow of a Source to S2K Conversion System job.

description capability for all levels was not necessary. Specifically, the UT-2D operating system has no direct means of describing file storage characteristics, other than stating the file is "local" or "foreign". Secondly, since the target file is an S2K database, commonly stored on disk, the physical conversion requirements will be small. Therefore the storage and physical level descriptions can be simple, consisting of keywords followed by user input. For example,

```
FILE = INPUT/1234/9876.
DEVICE = DISK.
```

indicates the input source file name is "INPUT" and it resides on permanent disk library number "1234" (password "9876").

The source file logical description is also simple, due to two restrictions. First, the file must be describable in a hierarchical manner. Secondly, since the UT-2D storage structure capabilities are limited, all source records must be fixed length. This implies all repeating groups have a defined maximum number of times they may repeat. The logical description is thus reduced to field names, field specifications, and the maximum number of times a group may repeat. The field names consist of the letter "S" followed by an integer, starting with one, increasing by one for each new field. A comment field is provided to make the field name more meaningful (i.e. "S3 A20. Company Name."). Since FORTRAN FORMAT statements will be generated from the source input, the field specifications use the same notation as the format statements. An example of a logical structure description is given in figure 4. More examples may be found in the User's Manual, Section 2.C.

```
S1   A18.              DAD Name.
REPEAT 5 BEGIN.        Start CHILDREN Repeating Group (Max=5).
   S2   A10.           CHILD Name.
   S3   I2.            AGE.
   REPEAT 3 BEGIN.     Start PET Repeating Group (Max=3).
      S4   A12.        PET Kind.
      S5   A8.         PET Name.
   END.                End of PET Repeating Group.
END.                   End of CHILDREN Repeating Group.
```

Figure 4. Source Description Language Example.

2.2.2 Conversion Language Design

A procedural language approach was taken for the conversion language. Based on the systems studied, it appeared to be the least ambiguous for the user and easiest to implement. Seven primitives were designed, each corresponding to either a data transformation operation, a conversion or validation operation, or the special STORE operation. CODS's conversion language, [18], strongly influenced this design. CODS is a DMS to DMS conversion system, requiring the source and target DMSs to handle all physical and storage structure conversions. Its conversion language primitives are concerned only with the logical level conversion, and focus on the three basic hierarchical model data transformations needed to map source to target data structures. These transformations are discussed in Chapter 1, Section 1.6.1, and the User's Manual, Section 2.E.

Along with the data transformation primitives, conversion language primitives for validation and conversion were also designed.

The conversion primitive allows the user to write FORTRAN code which will be included in the generated program. This code should perform a unique conversion on one, or several, source fields to produce a single target value. The validation primitive allows the user to input FORTRAN code for the purpose of validating a particular input source field. The user also specifies an option that execution should take (reject validated field or reject data set occurrence) should the validation fail. The validation primitive is a feature not seen in any of the implementations studied. Ramirez's system allows users to input PL/I procedures in order to perform validation checks, but provides no capability of altering the control of the execution should the validation fail. Since most conversion efforts desire some editing of the source data, the validation primitive is an important, practical feature.

The final conversion primitive is the special STORE operation. The user is expected to input a data transformation primitive for each target field in the order the fields are defined. After the transformation for the last field in a particular group is input, the primitive STORE must be input. This specifies to the system that all target fields for this group have been "filled" and the new data set should be written. A data transformation primitive for the first target field of the next group should then be input. The last input for this group should, again, be followed by a STORE primitive. This process should be continued until the end of the defined target database is reached. Further details of the conversion language and examples are contained in the User's Manual, Section 2.E. Figure 5 contains a

summary of the conversion language primitives.

2.2.3 Miscellaneous System Input

Information on the S2K database file name and several system options were needed to complete the generated FORTRAN program and generated UT-2D command files. A keyword followed by user input format was designed to give the user this input capability. For example

RUN = S

is an option card specifying the run is for syntax only. The proper input to specify the run is a full generation run is

RUN = F.

All of the key words and user input options are discussed in Section 2.4 of the User's Manual.

2.3 Logic Section Design

For generative conversion systems, the logic section is where the conversion program is generated. Using the user's input, read, conversion, and write modules must be generated. In addition other required code must be generated, such as database schema and local declarations, opening and closing of the database, and error detection procedures. This required code is fairly static, requiring little change from job to job. The read, conversion and write modules are far more dynamic and require more complex algorithms. Their design will be discussed here.

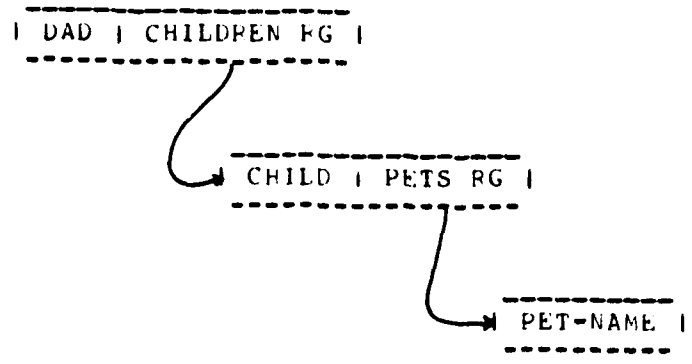
PRIMITIVE NAME	TYPE	FUNCTION
DIRECT	TRANSFORMATION	The transformation used to move source to target fields that are in correspondence.
REPEAT	TRANSFORMATION	The transformation used to move a source field in an ancestor data set to a target field.
LEVELUP	TRANSFORMATION	The transformation used to move a specific occurrence of a source field in a subordinate set to a target field.
UPOP	TRANSFORMATION	The transformation used to apply an operation against all occurrences of a source field in a subordinate set. The results of the operation are moved to the target field.
CONVERSION	USER WRITTEN	Signals the input of a user written FORTRAN module. The module will perform a conversion on one or several source fields.
VALIDATE	USER WRITTEN	Signals the input of a user written FORTRAN module and instructions for execution should the validation module return a "false" value.
STORE	SPECIAL	Signals the end of the conversion primitives for the target data set being built.

Figure 5. Summary of the 7 Conversion Language Primitives.

2.3.1 The Read Module Design

The purpose of the read module is to read a complete source record and separate each field so it can be individually moved to a target field. These two operations could be accomplished by a FORTRAN formatted read, but this statement restricts the source input to 150 characters. Since this restriction is unacceptable, an unedited FORTRAN read statement is used to read the source record and several DECODE statements are used to separate the fields. The number of words read by the unedited read is calculated from the source input description. The decode statements will separate the fields from the input buffer and put them in a temporary array, one field for each array word. Since the decode statement also has a 150 character limit, several statements may be necessary. After execution of the unedited read and decode statements, each source field resides in a separate array word and can be directly addressed. During the parsing of the input source file description, a symbol table is filled which maps the source field names and their corresponding temporary array addresses. For example, consider a source file consisting of the field DAD (18 characters), a repeating group CHILDREN (max=2, each 10 characters), and a repeating group PETS (max=3, each 8 characters) within the group CHILDREN. The source name to temporary array location mappings are shown in figure 6.

SOURCE FILE DESCRIPTION



SOURCE NAME TO ARRAY LOCATION MAPPING

Field Name	Array Address
DAD	1
CHILD #1	3
PET-NAME #1	4
#2	5
#3	6
CHILD #2	7
PET-NAME #1	8
#2	9
#3	10

Figure 6. Example of source file to temporary array mapping.

CHILD 1 starts in location 3 instead of 2 because DAD is greater than 10 characters. Array word 1 and 2 are used to store the DAD field. The symbol table does not itemize each field occurrence and its corresponding temporary array address, as shown in figure 6. Rather, the address of the first occurrence of each field, the number of words between the first and second occurrences, and the maximum number of occurrences is stored. This information is passed down from each level to its subordinate levels. The symbol table for the previous example

would be:

		level 1		level 2	
Field Name	1st Occ	Size Between	Max Occ.	Size Between	Max Occ.
DAD	1	0	0	0	0
CHILD	3	4	2	0	0
PET-NAME	4	4	2	1	3

Addresses for CHILD are the original (3) and the original plus the size between occurrences (3+4=7). Addresses for PET-NAME are the original (4) plus the size between occurrences for level 2 (4+1=5, 5+1=6) and the same iteration for the second occurrence of level 1 (4+4=8, 8+1=9, 9+1=10). This algorithm is used when generating the conversion assignment statements.

In summary, the *read module logic* consists of generating an unedited read statement to move an entire source record into an input buffer. Decode statements are then generated which convert each field to its proper internal representation, and moves the value to a temporary array word. The previously build symbol table allows retrieving the proper temporary array word for any occurrence of any source field.

2.3.2 The Conversion Module Design

The conversion module is responsible for generating the FORTRAN code for the data transformations and validation/conversion procedures. Since the user is responsible for the validation and conversion code, the only action the system takes is to replace the source field references with their proper temporary array locations. This is done

using the symbol table mappings built during the source definition parsing, and a set of indexes, one index corresponding to each source data group. The value of the indexes represents the "current" occurrence of its corresponding group. By computing

INDEX = <orig. pos.> + (<curr index1>*<group1 size>) + ...

for all groups the field in question is subordinate to, the correct temporary array subscript is found. This computation statement is generated before each source field reference. Then the field name is replaced with the temporary array name, subscripted by the variable INDEX (i.e. TEMP(INDEX)).

The data transformation algorithms must also generate similar statements for all source field references. Before a source data value is moved to a target field, INDEX must be computed. Then the value of the temporary array, subscripted by INDEX, is moved to the target field. The other task the data transformation algorithms must accomplish is generating proper looping statements. These statements are needed so that the data transformations are executed for all source field occurrences. The DIRECT data transformation (moving values on the same level) requires a loop for the group the source field is in, plus a loop for each group the source field is subordinate to. Consider the DAD, CHILDREN, PETS data structure in figure 6 as a source file, and the target is a "PETS" database, one pet per record. In order to address all of the pets contained in a single input record, the CHILDREN group must be looped through as well as the PETS group. Thus, this example would require generation of two FORTRAN DO loops.

The REPEAT data transformation (moving upper level values down) requires no additional loop statement generation. The current occurrence of the parent group will contain the correct source field value. Using the DAD, CHILDREN, PETS example again, consider moving the CHILD (name) into the target "pet" record. The proper occurrence of the CHILDREN group must be used. Since the previous DIRECT statement generated a loop for the CHILDREN group, the proper index is guaranteed. The argument for this is the following. If a source value is being moved "down" to a field in a target group, the target group must have a corresponding source group. At least one field in this corresponding source group must be moved to the target group using the DIRECT transformation. Since DIRECT generates loops for all groups above it, the parent group the REPEAT refers to will be properly incremented.

The UPOP (Up Operation) data transformation is designed to perform an operation on all field values contained in a subordinate group. Here again loops must be generated for the group itself plus all groups superior to it up to the group level which called the transformation (the DIRECT group level). Consider the previous example, but this time the target database is a "DADs" database rather than a "PETs" database. In this case the source level 0 fields would be moved to the target level 0 fields using the DIRECT transformation. Consider a target field defined "NUM-PETS-OWNED", with the desire to store in each DAD's target record the number of pets he owns. A loop for the CHILDREN group as well as the PETS group must be generated in order to count all of the pets belonging to each source DAD record. It is not sufficient to generate only a single loop for the PETS group.

In summary, the conversion logic algorithms must accomplish two tasks. The first is to generate code which will compute the correct temporary array subscript for each source field occurrence. The second is to generate looping statements so that a data transformation is executed for all source field occurrences.

2.3.3 The Write Module Design

All target database "writes" are accomplished using the S2K PLI statement INSERT <schema name>. The semantics of the INSERT statement are to attach the <schema name> data set to the database, positioning it according to the current values of each S2K set occurrence pointer. Thus, if the level 0 occurrence pointer equaled 3, an INSERT on a level 1 data set would become a subordinate set of the third occurrence of the level 0 data set. The entire write logic is, therefore, based on insuring the order of INSERT commands is correct. Using the DAD, CHILDREN, PETS data structure as a target database, an INSERT for the first DAD is followed by an INSERT for the first CHILD which is followed by as many INSERTS as there are PETS belonging to the first CHILD. Then the next CHILD INSERT is issued, followed again by as many INSERTS as there are PETS belonging to the second child. This order is continued until all CHILDREN for the first DAD have been inserted. Then the order repeats, starting with an INSERT for the second DAD, etc.

