The development of an interactive codasyl database loader.

Watt, Glenn Donald
1979

Find more at https://preserve.lib.lehigh.edu/

This document is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
THE DEVELOPMENT OF AN INTERACTIVE CODASYL DATA-BASE LOADER

by
Glenn Donald Watt, Jr.

A Thesis
Presented to the Graduate Committee of Lehigh University in Candidacy for the Degree of Master of Science in Computer Science

Lehigh University 1979
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

March 5, 1979
Date

Professor in Charge

Chairman of Department
ACKNOWLEDGMENTS

I would like to thank The Northampton County Area Community College for the use of their computer during the formatting of this paper. I would also like to express my appreciation to my parents who were a constant source of inspiration during my entire period of graduate study. A special word of thanks is due my lovely wife Janine, who not only put up with my relentless drive to finish this paper, but also helped in drawing the figures contained herein. One final acknowledgment is due for the person that gave me the intellectual capacity to complete this paper. I am of course referring to the all mighty, GOD.
Abstract

In an age of increasing Data-Base use more and more computer installations are converting traditional file systems to a Data-Base system. The conversion procedure on all CODASYL Data-Base systems is performed by writing one or more application programs to load the base. To revivify the transformation process an Interactive Loading System (ILS) was developed. This system provides the user with a conversational method of generating a Data-Base DMCL, DDL, and the resulting loading program. Included in the package are help routines and several editors to facilitate flexibility and ease of use.

The system is divided into subroutine DMCL, subroutine DDL, and subroutine Loader as well as an executor program. Each subroutine contains processors which generate the command statements needed. Each processor that accepts data checks the data for validity in the current response mode. After each subroutine is finished program control returns to the executor. When the ILS is finished a computer Data-Base schema, and loading program are produced.

After the ILS was completed it was compared to a manual system to evaluate the performance. During these evaluations the ILS proved to be as good, or in most cases, better than the equivalent manual system. A typical example is that of creation time for the DMCL, DDL, and loading program. A 70% reduction in creation time was realized through the use of
the ILS. Similar reductions in other areas were also noted. During a comparison of: ease of use and user support, the ILS was found to be easier to use and required almost no training to operate. In general the ILS is an easy to use, yet economical Data-Base Loading System.
INTRODUCTION

In an environment of ever increasing conversion from traditional files to Data-Base files, the computer industry is still generating Data-Base loaders using slow, time consuming methods. "To load the Data-Base, [the user] must write one or more application programs that store the information in the Data-Base."\(^1\) This type of loading can be attributed to the fact that much emphasis has been placed on designing and optimizing Data-Base schemas with little or no attention to ameliorating the loading procedure. The consensus seems to be that every installation should write a specific loading program for each Data-Base implemented. This view can be attributed to several factors that surround the design and eventual loading of a Data-Base. Almost all Data-Bases start out as a canonical schema derived from a number of end users views of data. These users views are in the form of canonical records which, when put together, form subschemas. By combining and minimizing the canonical records, the schema is created. The combining and minimizing process yields headaches for Data-Base loaders since traditional file records might be eliminated, concatenated, reduced or remain the same to fit the Data-Base design. To compound the problem, several files could be combined to create new record types that were nonexistent in the tradi-

tional file system. A traditional file record type is shown in Fig 1.1 with the data stored in a two dimensional array.

Traditional Personal Record

<table>
<thead>
<tr>
<th>Skill</th>
<th>Training</th>
<th>Wage</th>
<th>Emp-No</th>
</tr>
</thead>
</table>

Traditional Payroll Record

<table>
<thead>
<tr>
<th>Emp-No</th>
<th>Gross</th>
<th>Net</th>
<th>Name</th>
<th>Street</th>
<th>City</th>
<th>FIT</th>
<th>FICA</th>
</tr>
</thead>
</table>

Fig. 1.1

After going through the schema design process the records would appear as in Fig 1.2.