2.4 Execution Section Design

The Source to S2K Conversion System generates a complete, self-contained FORTRAN program which will perform the entire conversion job. This is in contrast to generating unique conversion procedures and then calling them when needed, as done by the EXPRESS system. In order to support the execution phase, the Source to S2K System generates two files containing UT-2D control commands. One file is needed to support generation of the FORTRAN program and the second to control its execution. Because several users may be using the system, unique names for the generated programs and the command files must be assigned. The rules for these names are contained in Section 3.2 of the User's Manual. In order to generate these unique files, as well as simplify the user's input, a single, fixed command file was designed. This file, named "GENRATE", is called by the user. It will take the user's description input and execute the Source to S2K Conversion System (see figure 7). Here the FORTRAN program and the two command files are generated. Next, file GENRATE calls the first command file just generated. The commands in this file will sort the generated FORTRAN program, (see section 2.5.2), compile it, change the program's name to a unique name for that user, and save it. When the user is ready to execute the conversion job, the second generated command file is called. This file will compile the FORTRAN program, ready the source input, target database and S2K software, execute the program and save all files. Details and examples of executing the system are contained in Section 3 of the User's Manual. The command statements for the file GENRATE, and an example of the two generated files, is contained in Appendix B.

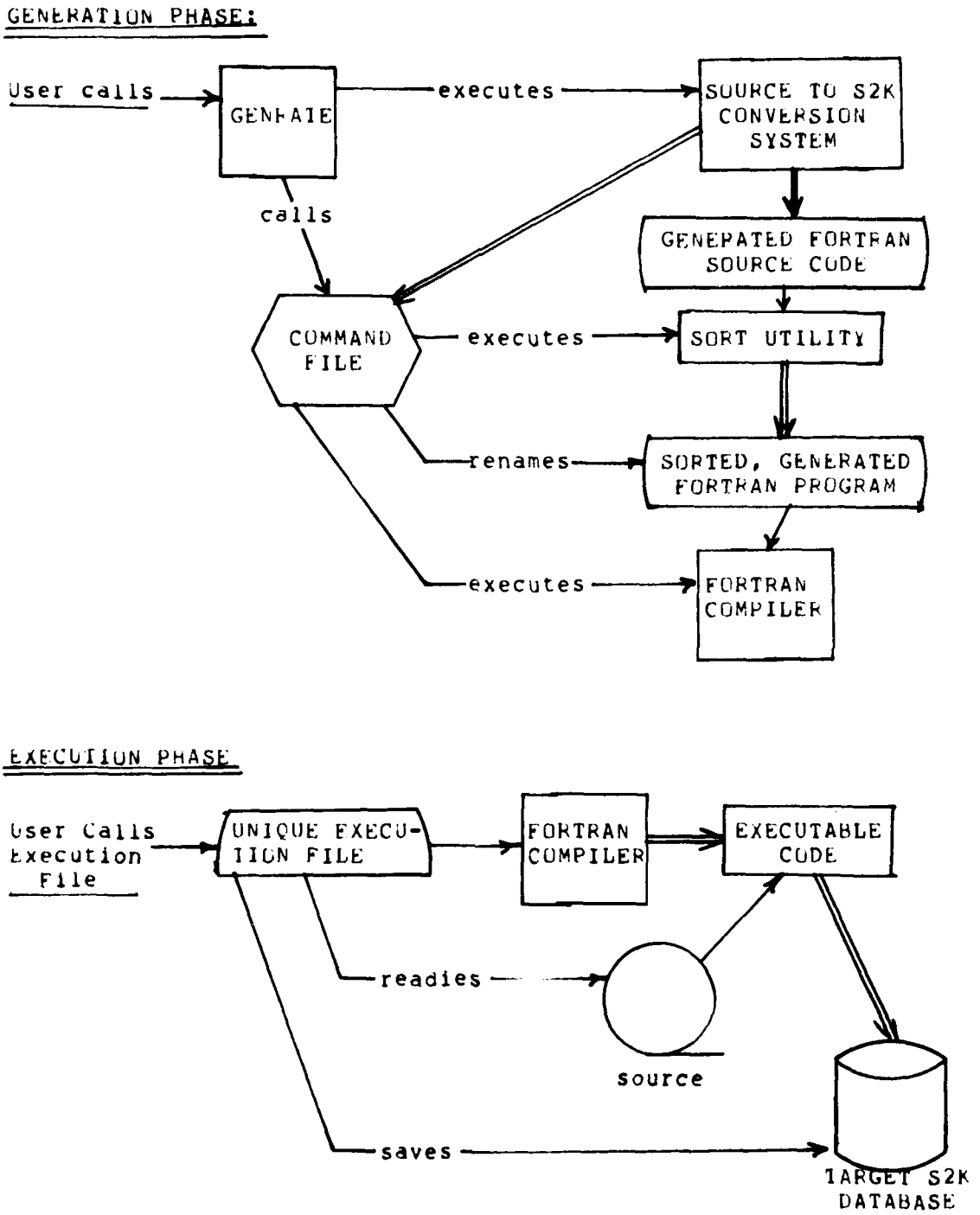


Figure 7. Files and tasks involved in each Source to S2K Conversion Job.

2.5 Implementation Details

Details of the system's design and major algorithms have already been presented. This section will present details on the software implementation of these algorithms. Specifically, the implementation philosophy, basic program structure, major data structures, and an itemization of each procedure, its function and inputs/outputs are presented.

2.5.1 Implementation Philosophy

The programming philosophy used for this implementation is a result of the author's 12 years programming experience and recent graduate work in Programming Methodology. While it is beyond the scope of this report to document the entire philosophy, it may be of interest to highlight certain programming concepts and techniques used to implement the Source to S2K Conversion System. The programming concepts discussed can be thought of as guidelines to "good" program construction. The program techniques are specific rules and procedures which complement the concepts and help realize the program construction.

2.5.1.1 Programming Concepts

Reliable programming is not an easy task. Most systems are very large and very complex. Due to this size, few programs can be completely tested where all possible input and output states are examined. In light of these facts, it is believed reliable programs must be constructed in a disciplined and systematic manner. This is the first and most important programming concept. The second concept is to

construct programs using a hierarchy of abstractions. This means to suppress the details of a function to the lowest level possible. The purpose of this is to improve both the clarity and understandability of the program. Dijkstra states, "The purpose of abstraction is not to be vague but to create a new semantic level in which one can be absolutely precise" [4]. How to recognize when a new semantic level is desirable, as well as the total organization of the program, should be guided by specific reasoning rather than intuition. The third concept, therefore, is to use Parnas' work [14] in program module decomposition as a criteria for determining the modules of a program. Briefly, this criteria includes:

1. Emphasize the interface between the modules rather then the traditional functional modularization.
2. From a given set of requirements, select the set of assumptions that are likely to change. Design modules around these assumptions and "hide" them in the module. Then select the assumptions that are unlikely to change and design the module interfaces around them.

The final concept is that of developing the program using a "stepwise refinement" approach, as introduced by Wirth [20]. Stepwise refinement means that program construction should be viewed as a sequence of refinement steps. In each step a task is broken up into several subtasks. As the descriptions of each subtask are refined, so should the data structures used to support the tasks. Thus, the program and supporting data structures are developed in parallel. The important aspect of this concept is to recognize the possiblity of improving an algorithm when the data structures are refined. Rather than design the data structures separately, they should be designed using the same

hierarchical process as that used to design the algorithm's logic.

2.5.1.2 Programming Techniques

The four concepts discussed, constructing the program in a disciplined, systematic manner, using a hierarchical level of abstractions, decompose the program modules based on their interfaces, and develop the program and data structures using a stepwise refinement approach, must have specific programming techniques to support them. The first technique used was to ensure all requirement specifications were completed, reviewed, and accurate before any system design work commenced. The emphasis here was to study the elements in the system which were likely to change and those which were likely to remain stable. From this study a better decomposition of the system could be made during the design phase.

Structured flowcharts were used to design the entire system before coding began. Figure 8 is an example of a structured flowchart for an algorithm to merge two sorted arrays into a single array. These charts encourage a structured organization and the use of levels of abstraction. The only programming constructs allowed are the assignment, if..then..else, and case statements, procedure calls, and "while" loops. The "while" looping invariant is clearly stated at the top of each loop. Using this limited number of primitives encourages simplier programming, fewer "tricks" and no GOTO statements.

The third technique used was to select a programming language suited to the philosophy. PASCAL was selected because of its block

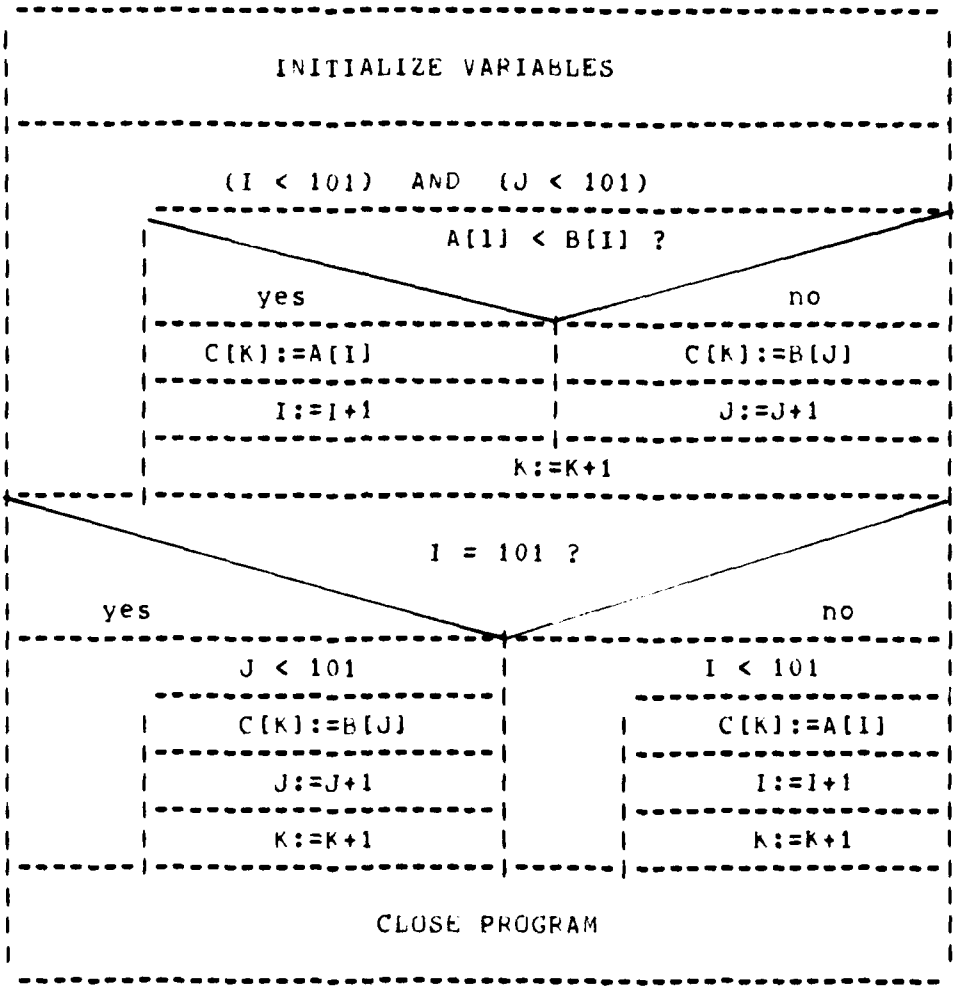


Figure 8. Structured flowchart for merging two sorted arrays (A and B) into a single array (C).

structure, its control mechanisms (FOR, WHILE, and REPEAT..UNTIL statements) and its capability to define heterogeneous data structures. PASCAL's weaknesses, string manipulation and input/output, caused some problems in parsing the user input. However, the capability to define elaborate symbol tables (see section 2.5.3) compensated for these weaknesses.

The final technique was to use the work in program verification (see Floyd [5], Hoare [8], and Yeh [21]) as a guide towards writing correct code. Individual procedures in the Source to S2K conversion system were not formally proven. However, each procedure was written with certain verification rules and steps in mind. These steps were:

1. Examine the algorithm/task to be programmed and find the loop.
2. If there is a loop, establish a loop invariant. This invariant is that condition (B) which remains true throughout the processing of the loop, and becomes false when the loop terminates.
3. Initialize all variables before entering the loop, insuring the invariant (B) remains true. If it does not, the initialization is in error or the invariant is not correct.
4. Ensure the code contained within the loop approaches the condition NOT (B). This step is taken to guarantee the loop will terminate.

The above steps, as well as the other techniques discussed, gives credence to the programming philosophy presented. These techniques were not just investigated, but faithfully used. As a result, the Source to S2K Conversion System implementation is well structured, can accommodate modification and is believed to be correct.

2.5.2 Basic Program Structure

Figure 9 is a structured flowchart of the Source to S2K Conversion System program. The PARSE module reads all of the user input, checks for syntax errors and builds the symbol tables. If an error is found in the user input, no FORTRAN program is generated. If no errors are found and the user "asked" for program generation, a series of generation modules are called, as shown in figure 9. Data needed to generate the FORTRAN statements are in the symbol tables and other variables which were filled by the PARSE module. A line of code is generated and then written to the program output file FORTSRC, (FORTRAN Source). Several situations arose where a line of code needed to be generated immediately, but its subsequent write put it out of order. Examples of this are generating a subroutine before all of the "main" code is complete, and generating the beginning and ending of a DO loop before generating the code contained in the loop. To solve this problem, each generated line of code is written with a leading tag. This tag represents the logical position of the generated line. When the program is completely generated, a sort utility is called to sort the program in proper tag sequence. This moves all generated lines of code to their proper location in the program.

All common parsing and generation functions were written outside of the main structure of the program and made global to the procedures needing them. These functions include searching for a particular character in an input string, putting a series of characters into an output line buffer, generation of leading tags, and all input and output. This allowed placing the details of many functions in a single

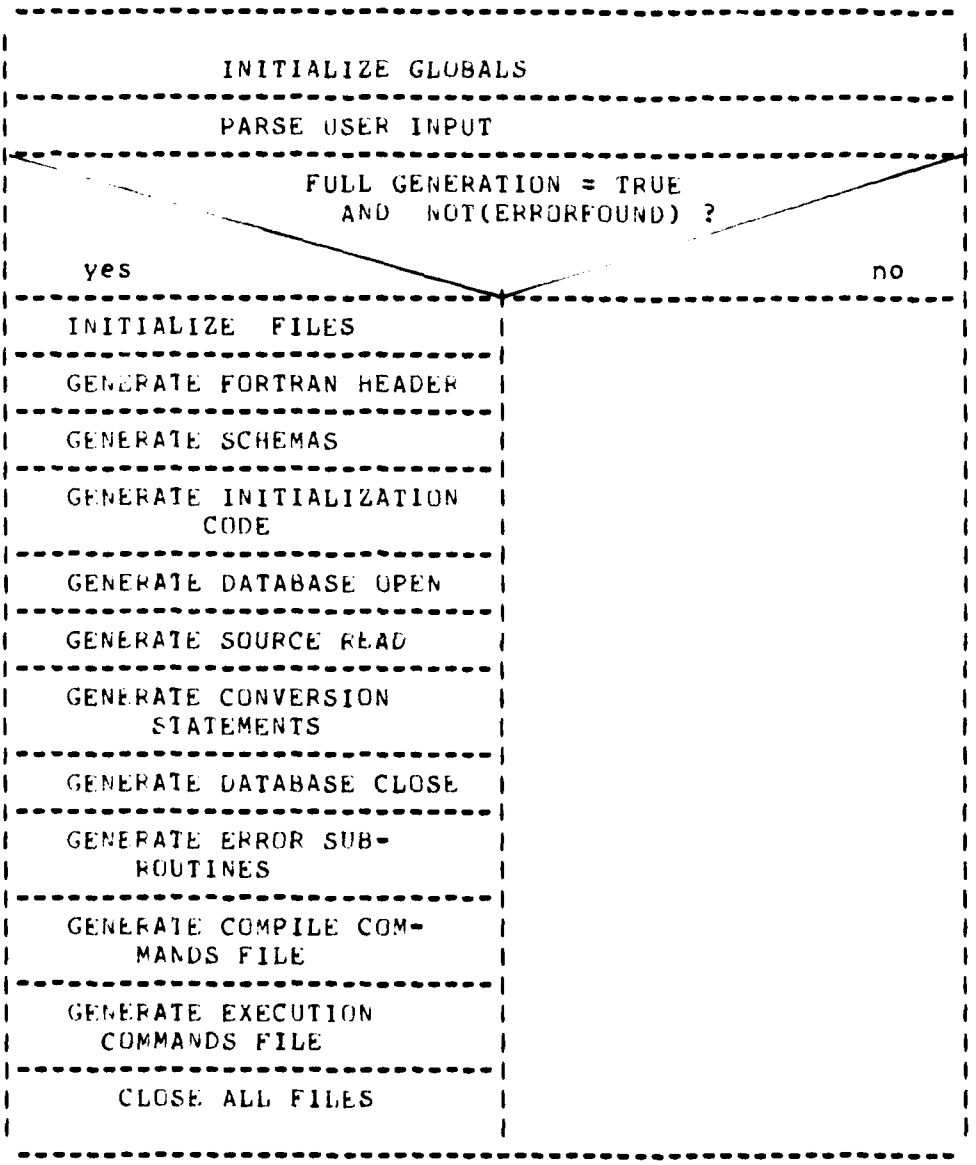


Figure 9. Basic program structure of the Source to S2K Conversion System.

location, hiding them from the main logic of the program. Other program details may be found in the Itemized Procedure Listing (section 2.5.4).

2.5.3 Data Structures Used in the Program

A number of global data structures were declared in order to facilitate communications between the PARSE and GENERATION routines. These global structures may be considered the symbol tables of the system. The parse routines "fill" the symbol tables with data received from the user. The generation routines then "read" this data, thereby generating a unique PL1 FORTRAN program for the user. This section will describe the layout and meaning of the important global data structures.

The Source to S2K Conversion program declared three primary data structures: the Source, the Target and the Conversion Symbol Tables. Several other single word arrays were globally declared, but their usage and description is evident upon examination of the program listing. Because PASCAL's declaration notion is exceptionably concise and readable, the actual PASCAL declarations for each symbol table will be presented, with explanation.

The Source Symbol Table Declaration:

```
TYPE
   SRCTBLTYPE = RECORD
                  GRPNUM          : 1..MAXGROUPS;
                  PARENTINDEX     : 1..MAXSRC;
                  ORIGPOSITION    : 1..MAXRECSIZE;
                  EDITSPEC        : CHAR;
                  DECIMALSPEC     : INTEGER;
                  FLDSIZE         : INTEGER;
                  REPEATARRAY     : ARRAY[ 1..MAXREPEATS ] OF REPTYPE;
                END;
```

```
   REPTYPE     = RECORD
                   REPNUM         : INTEGER;
                   REPINCREMENT   : INTEGER;
                 END;

VAR
   SOURCETABLE : ARRAY[ 1..MAXSRC ]  OF SRCTBLTYPE;
```

The meaning of the declared fields are:

SRCTBLTYPE - The name of the record type description for the Source Symbol Table.

GRPNUM - The group number of the group the field resides in.

PARENTINDEX - The Symbol Table index number of the field's group's parent.

ORIGPOSITION- The index number of the first occurrence of this field in the temporary array SRC.

EDITSPEC - The field's input edit specification.

DECIMALSPEC - The decimal specification for a type REAL.

FLDSIZE - The size of the field.

REPEATARRAY - Repeat array data for the level the field is at and all levels above it.

REPTYPE - The name of the type description for the repeat array.

REPNUM - The number of times this group repeats.

REPINC - Total size of the group.

SOURCETBL - The name of the Source Symbol Table with the layout as described in the type description SRCTBLTYPE.

The Larger Symbol Table Declaration:

```
TYPE
   TARTBLTYPE = RECORD
                  NAME          : ALFA;
```

```
                TARGETTYPE      : CHAR;
                SIZE            : INTEGER;
                REPEATED        : BOOLEAN;
              END;

VAR
   TARGETTBL : ARRAY[ 1..MAXTARGET ] OF TARTBLTYPE;
```

The meanings of the declared fields are:

TARTBLTYPE - The name of the type description of the Target Symbol Table.

NAME - The component name for this field, or the data set name if the entry is a repeating group header.

TARGETTYPE - The field's type (i.e. integer, real or character).

SIZE - The size of the field in 10 character words. (Type real and integer always are SIZE = 1 word).

REPEATED - A flag to indicate whether the field is contained in a repeating group.

TARGETTBL - The name of the Target Symbol Table as described by the type description TARTBLTYPE.

The Conversion Symbol Table Declaration:

```
TYPE
   CONVTBLTYPE= RECORD
                CONVTYPE        : CHAR;
                SRCNUM          : 0..MAXSRC;
                TRGNUM          : ALFA;
                TEMPTYPE        : CHAR;
                OPER            : 0..5;
                MISC            : ALFA;
              END;

VAR
   CONVTBL : ARRAY[ 1..MAXCONV ] OF CONVTBLTYPE;
```

The meaning of the declared fields are:

CONVTBLTYPE - The type description for the Conversion Symbol Table.

CONVTYPE - The type of conversion statement this entry is, (i.e., a DIRECT, REPEAT, LEVELUP, etc.).

SRCNUM - The Source component for the conversion transformation. If SRCNUM = 0 the source is in the temporary variable.

TRGNUM - The S2K component number for this conversion transformation. If CONVTYPE = STORE, this is the Repeating Group name to store.

TEMPTYPE - If SRCNUM = 0, this indicates which temporary variable holds the source value, (i.e. TEMPREAL, TEMPINT or TEMPCHAR).

OPER - If CONVTYPE = LEVELUP, OPER=0 if the term "LAST" is input. If CONVTYPE = UPOP, OPER indicates the operation to apply, (i.e. 1=MAX, 2=MIN, 3=AVG, 4=COUNT, 5=TOTAL).

MISC - If CONVTYPE = LEVELUP, MISC holds the specific occurrence numbers for each repeating group. If CONVTYPE = DIRECT, MISC holds the optional constant to be moved.

CONVTBL - The name of the Conversion Symbol Table as described by the type description CONVTBLTYPE.

2.5.4 Itemized list of Program's Procedures

As detailed in Section 2.5.2, the Source to S2K program is

divided into two main modules, (the PARSE and the GENERATION), with the global symbol tables providing the communication between the two. This section will discuss each module separately, presenting a "top-down" view of the procedures, and give some details of the most important ones.

Program's Top Level Documentation:

SRCTOS2K Program (Source to S2K)

```
(* Main Procedures *)
INITGLOBALS
PARSE
GENERATEALL

(* Utility Procedures Global to All *)
CONVTOCHR   (Convert to Character)
CONVTOINT   (Convert to Integer)
```

INITGLOBALS

FUNCTION -- All global data structures are initialized.

INPUTS/OUTPUTS -- No formal parameters. Global data structures needing initialization are altered.

PARSE

FUNCTION -- To read the user input, ensure correct syntax and semantics, (where possible) and fill the symbol tables.

INPUTS/OUTPUTS -- The user's input descriptions are the input. The output are filled symbol tables.

GENERATEALL

FUNCTION -- To generate the PLI FORTRAN program and control command files.

INPUTS/OUTPUTS -- The input are the filled symbol tables. The output are the text files containing the FORTRAN program and control commands.

CONVIOCHR

FUNCTION -- To convert the inputted integer to its character form.

INPUTS/OUTPUTS -- Formal parameter "INT" is the input integer to be converted to character form. The global word "token" is the output word where the character form of the integer is stored.

CONVIOINT

FUNCTION -- To convert the inputted character word to its integer value. CONVIOINT is declared a type integer function.

INPUTS/OUTPUTS -- The formal parameter is a 10 character word. The output is the function name itself, where the binary value of the inputted character form is stored.

The PARSE Procedure Documentation:

PARSE

```
(* Main Procedures *)
COMMANDMODE
SOURCEMODE
```

```
S2KMODE
CONVERSIONMODE

(* Utility Procedures Global to PARSE *)
READACARD
GETTOKEN
SKIPBLK
EQUALOK
CHKPERIOD
ERROR
```

COMMANDMODE (Command Mode)

FUNCTION -- The user's command language inputs are parsed and appropriate global data structures are filled with data.

INPUTS/OUTPUTS -- The input is the user's description input file. The output are globally declared single word arrays filled with data the generation module will use.

SOURCEMODE (Source Mode)

FUNCTION -- The user's source input description is read and the source symbol table is build.

INPUTS/OUTPUTS -- The input is the user's source file description. The output is the completed source symbol table.

S2KMODE (System 2000 Mode)

FUNCTION -- The user's S2K target file description input is parsed here. The target symbol table is also build.

INPUTS/OUTPUTS -- The input is the user's target file description. The output is the completed target symbol table.

CONVERSIONMODE (Conversion Mode)

FUNCTION -- The user's conversion statements are parsed and the conversion symbol table build.

INPUTS/OUTPUTS -- The input is the user's conversion statements. The output is the completed conversion symbol table.

READACARD (Read A Card)

FUNCTION -- To read a record from an input file and place the record in a temporary text file. This text file will act as an input record buffer, 80 characters long.

INPUTS/OUTPUTS -- Input formal parameter "FNAME" is the text file to be read. The output is the text record buffer "LINE".

GETTOKEN (Get Token)

FUNCTION -- The function is to scan a line of text and return the next "token". A token is defined as a string of alphanumeric characters separated by delimiters, where a delimiter is any non-alphanumeric character.

INPUTS/OUTPUTS -- Input formal parameter "FNAME" is a text file. The file is 80 characters long and represents a single input character buffer. Output is the global word "TOKEN" where the found token is stored left justified, blank filled.