Data Base Employee Reference Record

<table>
<thead>
<tr>
<th>Emp-No</th>
<th>Skills</th>
<th>Training</th>
<th>Gross</th>
<th>Net</th>
<th>FIT</th>
<th>FICA</th>
</tr>
</thead>
</table>

Data-Base Address Record

<table>
<thead>
<tr>
<th>Name</th>
<th>Street</th>
<th>City</th>
</tr>
</thead>
</table>

Fig 1.2

The record would then be compiled from the present record in the traditional file structure. Herein lies the problem. Each Data-Base record must be glued together from bits and pieces of the traditional files. The piecing procedure usually can be broken down into three major types:

1) Data base records are created from pieces of a single corresponding traditional file. *(The Good)*

2) Data Base records are created from several corresponding traditional files. *(The Bad)*
3) Data Base records are created from the concatenation of several fields of one or more corresponding traditional files. *(The Ugly)*

In *"The Good"* situation the loader needs to consider what fields of a traditional record have a one-to-one correspondence with some Data-Base record. Next the loader must consider the reduction of data redundancy inherent in traditional files. Finally the loader must create the appropriate links according to the Data Description Language (DDL) design. This could be elementary, if all the linking is accomplished automatically by the DDL; or complex, if the linking is left to the Data-Base loader.

*"The Bad"* has all of the problems of *"The Good"* with a few extras thrown in. In this situation the loader needs to examine not one but two or more files to find the appropriate fields of the traditional records from which to piece together the Data-Base record. After the record has been glued together, all of the linking problems of *"The Good"* remain to be solved.

*"The Ugly"* situation is one that is indeed ugly. Here the loader must consider one-to-one correspondences, of one or more traditional files to a particular Data-Base file. The concatenation of several fields from one or more files creates a new record type for the Data-Base file. This being the most difficult, it is also the most time consuming. The ugly situation can generate loaders of a very complex
This neglect in updating conversion procedures has undoubtedly taken its toll on many industries and institutions. It is conceivable that hundreds of thousands of man hours are spent in writing individual Data-Base loaders. The following thesis is based on the principle that an interactive Data-Base loader would elevate this portion of Data-Base design to a state-of-the-art technology.
METHOD OF SOLUTION

To test this concept an interactive loader was developed on a DEC-20 computer with TOPS-20 V1B,V2 operating system that supported a CODASYL based Data-Base Management System (DBMS). The system was structured in such a way as would allow further development at a later date in time. Any such additions could easily be integrated into the present system.

The DBMS provides three languages each performing a separate function.

Data Description Language (DDL). The DDL allows for the description of the overall mapping, both logical and physical, of the entire Data-Base. There are two types of DDLs, schema and sub-schema. The sub-schema allows for the description of a specific subset of the Data-Base.

Device Media Control Language (DMCL). The DMCL is used to define the physical storage space on storage devices used to record the Data-Base.

Data Manipulation Language (DML). The DML is not a stand alone language but instead is an extension of a host language like COBOL or FORTRAN. It is thru the DML that an interface is created between the traditional files, loader, and the Data-Base. The DML can best be thought of as the part of the host language that handles all Data-Base updating and retrievals.

An interactive approach was taken because of the easy of use by both novices and veterans alike. It seemed the

2. DEC-20 is a registered trademark of Digital Equipment Corporation.
best way to bridge the gap between ease of program generation and strict program guidelines. Interaction also allows for instantaneous error checking and editing as well as speeding data entry. Throughout the system brief instructions precede each unique input situation. These instructions usually consist of what input is being sought as well as valid input formats with a list of applicable codes wherever necessary.

The entire system was written in ANSI COBOL\(^3\) and produces a functional DMCL, DDL and loader program. Cobol was selected for two important reasons. First the language had to be compatible with the DBMS, leaving FORTRAN or COBOL; and secondly, the language had to be one normally found in a Data-Base environment. Since most Data-Base installations are business oriented, using COBOL application programs, the choice was obvious.

The method employed in solving the software problem was one of breaking it down into three sub-systems:

1) The DMCL generator
2) The DDL generator
3) The Loader Program generator

Each sub-system is represented by a subroutine of the same name with one executor program. The choice to break the software down into these component parts stemmed from both the scope of this project and the need to structure the system in such a way as to allow for its expansion at a later date. The executor program, which maintained control over what subroutines were being used, provided a common ground.

---
thru which to pass needed data. The executor provides the foundation on which expansion can take place in the future. The DMCL subroutine could stand alone since it has no information requirements and transmits nothing to the other components, but the DDL subroutine and loader generator subroutine do not fall into this category. Because the loader generator needs to know how the DDL is set up, subroutine DDL and the loader generator have a close relationship. It became clearly apparent that the most economical way to provide the loader generator with the information requirements needed was to incorporate the interactive design of the DMCL and DDL into the entire system. Those information requirements included the Schema Name, Record Names and Descriptions as well as duplicate conditions, allowed or not allowed, for each record and set member.