SKIPBLK (Skip Blanks)

FUNCTION -- The function is to pass over all consecutive blank characters in the input record buffer "LINE", starting with the current character pointer.

INPUTS/OUTPUTS -- The input is the current character position of text file "LINE". The output is the new character position of "LINE".

EQUALOK (Equal Ok?)

FUNCTION -- The function is to syntax check the presence of an equal sign, used in many of the user input statements. If the equal sign is not present, the proper error message is given.

INPUTS/OUTPUTS -- The input is the current position of the text file "LINE". The output is the new position of "LINE". If the check for the equal sign fails, the output also includes an error message.

CHKPERIOD (Check Period)

FUNCTION -- The function and inputs/outputs for CHKPERIOD are the same as those for EQUALOK. The difference in the two procedures is that CHKPERIOD checks for the presence of a period (.).

ERROR

FUNCTION -- The FUNCTION is to write the error message passed it and set the global error flag "ERRORFOUND".

INPUTS/OUTPUTS -- The input is an error message constant, 50 characters long. The ouput is the error message being written to the printer and

the boolean "ERROFFOUND" being set to true.

The GENERATEALL Procedure Documentation:

```
GENERATEALL

   (* Main Procedures *)
   GENHEADER
   GENSCHEMAS
   GENINITIAL
   GENOPEN
   GENREAD
   GENCONVERSION
   GENCLOSE
   GENERRORSUBS
   GENCMDFILES

  (* Utility Procedures Global to GENERATEALL *)
   WRITAG
   WRIWRD
   WRITINTEGER
   WRITECONT
```

GENHEADER (Generate Header)

FUNCTION -- This procedure generates the heading of the PLI FORTRAN program.

INPUTS/OUTPUTS -- The inputs are globally defined single word arrays containing data about the source inputs file. The output is the FORTRAN program statement, which includes the declaration of external files, and a program comment explaining the purpose of the program.

GENSCHEMAS (Generate Schema Declarations)

FUNCTION -- This procedure generates the common block declarations for

the necessary System 2000 communication areas. One common block declaration is required for each declared database repeating group.

INPUTS/OUTPUTS -- The input data is the target symbol table. The output is the generated common block declarations.

GENINITIAL (Generate Initialization Code)

FUNCTION -- This procedure performs six generations: 1. generation of real declarations for target database components of type REAL; 2. generation of the global arrays "BUF" and "SRC"; 3. generation of a parameter declaration for the "EMPTY" character; 4. generation of FORMAT statements used in the FORTRAN program; 5. generation of initialization code for the global buffers, and 6. generation of a print header for the FORTRAN program.

INPUTS/OUTPUTS -- The corresponding inputs for the six generations are: 1. the target symbol table; 2. the source symbol table; 3. the globally defined word "EMPTY"; 4. no input--a constant; 5. source symbol table, and 6. globally defined words "DBNAME", and "FILENAME". The outputs are the generation of the code, as described above.

GENOPEN (Generate Database Open Code)

FUNCTION -- This procedure generates the code for opening the target S2K database. All of this code is the same for each job, with only the database name and password being different.

INPUTS/OUTPUTS -- The inputs are the words "DBNAME", and "USERID", containing the database name and the password respectively. The output

is the generated code, as described above.

GENREAD (Generate Read Code)

FUNCTION -- This procedure generates the unformatted read statements, the decode statements and the format statements used by the decode statements. When generating this code, each decode statement must decode less than 150 characters of the input record, and must terminate each decode statement on an even 10 character word boundary.

INPUTS/OUTPUTS -- The input is completely contained in the source symbol table. The output is the generated code, as described above.

GENCONVERSION (Generate Converson Code)

FUNCTION -- Each entry of the conversion symbol table is read and code is generated for it. A procedure for each conversion statement is declared within GENCONVERSION, and is called depending on the statement type of the conversion symbol table entry.

INPUTS/OUTPUTS -- The inputs are the conversion symbol table and the source symbol table. The source symbol table is needed to determine which array word of "SRC" the source value resides. The output is the code to move the source fields into their respective target fields, to store the data sets using the S2K INSERT command, and generate FORTRAN DO LOOPS which will properly iterate through the entire source file.

GENCLOSE (Generate Database Close Code)

FUNCTION -- This procedure produces the code to close the database and

produce a summary print out of the job.

INPUTS/OUTPUTS -- This code generation is the same for all job. The differences are the database name and the print header generated. The input, therefore, is the word "DBNAME", and the output the generated code, as described above.

GENERRORSUBS (Generate Error Subroutine Code)

FUNCTION -- This procedure generates the code for the error subroutine. This subroutine is called in the FORTRAN program for all S2K database errors detected during execution. Because this is a standard subroutine, the same code is generated for all jobs.

GENCMDFILES (Generate Command Files)

FUNCTION -- This procedure generates the files "COMPILC" and "EXECC". These files contain the unique UT-2D control commands necessary for the user to generate the FORTRAN program (file "COMPILC") and execute it (file "EXECC"). Details on these processes are contained in the User's Manual.

INPUTS/OUTPUTS -- The inputs are many globally declared words containing information on the source input file, the target database file name and location, and execution options availible to the user. This data is gained primarily from the user's Command Language input and the first portion of the user's Source Language input. The output are the two control command files. A complete example of these files can be found in Appendix B.

wRTTAG (write Tag)

FUNCTION -- This procedure writes the leading tag for each generated line of FORTRAN code. This tag is later used to sort on in order to rearrange the generated FORTRAN code in proper sequence.

INPUTS/OUTPUTS -- The input is the formal parameter "INTAG" which contains an integer value to output. The output is a 10 character integer written to the file "FORTSRC", right justified, zero filled.

wRT (write String)

FUNCTION -- This procedure writes the string passed to it, to the current line of file "FORTSRC". FORTSRC is the file containing the generated FORTRAN program.

INPUTS/OUTPUTS -- The input is the formal parameter "STRING". This string is a 50 character constant which will be written, either in full or until the first percent (%) is found. The output is the constant string being written to the file "FORTSRC".

wRTwRD (write word)

FUNCTION -- This procedure performs the same function as wRT, except it will write a word to the file "FORTSRC" instead of a string.

INPUTS/OUTPUTS -- The input is the formal parameter "WORD" which is a 10 character word. The output is writing this entire word to the file "FORTSRC" or until the first blank is encountered.

WRITINTEGER (Write Integer)

FUNCTION -- This procedure performs the same function as WRT, but it write an integer, in character form, to the file FORTSRC.

INPUTS/OUTPUTS -- The input is the formal parameter "INT" which is an integer. This integer will be converted to its character form, then written to FORTSRC. Only as many characters the integer is long will be written.

WRITECONT (Write Continue)

FUNCTION -- This procedure will generate a FORTRAN CONTINUE statement with the tag and label as input in the parameters.

INPUTS/OUTPUTS -- The inputs are the formal parameters "INTAG" and "FORMATNUM". The output is a FORTRAN CONTINUE statement with the tag of INTAG and the label of FORMATNUM. This statement is used as a label to indicate the end of a DO LOOP.

2.6 Final Comments

This report concentrated on a specific subset of the data conversion requirement. This subset was converting a source file to a defined target database. The source file must be hierarchically describable and the target database must be defined and maintained by the System 2000 DBMS. Although the solution to this subset is a system limited only to the specific requirement, the ideas, system

architecture, and algorithms presented here are applicable to many data conversion requirements. The idea of implementing a small, simple system to satisfy a specific conversion requirement gives the system a better chance of succeeding and being used. As an example, the user's manual for Michigan's Data Translation system is 355 pages long. The user may be able to write a unique program to satisfy his conversion need faster than learning a system as large as Michigan's. The common architecture of data conversion systems presented in this paper may be used as a basis for design of any new conversion system. The specific design options taken by the Source to S2K System appear to be the best choices, however, a new conversion requirement would, obviously, dictate the best choice for the design of its system. Finally, the algorithms for implementing the data transformations are simple and satisfactory for a procedurally oriented system.

The transportability of the Source to S2K Conversion System is one of its weakest points. The system was written in PASCAL and generates a non-ASCII standard FORTRAN program and job control commands executable only by the UT-2D operating system. These languages were chosen because they are the best supported languages at the implementation site. If a conversion system of this type is intended for more than one organization and/or machine, the system should be written in COBOL or ALGOL 60, generate a COBOL PLI program, and generate job control commands for a standard operating system. COBOL allows for easier record description and editing than FORTRAN. ALGOL 60 does not offer any advantages over PASCAL, but it is supported on more machines than PASCAL.

The final comment to be made is on the philosophy of the Source to S2K Conversion system. It was designed with the philosophy of aiding the user in the conversion task rather than completely accomplishing it. This philosophy has several advantages. First, the implementation is simplified by not designing for unusually complicated conversion requirements which might arise. Second, the user's input is reduced and simplified since he does not have to translate some complex portion of his conversion requirement into an even more complex user's language. Finally, the conversion execution may be improved when the user is allowed to access the generated program prior to its execution. Designing and implementaing these conversion systems with the idea of helping the user rather than ensuring completeness will give the system a better chance of being used. The Source to S2K Conversion System has achieved that goal.

APPENDIX A

THE SOURCE TO S2K CONVERSION SYSTEM

USER'S MANUAL

December 1978

USER'S MANUAL

TABLE OF CONTENTS

SOURCE TO S2K CONVERSION SYSTEM USERS MANUAL

SECTION 1 -- GENERAL DESCRIPTION

1.A INTRODUCTION:

The Source to S2K Conversion System gives The user the capability of generating a complete S2K PLI FORTRAN program which, when executed, will load his defined S2K database with his described source input. The UT-2D control commands needed to execute the PLI program are also automatically generated. When the user wishes to perform the actual database load, a single card input is all that is needed. This two step process, generation of the load program and actual execution, provides the user with added flexibility. Because the generated program is stored as a permanent file and is accessible to the user before execution, the user may modify the generated program as much or as little as he desires. For example, should the user wish to generate only a "skeleton" load program and then write his own conversion routines, this can be done. Or the user may generate the complete program and then optimize heavily used routines for improved efficiency.

The intent is to automatically generate all source code, yet give the user complete control of the final program.

1.B SYSTEM CHARACTERISTICS:

The system is designed to convert a single source input file into a single S2K defined database. It is also possible to append new source data sets to an existing database. This feature is discussed in Section 1.D. The system supports a tree like hierarchical data model. Thus, all input data must be in a form that can be described in a hierarchical manner. Since S2K is the target database, all database terminology will be consistent with that used in the S2K documentation.

1.C SYSTEM USAGE:

In order to generate a complete PLI load program, the user is required to make several inputs. First, descriptions of the source and target files are necessary. The source file is described using the Source Description Language, as defined in Section 2.C. The target database is described using the same input as that used when the target S2K database was described to the S2K system. Documentation of this input is contained in the Basic S2K documentation (see Define Module). Next, a procedurally oriented Conversion Language is used to define the mappings between the source and target data fields. Since no data movement is automatically assumed, each target field must have at least one conversion statement describing how its data values are attained.

The three major inputs, the source, target, and conversion descriptions, constitute the majority of the user required input. In addition, a Command language is used to tell the system general characteristics about the job (user code, password, etc). All of these languages are fully described, with examples, in Section 2. A comprehensive example of an entire run is contained in Section 3.

1.0 LOADING DATA TO EXISTING DATABASES:

Occasionally an initial load may have several source input files. If it is not practical to combine these files into a single file, it is possible to generate several PL1 FORTRAN conversion programs which will initially load the database and then continue to append data sets to the existing database. A boolean expression is input which will identify a level 0 data set (see Section 2.B -- TYPE card). If the level 0 data set exists, the new data sets will be appended to it. If the boolean expression is not satisfied, (i.e. the level 0 data set does not exist), a new level 0 data set will be created that does satisfy the expression. Then the input data sets will be appended to the newly created level 0 data set. Note the restriction that the boolean expression identifies a level 0 data set. This means that the adding of data sets starts at level 1 (if the level 0 data set already exists), or level 0 (if it does not already exist). For example, suppose the input source consists of two files, one of DEPARTMENT data and the other of EMPLOYEE data. Suppose the target database desires level 0 data sets of DEPT data and level 1 data sets of the employees working in that Department. First, the DEPT data would be loaded,

followed by a run for the EMPL data. Each employee record would be read and a search would be made for the department he works in. The search is expressed by a boolean expression, such as C3 EQ S3 where C3 is the S2K component number for Department Name, and S3 is the Employee file component number for DEPT-WORKS-IN. Obviously, if the employee file did not have a field specifying which department he worked in, it would be impossible to realize the desired target database. Additional restrictions on the creation of the boolean expression are contained in Section 2.B -- TYPE card.

SECTION 2 -- LANGUAGE DESCRIPTIONS

2.A GENERAL RULES and RESTRICTIONS:

1. There are 4 required language description modes. Before submitting input to any mode, input the following card (* starts in col 1):

** <mode name> .

where <mode name> = COMMAND (general description)
SOURCE (source input file description)
S2K (S2K schema description)
CONVERSION (source to target mappings)

2. All input must be submitted on cards. Each language statement must be terminated with a period (.), and contained on a single card. Whenever a single blank is syntactically legal, any number of consecutive blanks are also legal.

3. The formal syntax is described using a "railroad track" notation. Syntactically legal statements are derived by traveling the track from left to right. All required entries are in bold print. Entries contained in brackets ([ ]) indicate there are several options and to choose one.

4. Any error found in any description will prevent generation of the FORTRAN program. All source input will, however, continue to be checked for syntax.

1. This notation is used by the Burroughs Corporation in their Programming Language Manuals. Niklaus Wirth also uses this notation in his description of PASCAL, referring to it as "syntax diagrams" [1].

2.8 COMMAND DESCRIPTION LANGUAGE

PURPOSE:

The Command Description Language is used to provide general information needed by the system. The only card required is the LOCATION card.

LOCATION Card:

The UT-2D system requires the S2K database to be stored under a specific file name and permanent library ID. The LOCATION Card is used to specify these inputs.

SYNTAX:

---- LOCATION = ---<filename>--/--<libr. id>--/--<password>---- .

where <filename> = The file name the S2K definition is stored and and the S2K database itself will be stored.
<libr. id> = The permanent library id number where the above file name resides.
<password> = The password for the permanent library id.

EXAMPLE:

LOCATION = DUNSDB/9294/1234.

The following optional cards may be input:

RUN Type Card:

The RUN type card specifies whether to generate a FORTRAN program and the UT-2D control commands, or check for syntax only. Default is full generation.

SYNTAX:

where S = Syntax only
F = Full generation (default)

EXAMPLE:

RUN = S.

TYPE Generation Card:

The TYPE generation card specifies whether the generated program is an initial load or an update program. If it is an update program (see Section 1.D), a correct boolean expression must be included which identifies a level 0 data set the new data is to be attached to. If the boolean expression is not satisfied, a new level 0 data set is created which does satisfy the expression. If more than one data set satisfies the boolean expression, the first data set found will be used to attach the new data to. The user is advised to select a boolean expression which will uniquely identify the desired level 0 data set. This will

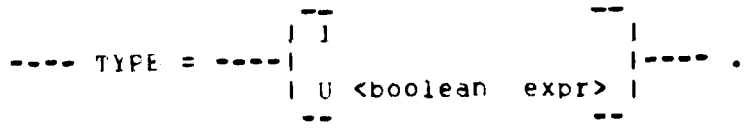
preclude erroneous database construction due to unknown input file record order. Since the boolean expression will be included in the FORTRAN program unaltered, its syntax should be the same as that described in the basic S2K Documentation, (see Procedural Language Fortran, PLF 6.6). The following additonal restrictions should be followed:

a.) No more than 10 S2K components may be used in a single boolean expression.

b.) Use complete Source component numbers (see Section 2.C) and S2K component numbers, (no component names allowed).

c.) All S2K components must be in the level 0 data set.

SYNTAX:



where I = initial load (default)
U = update load
<boolean expr> = a boolean expresxion which identifies a level 0 data set.

EXAMPLE:

```
TYPE = I.
TYPE = U C2 .EQ. 10HPROGRAMMER.
```

EMPTY Field Character:

This card is used to identify which input source character will signify null, or empty data. The default empty character is blank.

SYNTAX:

---- EMPTY = ---<character>--- .

where <character> = Any legal CDC character.

EXAMPLE:

EMPTY = %.

COMMAND LANGUAGE EXAMPLE:

```
** COMMAND.                  (enter command mode.)
LOCATION = DB/9892/1234.     (the S2K file name and location)
RUN = F.                     (generate FORTRAN program)
TYPE = I.                    (initial load run)
EMPTY = 0.                   (the input source character 0 means no data)
```

2.C SOURCE FILE DEFINITION LANGUAGE

PURPOSE:

The purpose of the Source Definition Language is to provide a means of describing the logical, storage, and physical characteristics of the incoming file. A thorough knowledge of the incoming source file is necessary.

GENERAL RULES and RESTRICTIONS:

1. The first input statement must be ** SOURCE.

2. All Source statements must be on a single card, one per card, each ending with a period. All input after the period is treated as a comment.

3. No variable length records may be described.

4. If the source file originated on the UT CDC 6400-6600 under control of the operating system UT-2D, the system will handle the file without any user intervention. If, however, the file originated elsewhere, the user may have to examine the generated FORTRAN READ module prior to actual execution of the conversion. This is because the UT-2D file system uses unique end-of-line and end-of-file markers. Foreign file formats may need to be read in an unorthodox manner to get the proper results. The user is advised to get the "foreign file" into a compatible CDC and UT-2D format.

SOURCE DEFINITION LANGUAGE STATEMENTS:

The storage and physical characteristics of the file are input first. These include the File ID and the Device Type. If the Device Type is TAPE, several additional statements must be input describing the tape's characteristics.

FILE Id Card:

The FILE id card is used to indicate the name of the input source file and its permanent library id or local tape id. If the file is a tape, input its tape number and password in place of the permanent library id and password. If the file has no name (such as a card file) then input the word NONE.

SYNTAX:

```
---- FILE = ---<file name>--/--<libr. id>--/--<password>--- .
```

where <file name> = The name of the input source file.
<libr. id> = The permanent library id or the local tape id for disk and tape files.
<password> = The password for the permanent library id or local tape id.

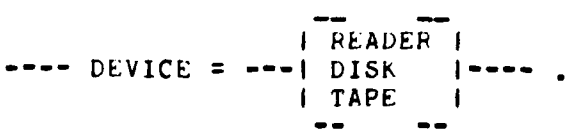
EXAMPLES:

```
FILE = DATA1/9294/1234.    (disk file)
FILE = NONE.               (card deck)
FILE = NONE/1234/5555.     (unlabeled tape)
```

DEVICE Type Card:

The input file device type can be either READER, DISK, or TAPE. No other device types can be handled.

SYNTAX:



EXAMPLE:

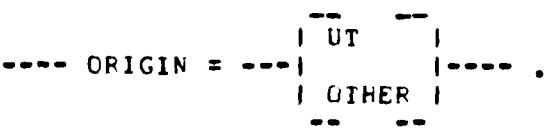
DEVICE = READER.

If the Device Type is TAPE, the following statements should be input, where appropriate.

Tape ORIGIN Card:

The tape's origin must be identified. This will tell the system whether the tape is in UT-2D tape format or a "foreign format".

SYNTAX:



where UT = The tape was written under UT-2D control (default).
OTHER = The tape was not written under UT-2D control.

EXAMPLE:

ORIGIN = OTHER.

RECORD Size Card:

If the tape origin is not UT, then the file's physical record size must be given. This card should not be used if the tape origin is UT.

SYNTAX:

---- RECORD = ---<n>---- .

where n = The decimal integer value of the physical record length in units of 12-bit bytes.

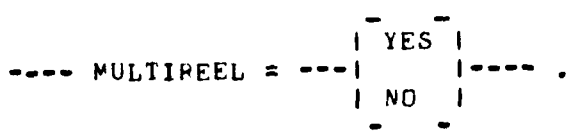
EXAMPLE:

RECORD = 100.

MULTIREEL Card:

The system must be notified if the input source file is more than one reel long. The default is single reel.

SYNTAX:



where YES = The file is more than 1 reel long.
NO = The file is 1 reel long (default).

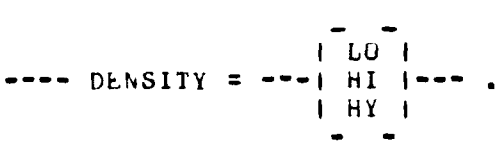
EXAMPLE:

MULTIREEL = YES.

DENSITY Card:

The tape's density must be input if it is not the default value of 556 BPI.

SYNTAX:



where LO = 220 BPI.
HI = 556 BPI.
HY = 800 BPI.

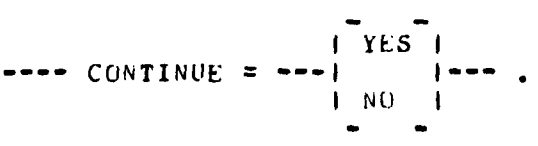
EXAMPLE:

DENSITY = LO.

CONTINUE After Parity Card:

The option of continuing processing after a parity error has been encountered is availible. The default option is to terminate the run if a parity error occurs.

SYNTAX:



where YES = Processing will continue regardless of num of parity errors.
NO = Processing will halt on occurrance of first parity error. (default)

EXAMPLE:

CONTINUE = YES.

EXAMPLES OF STORAGE and PHYSICAL DESCRIPTIONS:

I. Source file is a card deck.

```
** SOURCE.
FILE = NONE.
DEVICE = READER.
```

II. Source file is a single reel, UT produced tape.

```
** SOURCE.
FILE = DATA2/1346/1441.
DEVICE = TAPE.
DENSITY = HI.
ORIGIN = UT.
CONTINUE = YES.
```

III. Source file is a foreign multireel tape. Each record contains 1440 bits.

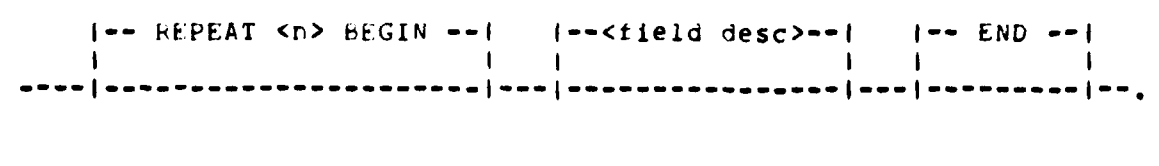
```
** SOURCE.
FILE = NONE/1334/1234.
DEVICE = TAPE.
DENSITY = HY.
ORIGIN = OTHER.
RECORD = 120.
MULTIREEL = YES.
```

LOGICAL DESCRIPTION of the SOURCE FILE:

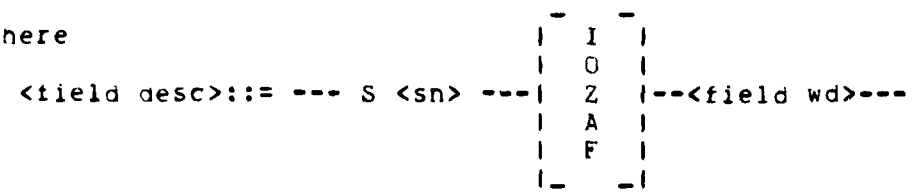
The logical description of the file is, essentially, its file layout. Since only fixed length records may be defined, the description language is quite straight forward. All fields must be identified with an "S" and a unique integer, starting with 1 and incrementing by 1. To describe the field's contents, an editing identifier is used, followed by the field's size. For example, if the first source field were

-NAME-, a 25 alphanumeric character field, the source definition would be S1 A25. This syntax is very similiar to FORTRAN FORMAT conversion and editing specifications. If a field or group of fields repeat themselves in the file layout, the REPEAT <n> BEGIN ... END verbs may be used. The fields described between the BEGIN and END statements will be repeated n times. When using the REPEAT verb, the REPEAT, the field description statements, and the END card must all be on seperate cards.

SYNTAX:



where



<n> = The number of times the fields are to be repeated.
<sn> = The unique field number, starting with 1, incrementing by 1.
<fld wd> = This integer represents the number of bytes in the field. If the editing specification is F, the number of digits to the right of the decimal point must be input, ie, 9.3.

Editing Specifications:

I = Field's bytes represent an integer.
O = Field's bytes represent 3 bit octal integers.
Z = Field's bytes represent 4 bit hexidecimal int.
A = Field contains alphanumeric characters.
F = Field is a decimal number. Number of decimal characters must be given in the <fld wd> specification . ie. 9.3 would mean the field is 9 bytes long, with 3 digits to the right of the decimal point.

EXAMPLE:

Input Record--

jon	29	3.60	264	5141B	3	5144A	2	5116A	10

LOGICAL DESCRIPTION--

```
S1 A6.              Name  (this is a comment field)
S2 I2.              Age
S3 F5.2.            Salary per hr.
S4 U4.              Days worked
REPEAT 3 BEGIN.     Start -SKILLS- RG
   S5 A5.           Skill code.
   S6 I2.           Num years experience at this skill.
END.                End -SKILLS- RG
```

2.0 TARGET S2K DATABASE INPUT

PURPOSE:

The purpose of inputting the S2K database definition is to define, for the program, the target database. The cards submitted here must be the same define cards used to define the S2K database. These include the database name and database password declaration cards. Since these inputs should Since these inputs should have already been submitted to the S2K system, they are assumed to be syntactically and semantically correct. If the user described his database interactively and does not have card input, a proper deck may be produced by

1. Change the S2K REPORT file to a temporary disk file.
2. Issue a DESCRIBE command.
3. Dump the temporary disk file to PUNCH.