The schema name was stored in a variable "SCHEMA-NAME" and was transferred to the loader generator for processing. Record names, descriptions, duplicates conditions as well as set memberships were stored quite differently. They were placed in arrays entitled "RECORD-ARRAY", "DUP-COND-ARRAY", and "I-MODE" respectfully. This was necessary to allow for the threading of related data descriptions and duplicate conditions, for the natural flow of the DDL development does not permit sequential storage of these parameters. The array structure also allows for error searching in later parts of the DDL design. Thus the record names and descriptions are stored in a two dimensional array to support the threading process. The array allocates enough storage for fifty records with fifty fields each. The record name occupied the first column of each row of storage, while each field of a
record description occupies the remaining columns. The array
forms the thread from record name to field(1), field(2),
field(3), ..., field(n). To further the stringing process,
another one dimensional array was allocated for the storage
of the duplicate condition. A one-to-one correspondence
between the record name and its duplicate condition is real-
ized thru this array. To complete the threading process, a
third one-dimensional array was sanctioned to accommodate
the method of attaching member records to sets. If a record
is listed as a "Mandatory Automatic" member of a set, the
DBMS will take care of the linking itself. However, any
other type of membership requires the loader program to per-
form the linking. Therefore the information must be
transmitted to the loader generator so that the appropriate
steps can be taken.

Four output record types were used for the actual writ-
ing of information on the output file. They range from nine
characters to seventy two characters in length, incremented
in spaces of nine, eighteen, thirty two, and seventy two.
The inclusion of these record types was needed to prevent
wasted storage and to increase the efficiency of the entire
system. Clearly if an eighteen character record is being
written as a seventy two character record, fifty four spaces
are being wasted. In a typical DDL 5400 spaces would be
wasted per 100 lines, or the equivalent of 300 eighteen
character records. That's enough space to contain a small
DDL by itself. However, the total storage required for
enough record types to break this size down even further
would not be efficient.
The output files created from this system are "SCMFILDDL", and "DBLOADCBL". The first file, generated by subroutines DMCL and DDL, represents the software configuration of a user's schema. This file would be compiled into the working Data-Base which would require loading from the traditional data files. The second output file consists of the loader program that is associated with the DMCL and DDL described in the first part. After the execution of the loader program, the entire Data-Base should be loaded.

To further the emendation of the system, an internal editing package is included to allow for the correction of some errors. A one dimensional array capable of handling one hundred lines of text, each seventy two characters long, was allocated for this purpose. The editor allows the user to take a snapshot view of what he has entered, change any number of entire lines, review the change, and upon approval transfer its contents to one of the threaded arrays. The editor recognizes Cobol record descriptions and automatically adjusts the margins accordingly. Beyond that the editor makes sure that a level number, variable name, and variable description are included for each field entry. After the editor relinquishes its contents, the arrays are cleared allowing for the reutilization of the allocated memory.

Another user aid incorporated into the system is help routines. These routines provide instructions on the use of an area in question simply by responding with an "H" after any terminal interrupt is encountered. After the instructions are displayed at the terminal, control is transferred back to the point just prior to the initialization of the help routine, thus allowing another response. If an
unrecognizable response is entered, the question is repeated until a valid response is rendered. This facility almost entirely eliminates the need to carry a reference manual during the system execution.

After the completion of the software, a comparison was run to evaluate how and in what ways performance was improved over an equivalent manual system. The evaluation was divided into two parts, tangible and intangible benefits. A direct comparison of creation time, cost, and manpower requirements fell under the tangible benefits, while user convenience and ease of use fell under the intangible. In addition, CPU utilization and other statistics were compiled to evaluate the overall efficiency and response time of the system.
DMCL GENERATOR

The DMCL can be broken down into two sections. The environment- section deals with the specifying of parameters of the Data-Base schema in general; and the area-section, which assigns a particular area to a file while describing the physical characteristics of that file. In developing the interactive DMCL generator several items had to be taken into consideration for the environment-section. They include several possible image, note, and intercept command entries. To facilitate ease of command entry, all possibilities were coded to allow the user to simply enter a one digit code that corresponds to the desired command. After making a selection, the appropriate command is moved into a variable and then written upon the output file.