S2K DATABASE DEFINITION INPUT EXAMPLE:

** S2K.
< S2K database definition card deck >

2.E CONVERSION DEFINITION LANGUAGE

PURPOSE:

The purpose of the Conversion Definition Language is to describe how each target field's data value is derived and whether there are any conversion or validation procedures to be applied to it.

GENERAL RULES and RESTRICTIONS:

1. The first input statement must be ** CONVERSION.

2. Each conversion statement must be contained on a single card, one to a card, ending with a period. Text after the period is treated as a comment.

3. Only component numbers may be used when referring to the target fields. No component names may be used.

4. No moving of source to target data will take place without an explicit conversion statement. Therefore, there must be at least one conversion statement for each defined target file field.

5. The system makes no semantic analysis of the conversion statements. The user must ensure the logical correctness of its statements. Execution will be in the order of the inputted conversion statements. Take care to ensure all desired data transformations are stated before the END statement is input.

LANGUAGE DESCRIPTION:

There are 7 different conversion statements, grouped into three categories: Data Transformations, Conversion and Validation Operations, and the special STORE statement.

DATA TRANSFORMATION STATEMENTS:

The basic transformations required to restructure hierarchically modelled data structures are:

1. Lateral--Move values from source to target fields which are on the same corresponding level.

2. Down--Move a single source value into each of its lower level (descendant) target fields.

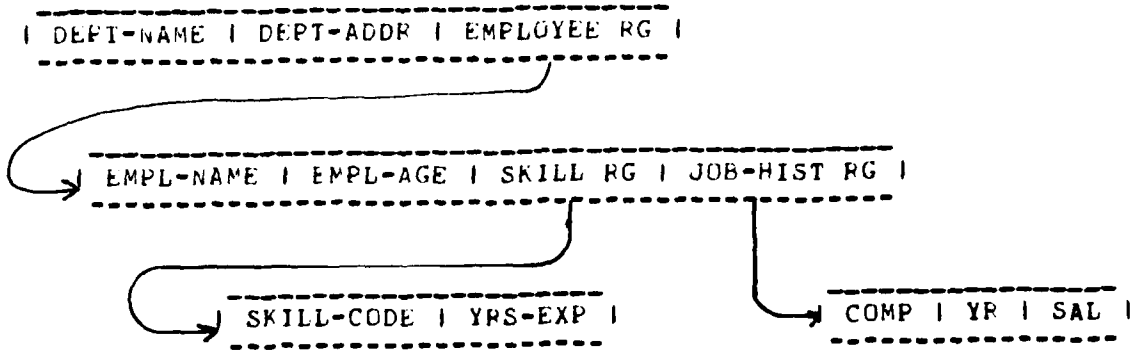
3. UP--Move a single occurrence of a lower level source repeating group field up to a target entity, or perform an operation on all members of the source repeating group field and move this single result up to the target field.

The central concept necessary to comprehend data transformations is that of "correspondence" between the source and target data group levels. Although a source and a target group may be on different hierarchical levels, they may be in correspondence. A more formal definition of this concept is:

> correspondence: A target group X corresponds to a source group Y if for every group instance in X there exists a unique group instance in Y.

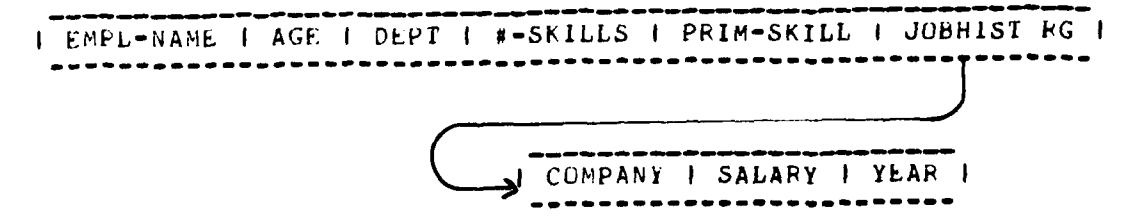
Consider the following source and target file descriptions:

SOURCE FILE DESCRIPTION




```
S1  A10.              Department name.
S2  A20.              Department address.
REPEAT 10 BEGIN.      Start EMPLOYEE RG (max 10 sets)
   S3 A15.            Employee Name.
   S4 I2.             Employee Age.
   REPEAT 3 BEGIN.    Start SKILL RG (max 3 sets).
      S5 A5.          Skill code.
      S6 I2.          Years experience at this skill.
   END.               End of SKILL RG.
   REPEAT 5 BEGIN.    Start JOB-HIST RG.
      S7 A15.         Company name.
      S8 I2.          Year started with company.
      S9 F9.2.        Yearly Salary.
   END.               End of JOB-HIST RG.
END.                  End EMPLOYEE RG.
```

TARGET (S2K) FILE DESCRIPTION



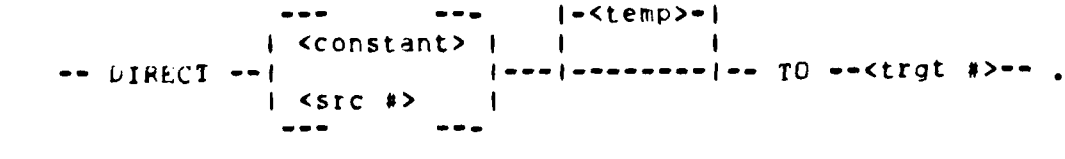

```
 1*  EMPL-NAME(NAME X(15))
 2*  EMPL-AGE (INTEGER 9(2))
 3*  PRESENT-DEPT (NAME X(10))
 4*  NUM-OF-SKILLS (NON-KEY INTEGER 9)
 5*  PRIMARY-SKILL (NAME X(5))
 6*  JOBHIST (RG)
  7*  COMPANY (NON-KEY NAME X(15) IN 6)
  8*  SALARY (NON-KEY MONEY 9(9).99 IN 6)
  9*  YEAR-STARTED (NON-KEY INTEGER 99 in 6)
```

The source file contains three hierarchical levels, while the target file contains two. The source file's second level (EMPLOYEE record) corresponds to the target's first level (EMPLOYEE). Thus, desired data on the first source level would have to be moved "down" to the target file (i.e. DEPT-NAME to DEPT). Likewise, desired level 3 source data would either be moved up (i.e. NUM-OF-SKILLS) or moved laterally across (i.e. all fields in JOBHIST repeating group). Specific examples of these operations are presented with the discussions of each data transformation operation.

DIRECT Statement:

DIRECT will perform the "lateral" data transformation, as previously defined. Both source and target fields must be on the same corresponding levels. If a constant value is desired in a target field, the constant may be input in place of a source component number. This should be a legal FORTRAN constant, i.e. nH should precede character typed data.

SYNTAX:



where <src #> = A component number from the source file definition.
<constant> = Any legal value which will be placed in all occurrences of the target field.
<temp> = A FORTRAN variable used in a user written CONVERSION statement.
<trgt #> = A component number from the target S2K database definition.

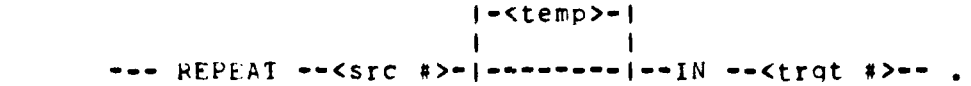
EXAMPLES:

```
DIRECT S3 TO C1.            Employee Name.
DIRECT S7 TO C7.            Company name (JOBHIST RG).
DIRECT S9 TEMPREAL TO C8.   Converted Salary.
DIRECT 9999.99 TO C8.       Constant put in Salary.
```

REPEAT Statement:

REPEAT will perform the "down" data transformation, as previously defined. The source field, which is at a higher corresponding level then the target field, will be moved "down" to the target field.

SYNTAX:



where <src #> = A component number from the source file definition.
<temp> = A FORTRAN variable used in a user written CONVERSION statement.
<trgt #> = A component number from the S2K database definition.

EXAMPLES:

```
REPEAT S1 IN C3.            Dept-name.
REPEAT S1 TEMPINT IN C3.    Converted Dept-name.
```

LEVELUP Statement:

LEVELUP performs the "up" data transformation for the case of moving a specific occurrence of a source repeating group field up to the target field. The specific occurrence is indicated by the clause "I=<n1,n2,...nn>". The values of <n1,n2,...> represent the specific occurrences of each of the field's ancestor data sets, with the last value <nn> representing the occurrence of the field's repeating group itself. The order of "n" should be input in the same hierarchical order as the database schema. Therefore, the first occurrence of a field which is two levels down would be indicated by "I=n,1", where "n" represents the occurrence of the field's parent. If the parent's third occurrence is desired, "I=3,1" would be the proper input. In most cases, the LEVELUP transformation will move source fields that are only one level away. For example, the proper input for the second occurrence of a repeating group only one level down would be "I=2". All occurrences must be referenced by an integer except the "last" occurrence in a repeating group. Since the last occurrence may be a different relative number for each set, the term "LAST" may be used in place of the integer "n".

SYNTAX:

where <src #> = A component number from the source definition.
<temp> = A FORTRAN variable used in a user written CONVERSION statement.
<trgt #> = A component number from the target S2K database definition.
<n1,n2..>= The relative occurrence number for the field's ancestor groups and the groups the field resides in. "n" may also be the term "LAST".

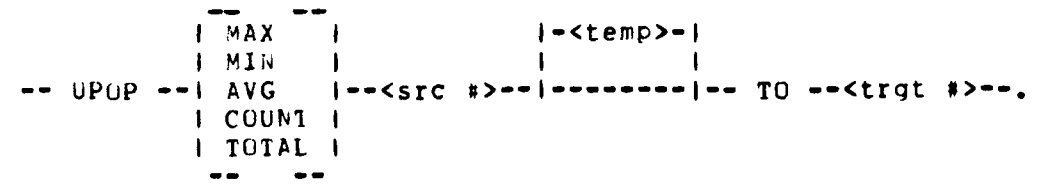
EXAMPLE:

```
LEVELUP S5 I=1 TO C5.           First Skill Code.
LEVELUP S5 I=LAST TO C5.        Oldest (last) Skill Code.
LEVELUP S5 I=1 TEMPCHAR TO C5.  Converted Skill Code.
```

UPOP Statemment:

UPOP (Up-operation) performs the "up" data transformation for the case where an operation is performed on all occurrences of a source repeating group field, deriving a single result from the operation. This single result is then moved up to the target field. The availible operations include MAX, MIN, AVG, COUNT and TOTAL.

SYNTAX:



where MAX = The source field's largest value in the RG.
MIN = The source field's smallest value in the RG.
AVG = The source field's average value.
COUNT = The number of sets in the source RG where the source field's value is anything but -null-.
TOTAL = The total of the source field's values in the RG.

EXAMPLE:

```
UPOP COUNT S5 TO C4.  Number of Skills
```

VALIDATION AND CONVERSION STATEMENTS:

Occasionally the source data values are not in the format or content desired for the target record. Also, editing of the source input is sometimes desired to provide increased data integrity. These two capabilities are provided by the CONVERSON and VALIDATE statements. Because it is impossible to predict the type of conversion or validation routine a user may need, these statements only provide the means for the user to write the actual conversion/validation code necessary. The user written code is incorporated in the generated FORTRAN program unaltered (except for the source component number). Thus, the user must adhere to proper syntax, column spacing, etc.

CONVERSION Statement:

The CONVERSION statement gives the user the capability of inputting FORTRAN source code which will execute desired conversions on source fields. If possible, the results of the conversion should be placed in the original source field. If however the result is a value which will not legally fit in the original source field, the result should be placed in a temporary variable. This temporary variable should be one of the following, depending on the type of the result:

If result =	Use temporary
integer	TEMPINT
real	TEMPREAL
character	TEMPCHAR(1..N)

If the result is character, left justify the characters, (10 characters

per word), in the variable "TEMPCHAR" starting with index number 1. The proper TEMPCHAR subscripts must be used in the user written FORTRAN code. However, no subscript should be input when referring to TEMPCHAR in a data transformation statement. Subscript "1" will be assumed.

Since the CONVERSION statement only alters the value of the source field, a data transformation statement must be used to actually move the converted value to a target field. If a temporary variable is used, both the original source field component number and the temporary name must be included in the transformation statement. The temporary name must be in the statement to tell the system the source value is in the temporary and not the source field.