The first command statement processor in the DMCL generator handles the image statement. This command specifies whether the Data-Base Control System (DBCS) or the Run-Unit defines the smallest unit of recovery for the Data-Base. There are two possible commands to do this: "IMAGES IN ORDER BY COMMAND" and "IMAGES NOT IN ORDER BY COMMAND". These two possibilities were coded as 1 and 2 respectfully. The system displays the request for the user's desired image clause, with the two possibilities for their codes; and then waits for input. If an invalid entry is made, control passes back to the beginning of this processor; otherwise, the command statement is written into the output file and control passes to the NOTE/INTERCEPT processor. The following two command statements, Note and Intercept, have seven possible combinations each: Note and/or Intercept with exceptions for All, Bind, Call, System, Update, Unanticipated or No. In order to

-13-
minimize redundancy, each of the six suffixes as well as the last possibility are coded as 1 thru 7 respectfully and are displayed only one time during system execution. The Note/Intercept statements specify the exceptions that the DBCS will intercept or note during a Run-Unit. Each statement is optional and the user can choose to intercept one type of exception while noting another. A request is made to enter the desired notes clause as the seven choices are printed. After the selection is made, another request for the desired intercept clause is given, but the user is asked to refer to the previous codes. If an invalid entry is made, control is returned to the beginning of the processor; otherwise, the note command is assembled and written to the output file. The system then moves on to handle the journaling specifications.

Since the journal command is used to specify the file specifications during this process, journaling may not be needed if such records are not desired. This system, for the unwanted situation, provides all necessary revisions, as will be seen later. Assuming the journal is desired, the DMCL generator asks for the number of transactions that the user wants in the journal at any one time. The generator then assigns the journal to a file entitled "JRN1" and establishes the size of the journal according to the number of transactions requested. The number of transactions can be any unsigned, positive, decimal integer. After the integer has been entered and the system has generated the journaling commands, they are written on the output file and control is passed to the records per page processor.
The last command in the DMCL environment section is the "Records Per Page" command. This statement specifies the maximum number of records that can be stored on a page for all areas in the schema. The number of records can be any integer between 2 and 511 inclusive. If an entry outside this range is placed in the DMCL generator, it will display the entry and ask for the data again. If the user specifies the Record per Page at this point in the system and again in the area section, the later entry will override the environment section data.

The area-section of the DMCL starts with the assigning of a schema area to a particular file. To accomplish this, the system asks for an "Area Name" and then assigns that area to file 001. The assign command statement is then stored in the editing buffer. The area name is also stored in an area name array for later use in threading the information requirements of the subroutine loader. If a period is entered instead of as area name control passes back to the executor. A count of the number of area names is also kept for later use by subroutine DDL.

During the rest of the DMCL generator, operation control passes from one processor to the next, but will eventually return to the assign processor. During the cycling process, areas are assigned to File001, File002, File003, ... FileNNN.

Next in line for processing are the number of records per page, for an individual area. Once again the range spans 2 thru 511 with any other data entry cited and the request repeated. As with the first version in the environment section, the option of skipping is provided for, and if the
user has already specified the records per page, it will be superseded by this entry.

At this point control passes to one of two statement processors. If a journal was specified in the environment section, the system will begin assembling the back up command statement. The "Back Up" command has two forms: Back Up Before Images and After Images. The two possibilities were coded as 1 and 2 respectfully. The back up command specifies the type of images that will be written into the journal during an updating Run-Unit. This statement is optional. If an "N" for none is desired, the control is advanced to the buffer count processor. Otherwise the command is assembled into the form of "BACKUP before IMAGES". The command is then stored in the editing buffer and control passed to the buffer count processor.

In the case where journaling is not desired, the DMCL generator will automatically transfer control to the Buffer Count processor.

The Buffer Count statement specifies the amount of buffer memory to be allocated for an area of the Data-Base. The count can be any positive, unsigned, decimal integer of value equal to or greater than 2. The system allows for the absence of a buffer count, and it also checks to be sure of a valid entry. The buffer count processor displays a request for the buffer count. After checking the data, it begins to assemble the statement. The command statement is assembled as "BUFFER integer-1" and is stored in the edit buffer. If the buffer count is not specified, the Data-Base automatically assumes one of three buffers.
Control now passes to the part of the system that handles the number of "calc" chain headers associated with each page of an area. A question is displayed asking for the number of "calc" chain headers desired. In this processor, valid data entries include any unsigned, positive, decimal integer equal to or greater than 0. As in previous command processors, any invalid response will cause control to return to the beginning of the process involved. After the entry is made, it is pieced together to form a command statement of the form "CALC integer-1 RPP". This statement is then stored in the editing buffer. If desired this statement can be skipped and control passed to the next processor. If no "calc" chains are specified, the DBMS assumes one chain.

The succeeding process is one of assigning the first page and the last page, of an area, in the Data-Base. The range of page numbers of an area cannot overlap the range of page numbers of another area in the Data-Base. To accommodate, this situation the processor always remembers the last page of the previous area and requires that the first page of the present area be at least one digit higher in value. As in the other entries, the only valid format for data entry is an unsigned, positive, decimal number. In this case however, there is no default for the first page and last page statements; therefore, they must be specified. The system asks for the first page, checks the response for validity, and then assembles a command statement: "FIRST PAGE integer-1". The question of the last page number is then posed. After the data is entered and validated, the processor again puts together a command of the form "LAST
PAGE integer-2". These two command statements are then stored in the editing buffer. Control now passes to the page size processor.

Page size refers to the amount of data that can be stored on any one page of an area, as well as the size of that area's buffer. This command statement is not optional and must be an unsigned, positive, decimal integer of value less than 4096. The system prints a request for the data and upon receiving it forms the following command statement: "PAGE SIZE integer-1 WORDS". The statement is then stored in the editing buffer and control moves on to the range description.

The last command processor in subroutine DMCL is the range description. This is an optional command and the system will allow the user to skip this entire processor if desired; in which case, control is passed to the DMCL editor. If a range description is desired, however, a set of instructions are displayed for the user. Because of the complexity of data entry for this processor, a brief set of guidelines are considered necessary. The range command specifies the location in an area where a particular record can be stored in terms of page numbers. Therefore, the record-name, starting page number, and ending page number must be known to assemble this statement. The system asks for each of these items in that order. After receiving all the information the processor strings these items together to form a command string of the form "RANGE OF record-name IS PAGE integer-1 TO PAGE integer-2". Since an area contains more than one record, after storing this statement in the editor's buffer, control is returned to the beginning of
this processor, for the range description of another record. If a period is entered instead of a record name control is passed to the DMCL editor.

When the DMCL editor is invoked by the system the user is asked if a display of the buffer contents is desired. If no display is preferred the buffer is written, line for line, into the output file and then cleared. On the other hand, if the user requests a display, the entire buffer contents are displayed with a corresponding line number for each line. The items displayed are the assign command statements thru the range descriptions. If the user wishes to change any line he simply enters the line number of the erroneous line, inserts a comma, makes his correction and then the line is updated. After making all the desired changes, the user simply enters a period to signal the end of editing and control returns to the system. At this point the user is asked if a display is preferred again, and the process repeats itself. After the editing buffer is written into the output file, control passes to the assign command processor.
The DDL generator can be subdivided into four main sections: the schema section, area section, record section, and the set section. When constructing the DDL for a particular schema, the order of assemblage must follow these guidelines:

1) The schema name must be the first entry.
2) An area must be described before any record descriptions within that area can be entered.
3) All record descriptions must precede any set of which it is a member or owner.

Therefore the first command processor is the schema name. The system asks for the name the user wishes to call the schema and then checks to be sure it contains no more than six characters. If the entry is valid, the name is stored in a register for later transference to the subroutine loader. After this, the processor assembles the statement: "SCHEMA NAME IS schema-name". The command is then written into the output file. Control now passes to the next section of the DDL generator, the area section.

The first area section command statement processor is the area name processor. Initially this processor retrieves the number of areas entered in subroutine DMCL. This is accomplished to make sure each area assigned in the DMCL will be described in the DDL. The system then automatically generates an area name command statement for each area identified in the DMCL. The general format is "AREA NAME IS area-name". After assembling the first area name, it is written into the output file, then control is passed to the privacy lock processor. This is done to allow the assigning
of such locks to each area of the Data-Base. When control returns to the area name processor, the next area name is processed as the one before it. This procedure continues until the last area name has been specified. At this point, control passes to the record section for further processing.

The privacy locks specify how an area may be updated or information retrieved. There are four possible locks: Exclusive Update, Exclusive Retrieval, Protected Update, and Protected Retrieval. The user could also decide to have no privacy locks at all. To handle these choices, the processor was developed in two parts. The first division prints a request to enter the update lock; Exclusive = 1, Protected = 2, and None = N, for the present area. The second division is then invoked and a similar question asked about the retrieval locks. After both choices have been made, the lock commands are assembled and written upon the output file. The processor continues to do this for each of the areas entered.