SYNTAX:

```
---- CONVERSION BEGIN. ----
----<user written FORTRAN code>----
---- END. ----
```

EXAMPLES:

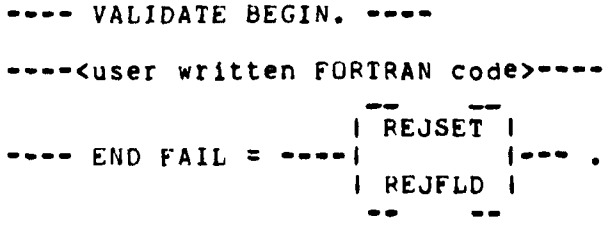
```
CONVERSION BEGIN.
CCC
C     Add 1 year to each EMPLOYEE-AGE.
CCC
      TEMPINT = S4 + 1
END.
DIRECT TEMPINT TO C2.
```

```
CONVERSION BEGIN.
CCC
C     Change all "5661B" skill codes to "9661B"
CCC
      IF (S5.EQ.5H5661B)  S5=5H9661B
END.
LEVELUP S5 I=1 TO C5.
```

VALIDATE Statement:

VALIDATE gives the user the capability of validating a particular source field value before it is stored in the database. The user must write the validation code, just as is necessary for the Conversion statement. Somewhere within the user written code the FORTRAN variable FAIL must be set to TRUE or FALSE. If the validation of the source field fails (i.e. FAIL = TRUE), the user may choose to reject the data set being processed (REJSET) or put nulls in the source field and continue processing the data set (REJFLD). If the user wishes no action to be taken on a validation failure, the Conversion statement should be used instead of the Validation statement. No data transformation operation is associated with the Validation statement, as is the case with the Conversion statement. Thus, the proper data transformation operation must be input following the Validation statement.

SYNTAX:



EXAMPLE:

```
VALIDATE BEGIN.
CCC
C     Validate AGE--  18<=AGE<=75
CCC
      FAIL=.FALSE.
      IF (S4.LS.18  .OR.  S4.GT.75)  FAIL=.TRUE.
END FAIL = REJFLD.
DIRECT S4 TO C2.
```

STORE Statement:

S2K data sets must be built in the hierarchical order that they are defined. A Level 0 data set must be created before its descendant data sets may be "attached" to it. Data sets are created by loading the fields with the desired data and "storing" the data set. Using this system, the user loads the data fields using the data transformation statements previously defined. He must also "store" the data set by inputting the STORE statement. These statements should be input immediately after the data transformation statement for the last field in each data set. The set name should be the same as that used in the S2K database definition. Use the name "LEVEL0" for the Level 0 set name.

SYNTAX:

---- STORE ---<data set name>--- .

where <data set name> = The name of the data set, as defined in the S2K databaseinput.

EXAMPLES:

```
STORE LEVEL0.
STORE JOBHIST.
```

SECTION 3 -- SYSTEM USAGE

This section will give instructions on how to generate a FORTRAN program, how to review and modify it, and how to execute it. A complete example input and resulting generated program is then presented.

3.A HOW TO GENERATE A FORTRAN PROGRAM

To generate a FORTRAN program, the user must first have a card deck containing the required Command language input, Source and Target file descriptions, and the Conversion language inputs, (see Sections 1 and 2 of this manual). This deck will serve as input to the system. The complete job set-up is shown below. Note that the user is required to input only a single command card with the input card deck. All of the other commands needed are supplied by the Source to S2K System.

Job Set-up for Generating a FORTRAN Program:

```
<user id>
<password card>
<run card>                              (optional)
READCCF, 9294, GENRATE
7/8/9                                   (multi-punched)
<users complete input card deck>
6/7/8/9                                 (multi-punched)
```

The user will receive output from the system showing what was input and any error messages. If there were no errors, a FORTRAN program is generated and passed to the compiler. The user will then receive the compiler output. It is possible to have FORTRAN syntax errors in the

generated program due to erroneous user input the Source to S2K System did not find. The user may correct these errors in two ways. The first is to correct the original input and generate a new FORTRAN program. The second is to modify the generated FORTRAN program itself. This procedure is described in Section 3.B.

3.B HOW TO MODIFY THE GENERATED FORTRAN PROGRAM

Each generated FORTRAN program must have a unique name, otherwise different users would be erasing each others's files. Thus, the name of the generated FORTRAN program is the first four letters of the database name followed by the letters "SRC" (for source). For example, if the database name is EXAMPL1, the generated FORTRAN program would be stored under file name "EXAMSRC". All source files are stored on permanent library 9294. Thus, in order to edit the FORTRAN program for the EXAMPL1 database, the command

READPF, 9294, EXAMPL1

is all that is needed. The user can then modify this file using the UT-2D editor EDIT. If batch editing is required, dumping the file to PUNCH will produce a card deck of the source FORTRAN program.

3.C HOW TO EXECUTE THE GENERATED PROGRAM

Once the generated program is free of errors, the user is ready to perform the actual conversion. A file containing all of the required

control commands for each job is generated by the Source to S2K System at the same time it generates the FORTRAN program. Since the file must have a unique name, it is made up of the letters "EX" (for execute) followed by the first four letters of the database name. Using the example database EXAMPL1, the generated command file name would be "EXEXAM". This file will also be stored on permanent library 9294. The only input needed to execute the generated conversion program is shown below. If the source input is on cards, they should be included in the deck after the 7/8/9 multi-punched card.

Job Set-up for Executing the Generated FORTRAN Program

```
<user id>
<password card>
<run card>                          (optional)
READCCF, 9294, EXEXAM               (file name will be different for each job)
7/8/9                               (multi-punched card)
<if source input is cards, input them here>
6/7/8/9                             (multi-punched card)
```

Execution of the above job deck would result in execution of the generated FORTRAN program as well as saving the S2K database on the permanent library (as directed by the LOCATION card, see Section 2.B). Should the user need to modify the generated command file, it may be done in the same manner as modifying the generated FORTRAN program (see Section 3.B).

AD-A106 282 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2
THE SOURCE TO S2K CONVERSION SYSTEM.(U)
DEC 78 J L STEVENS
UNCLASSIFIED AFIT-CI-79-262T NL

3.D COMPLETE EXAMPLE

This example will use the source and target databases described in Section 2.E. The system will generate a program to convert the source input file "DBINPUT" to the S2K database "EXAMPL1". The S2K database description is stored in file "EX1DESC" on permanent library 9899 (password 1221). The source file (DBINPUT) is a disk file on permanent library 6656 (password 1334). A null field will be indicated by the blank character.

Job Input for Database EXAMPL1.

```
** COMMAND.
LOCATION = EX1DESC/9899/1221.
RUN = F.
TYPE = I.
EMPTY =  .
** SOURCE.
FILE = DBINPUT/6656/1334.
DEVICE = DISK.
S1  A10.              Department name.
S2  A20.              Department address.
REPEAT 5 BEGIN.      Start EMPLOYEE RG (max 5 sets).
   S3 A15.            Employee name.
   S4 I2.             Employee age.
   REPEAT 3 BEGIN.    Start SKILL RG (max 3 sets).
      S5 A5.          Skill code.
      S6 I2.          Years exp. at this skill.
   END.               End of SKILL RG.
   REPEAT 3 BEGIN.    Start JOBHIST RG.
      S7 A15.         Company name.
      S8 I2.          Year started with company.
      S9 F9.2.        Yearly salary.
   END.               End of JOBHIST RG.
END.                  End of EMPLOYEE RG.
** S2K.
USER,PASS1
NEW DATA BASE IS EXAMPL1
1*  EMPL-NAME(NAME X(15))
2*  EMPL-AGE (INTEGER 9(2))
3*  PRESENT-DEPT (NAME X(10))
4*  NUM-OF-SKILLS (NON-KEY INTEGER 9)
5*  PRIMARY-SKILL (NAME X(5))
6*  JOBHIST (RG)
 7*  COMPANY (NON-KEY NAME X(15) IN 6)
 8*  SALARY (NON-KEY MONEY 9(9).99 IN 6)
```

```
 9*  YEAR-STARTED (NON-KEY INTEGER 99 IN 6)
** CONVERSION.
DIRECT S3 TO C1.              (Employee Name)
DIRECT S4 TO C2.              (Employee age)
REPEAT S1 IN C3.              (Department name)
UPOP COUNT S5 TO C4.          (Number of skills)
LEVELUP S5 I=1 TO C5.         (Primary skill)
STORE LEVEL0.                 (End of LEVEL0 data set)
DIRECT S7 TO C7.              (History-company name)
DIRECT S9 TO C8.              (History-salary)
DIRECT S8 TO C9.              (History-year started)
STORE JOBHIST.                (End of JOBHIST data set)
```

The following FORTRAN program was generated by the Source to S2K Conversion System as a result of the above input.

```
      PROGRAM PROGEX1 (DBINPUT, OUTPUT, TAPE1=DBINPUT)
      IMPLICIT INTEGER (A-Z)
C
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C *
C *     THIS PROGRAM WILL READ FILE "DBINPUT" AND CONVERT
C *     IT TO THE S2K DATABASE "EXAMPL1".  THE NEWLY
C *     BUILT DATABASE WILL BE STORED UNDER THE FILENAME
C *     "EX1DESC" ON PERMANENT FILE NUMBER 9899.
C *
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
C           -- COMMON BLOCK DECLARATION --
*PL   COMMBLOCK/EXAMPL1/ SCHNME, RCODE, FILLER, LDSET, PASSW, NUMRG,
*PL                      RGPOS,  LEVEL, TIMEX, SDATE, CYCLE, SEPSYM,
*PL                      ENDTERM, STATUSX.
C
C           SCHEMA NAME  -- LEVEL0 --
*PL   SCHEMA/LEVEL0 OF EXAMPL1/ C1(2), C2, C3, C4, C5.
C
C           SCHEMA NAME  -- JOBHIST --
*PL   SCHEMA/JOBHIST OF EXAMPL1/ C7(2), C8, C9.
C
*PL   END SCHEMAS.
C
C           -- REAL DECLARATIONS FOR SCHEMA --
      REAL   C8, TEMPREAL, AVG
C
C           -- GLOBAL DECLARATIONS --
      DIMENSION BUF(61),   SRC(108)
```

```
C
C           -- PARAMETER DECLARATIONS --
      EMPTCHR = 1
C
C           -- MISCELLANEOUS FORMATS --
   60 FORMAT(X,* INITIAL LOAD OF THE EXAMPL1 DATABASE*,//)
   62 FORMAT(X,* TIME = *,A10,/,X,* DATE = *,A10,/)
 1300 FORMAT(//,X,*-- EOF --*,/)
 1320 FORMAT(//,X,*-- CLEARED DATABASE.  CYCLE = *,I4)
 1500 FORMAT(//,X,*-- PARITY ERROR ON LAST READ.  BUF = *,/)
 1520 FORMAT(//,X,*-- FORMAT ERROR ON LAST DECODE.  BUF = *,/)
 5000 FORMAT(//,X,*-- SUMMARY OF INITIAL LOAD RUN FOR EXAMPL1*,/)
 5010 FORMAT(X,*NUMBER OF SOURCE RECORDS READ = *,I6)
 5020 FORMAT(//,X,*NUMBER OF I/O ERRORS = *,I6)
C
C           -- INITIALIZE LOCAL DATA --
      DO 55 I=1,108
      SRC(I) = 10H
   55 CONTINUE
      ERR = 0
      ICNT = 0
C
C           -- PRINT INITIAL PROGRAM HEADER --
      PRINT 60
      CALL TIME(I,J)
      CALL DATE(J)
      PRINT 62, I, J
C
C           -- OPEN DATABASE --
*PL   START S2K.
      PASSW = 10HPASS1
*PL   OPEN EXAMPL1.
      IF (RCODE.EQ.0 .AND. STATUSX.EQ.0)  GOTO 70
      CALL PRTERR(1,1,RCODE)
      GOTO 999
C
   70 CONTINUE
C
C           -- PUT IN QUEUE MODE --
*PL   QUEUE.
C
C           -- MAJOR READ LOOP --
  100 CONTINUE
      ICNT = ICNT + 1
      READ(END=900,1)  BUF
C
C           -- CHECK FOR PARITY ON LAST READ --
      IF (IOCHEC(1).NE.0)  GOTO 950
C
C           -- ECHO PRINT INPUT (EVERY TENTH RECORD) --
      I=MOD(ICNT,10)
      IF (I .EQ. 0)   PRINT *, BUF
C
C           -- DECODE STATEMENT NUMBER 1 --
      NUMCHAR = 120
```

```
      DECODE(.ERR.=960, NUMCHAR, 2010, BUF)   (SRC(I), I=1,20)
 2010 FORMAT(A10,A20,A15,I2,A5,I2,A5,I2,A5,I2,A15,I2,
     -       F9.2,A15,I2,F9.2)
C
C           -- SHIFT REST OF BUFFER TO WORD ONE --
      J=1
      DO 151 I = 13,61
      BUF(J) = BUF(I)
      J=J+1
  151 CONTINUE
C
C           -- DECODE STATEMENT NUMBER 2 --
      NUMCHAR = 90
      DECODE(.ERR.=960, NUMCHAR, 2020, BUF)   (SRC(I), I=21,37)
 2020 FORMAT(A15,I2,F9.2,A15,I2,A5,I2,A5,I2,A5,I2,A15,
     -       I2,F9.2)
C
C           -- SHIFT REST OF BUFFER TO WORD ONE --
      J=1
      DO 152 I = 22,61
      BUF(J) = BUF(I)
      J=J+1
  152 CONTINUE
C
C           -- DECODE STATEMENT NUMBER 3 --
      NUMCHAR = 90
      DECODE(.ERR.=960, NUMCHAR, 2030, BUF)   (SRC(I), I=38,54)
 2030 FORMAT(A15,I2,F9.2,A15,I2,F9.2,A15,I2,A5,I2,A5,I2,
     -       A5,I2)
C
C           -- SHIFT REST OF BUFFER TO WORD ONE --
      J=1
      DO 153 I = 31,61
      BUF(J) = BUF(I)
      J=J+1
  153 CONTINUE
C
C           -- DECODE STATEMENT NUMBER 4 --
      NUMCHAR = 100
      DECODE(.ERR.=960, NUMCHAR, 2040, BUF)   (SRC(I), I=55,70)
 2040 FORMAT(A15,I2,F9.2,A15,I2,F9.2,A15,I2,F9.2,A15,I2,A5)
C
C           -- SHIFT REST OF BUFFER TO WORD ONE --
      J=1
      DO 154 I = 41,61
      BUF(J) = BUF(1)
      J=J+1
  154 CONTINUE
C
C           -- DECODE STATEMENT NUMBER 5 --
      NUMCHAR = 130
      DECODE(.ERR.=960, NUMCHAR, 2050, BUF)   (SRC(I), I=71,95)
 2050 FORMAT(I2,A5,I2,A5,I2,A15,I2,F9.2,A15,I2,F9.2,A15,
     -       I2,F9.2,A15,I2,A5,I2,A5,I2,A5)
C
```

```
C           -- SHIFT REST OF BUFFER TO WORD ONE --
      J=1
      DO 155 I = 54,61
      BUF(J) = BUF(I)
      J=J+1
  155 CONTINUE
C
C           -- DECODE STATEMENT NUMBER 6 --
      NUMCHAR = 80
      DECODE(.ERR.=960, NUMCHAR, 2060, BUF)   (SRC(I), I=96,108)
 2060 FORMAT(I2,A15,I2,F9.2,A15,I2,F9.2,A15,I2,F9.2)
C
C           -- CONVERSION PROCESSING --
C
C           -- LEVEL0 DATA SET --
  300 CONTINUE
      DO 320  I2 = 1,5
      INDEX2 = 4 + (I2-1)*21
      C1 = SRC(INDEX2)
C
      INDEX2 = 5 + (I2-1)*21
      C2 = SRC(INDEX2)
C
      INDEX1 = 1
      C3 = SRC(INDEX1)
C
      COUNT=0
      DO 330  I3 = 1,3
      INDEX3 = 7 + (I2-1)*21 + (I3-1)*2
      IF ( SRC(INDEX3) .EQ. EMPTCHR )   GOTO 330
      COUNT=COUNT+1
  330 CONTINUE
      C4 = COUNT
C
      INDEX3 = 7 + (I2-1)*21 + (1-1)*2
      C5 = SRC(INDEX3)
C
*PL   INSERT LEVEL0.
      IF (RCODE.NE.0) CALL PRTERR(2,1,RCODE)
C
C           -- JOBHIST DATA SET --
  400 CONTINUE
      DO 410  I4 = 1,3
      INDEX4 = 28 + (I2-1)*21 + (I4-1)*12
      C7 = SRC(INDEX4)
C
      INDEX4 = 30 + (I2-1)*21 + (I4-1)*12
      C8 = SRC(INDEX4)
C
      INDEX4 = 29 + (I2-1)*21 + (I4-1)*12
      C9 = SRC(INDEX4)
C
*PL   INSERT JOBHIST.
      IF (RCODE.NE.0) CALL PRTERR(2,2,RCODE)
  410 CONTINUE
```

```
C
  320 CONTINUE
C
C           -- FINISHED WITH THIS RECORD.  LOOP BACK UP. --
      GO TO 100
C
C
C           -- EOF DETECTED ON LAST READ --
  900 PRINT 1300
C
C           -- CLOSE UP DATABASE --
*PL   IERMINATE.
      IF (RCODE.NE.0)  CALL PRTERR(4,1,RCODE)
*PL   CLEAR.
      PRINT 1320, CYCLE
*PL   CLOSE EXAMPL1.
      IF (RCODE.NE.0)  CALL PRTERR(3,1,RCODE)
*PL   END PROCEDURE.
      GOTO 999
C
C
C           -- PARITY ERROR DURING LAST READ --
  950 PRINT 1500
      PRINT *, BUF
      GOTO 100
C
C           -- FORMAT ERROR DURING LAST DECODE --
  960 PRINT 1520
      PRINT *, BUF
      ERR=ERR+1
      GOTO 100
C
C
C           -- PRINT JOB SUMMARY --
  999 CONTINUE
      PRINT 5000
      PRINT 5010, ICNT
      PRINT 5020, ERR
      END
C
C
C           -- SUBROUTINE PRTERR (PRINT ERROR) --
C
C  THE PARAMETERS ARE:
C           INST : INSTRUCTION NUMBER, WHERE
C                  1=OPEN,  2=INSERT,  3=CLOSE, 4=TERMINATE
C           LOC  : THE LOCATION IN THE PROGRAM THE ERROR
C                  WAS DETECTED.
C           RTNC : THE RETURN CODE THE S2K SYSTEM RETURNED.
C
      SUBROUTINE PRTERR(INST, LOC, RTNC)
 9000 FORMAT(/,X,* ----- DATABASE ERROR -----*,/)
 9010 FORMAT(X,*INSTRUCTION = *,I3,*  LOCATION = *,I3,
     -       *  RETURNCODE = *,I3)
C
```

```
PRINT 9000
PRINT 9010, INST, LOC, RTNC
RETURN
END
```

INDEX

APPENDIX B

Generated Command File Examples

The following example file listings contain UT-2D commands generated by the Source to S2K Conversion System.

FILE "GENRATE"

File "GENRATE" is called by the user to read his input and generate a conversion program. "GENRATE"'s commands, as listed below, are the same for each user and database.

```
File GENRATE
                EXECPF, 9294, SRCTS2K
                READCCF, 9294, COMPILC
```

FILE "COMPILC"

File "COMPILC" is called by file "GENRATE" during the conversion program generation phase. This file is unique to each

database user. The database name for this example is "DUNS". Note the name of the generated conversion program ("DUNSSRC") and generated execution commands file ("EXDUNS").

File COMPILC

```
REWIND FORTSRC
SORTPRG
RENAME FORTSRC DUNSSRC
SAVEPF, 9294, 3217, DUNSSRC
RENAME EXECC EXDUNS
SAVEPF, 9294, 3217, EXDUNS
REWIND DUNSSRC
PUBLIC, PLF, I=DUNSSRC, B=DUNSOBJ, P,E=3
```

FILE "EXDUNS"

File "EXDUNS" is a unique file and unique file name for each database user. Note the file recompiles the generated conversion program, readies the database files, readies the input source file, executes the generated conversion program and finally, saves the database. The following data is relevent to this example:

```
Database Name: DUNS
File Name Database Is Stored Under: DUNSDB
Database File Library Number and Password: 9299/3642
Source File Name: INFILE
Source File Device: TAPE
Source File Tape Number and Password: 8868/1648
Source File Characteristics: Foreign Tape; Density=800BPI;
                             Record Length = 1200 bytes
```

File "EXDUNS"

```
READPF, 9294, DUNSSRC
PUBLIC, PLF, I=DUNSSRC, B=DUNSOBJ, P,E=3
READPF, 9299, DUNSDB
S2KRS, DR, DUNSDB
REQUEST, INFILE, 8868/1648, RO, HI, B, 100.
DUNSOBJ
S2KRS, DS, DUNSDB
SAVEPF, 9299, 3642, DUNSDB
```

REFERENCES

[1] BAKKOM, David E.,
"Implementation of a Prototype Generalized File Translator", Honeywell Information Systems, Inc., Proc. 1975 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Cal., 1975, pp. 99-110.

[2] BUNEMAN, Peter O., et al.,
"ASAP to REL: Efficient Relational Data Bases from Very Large Files", University of Pennsylvania, Naval Research Technical Report NR-049-474, January 1975.

[3] CODS User Guide,
Systems Development Corporation, November 14, 1975.

[4] DIJKSTRA, E.W.,
"The Humble Programmer", Communications of the ACM, Vol. 15, No. 10, pp. 859-866, October 1972.

[5] FLOYD, R.W.,
"Assigning Meanings to Programs", Procedures of American Mathamatical Society Symposium. Applied Mathematics, Vol. 19, pp. 19-31, 1967.

[6] FRY, James P., et al.,
"A Developmental Model for Data Translation", Proc. 1972 SIGFIDET Workshop on Data Description, Access and Control, Denver, Colo., pp. 77-105, 1972.

[7] FRY, James P., et al.,
"An Approach to Stored Data Definition and Translation", University of Michigan, Air Force Office of Scientific Research Report #72-2219, December 1972.

[8] HOARE, C.A.R.,
"An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol. 12, No. 10, pp. 576-583, October 1969.

[9] JENSEN, K., and WIRTH, N.,
PASCAL User Manual Report, Springer-Verlag, New York, 1974.

[10] KOEHR, G.J., et. al.,
Data Management Systems Catalog, The Mitre Corporation, Bedford, Mass., January 1973.

[11] MERTEN, Alan G., et.al.,
"A Data Description Language Approach to File Translation", Data Translation Project, Proceedings ACM SIGMOD Workshop on Data Description, Access and Control , Ann Arbor, Michigan, pp. 191-205, May 1974.

[12] MERTEN, Alan G.,
"A Theoretical Analysis on Data Definition and Translation", Air Force Office of Scientific Research Report #76-0556, December 1976.

[13] NAVATHE, Shamkant B., et. al.,
"Restructuring for Large Databases: Three Levels of Abstractions", ACM Transactions on Database Systems, Vol. 1, No. 2, pp. 138-158, June 1976.

[14] PARNAS, D.L.,
"On the Criteria To Be Used In Decomposing Systems Into Modules", Communications of the ACM, Vol. 15, No. 12, pp. 146-151, December 1972.

[15] RAMERIZ, J.A., et. al.,
Automatic Generation of Data Conversion Programs Using a Data Description Language, Vols I, II, University of Pennsylvania, Philadelphia, Pa., May 1973.

[16] RAMERIZ, J.A.,
"Automatic Generation of Conversion Programs Using a Data Description Language (DDL)", Ph.D. Dissertation, University of Pennsylvania, 1793.

[17] ROOT, David J.,
"Converting from Rectangular to Relational Data Bases", University of Pennsylvania, Office of Naval Research Report #NR-049-272, September 1976.

[18] SHOSHANI, A.,
"A Logical-Level Approach To Data Base Conversion", Systems Development Corporation, Proceedings 1975 ACM SIGMOD Conference on Mangement of Data, San Jose, Cal., pp. 112-122, 1975.

[19] SHU, N.C., et. al.,
"EXPRESS: A Data EXtraction, Processing and REStructuring System", ACM Transactions on Database Systems, Vol. 2, No. 2, pp. 134-174, June 1977.

[20] WIRTH, N.,
"Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No 4, pp. 221-227, April 1971.

[21] YEH, R.T., and BASU, S.K.,
"Strong Verification of Programs", IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, pp. 339-346, September 1975.

VITA

Jonathan Lee Stevens was born in Roswell, New Mexico on July 24, 1948, the son of Jack D. and Yvonne K. Stevens. After graduating from Lakenheath High School, Lakenheath England, he attended the University of Washington, Seattle Washington, for one year. He then received a Presidential appointment to the United States Air Force Academy, entering in June 1967. He graduated with the degree of Bachelor of Science in Computer Science and the rank of 2nd Lieutenant in the United States Air Force. From August 1971 to October 1973 he was assigned with the 4629th Support SAGE Squadron as a Computer Programmer, and was promoted to 1st Lieutenant. From October 1973 to August 1977 he worked as a Systems Analyst at the Military Personnel Center and was promoted to Captain. In August 1977 he entered the Graduate School of the University of Texas.

Permanent Address: 9117-189th Place, S.W.
Edmunds, Washington, 98020

END

DATE
FILMED

11-81

DTIC