The record section has several command processors that are used to describe the records in the Data-Base. There is a great deal of interaction between these processors since there are many paths that the control could follow. The record section starts with the record name processor and ends with the DDL editing processor.

The record name specifies a generic name for all occurrences of the record type in the Data-Base. The system displays a request to enter a record name and then waits for the data to be recorded. The record name entered is first stored in two arrays, "Recordo" and "Recorda". The first array stores the record name for later use in identifying
input errors, while the second array forms the first entry of the threaded array discussed earlier. After completing those tasks, a record name command statement is strung together and stored in the editing buffer. The command is of the form "RECORD NAME IS record-name".

Control now passes to the location processor. The location clause is used to define how a record is physically stored in the Data-Base as well as to which sets it should be associated when the set occurrence selection is, "Location Mode of Owner". There are three possible location modes: direct, calc and via. Each of these possibilities has a different set of input and output guidelines with different processors. The location switch simply diverts the program control to the appropriate processor. To do this, the three choices were coded as 1,2,3 respectfully. They are displayed along with a request to enter the location mode. The entry is checked and control passed to the appropriate processor. If an erroneous entry is made the question is repeated.

The direct location option requires an identifier which causes the DBCS to store the record on the same page as the record that is current of area, when the identifier is 0; otherwise, according to the page number of the identifier. To accomplish this task, the system displays a request to enter the identifier desired. The location mode command statement is then assembled in the form of: "LOCATION MODE DIRECT identifier". The command is then stored in the editing buffer and the type of location mode stored in part of the threaded array for subroutine Loader. Control now passes on to the "Within" command statement processor.
Location mode calc requires the input of data names to be used in directing the DBCS on where physically to store the record. It also needs to know whether to allow duplicates or not. To accommodate these information requirements, the processor displays a request for the user to enter the data names, separated by a comma. The names are then used to generate the command statement which is indented seven spaces to make it more readable. The statement looks like "LOCATION MODE CALC USING data-name-1, data-name-2, ..., data-name-n". This command is stored in the editing buffer while the system displays the question "Duplicates Allowed ?". If the answer is negative, control moves on to the No Duplicates processor, otherwise the type of location mode is stored in the current record location array to further the threading process necessary for subroutine Loader. Control then passes to the "Within" command processor.

The duplicates command checks for a duplicate record within the Data-Base each time an attempt is made to store a new record. If a duplicate exists, the DBCS returns an exception code that can be identified during the actual loading of the Data-Base. When invoked this processor generates the command statement "DUPLICATES NOT ALLOWED" which is stored in the editing buffer. The statement is indented seven spaces to conform to the surrounding commands. After its completion, control is returned to the Location Calc processor.

The final type of location, Via, requires a set name, with which the record is associated. With a set name, the DBCS stores the current record, physically as closely as possible to the record it will most likely be "NEXT OF"
after the DBCS applies the order of the set type. To meet these information needs, the system displays a request for the set name to be used. After the data is entered, the command statement is assembled. It is indented seven spaces for ease of reading and then deposited in the editing buffer. The command is "LOCATION MODE VIA set-name". The type of location is saved in the current record location array to continue the threading necessary for subroutine Loader. After its completion, program control passes to the Within command processor.

The "Within" command tells the DBCS what areas this record type can occur in. To do so, the area names must be entered. The system displays a request for the area names and waits for input. When the area name is input, the processor begins searching the area name array established in subroutine DMCL. If the area name entered is not found, an error message is printed and control returns to the beginning of the processor. Otherwise, the processor asks if this record is contained within another area. If so, control is shifted to a subprocessor that continues to ask for area names until a period is entered. It also checks each entry, as did the parent processor. When all the area names are entered, the command statement is assembled, indented seven spaces, and stored in the editing buffer.

Control is now handed over to the AREA-ID processor. This command allows the user to assign an identifier. The DML can determine in what area to store a record during a command. The system displays a request for such an identifier or "N" for none desired. If an "N" is entered, control shifts to the Schema-Data processor; otherwise a command
statement of the form "Within" area-name AREA ID IS identifier" is generated. This command is indented seven spaces and then stored in the editing buffer. After saving the command, control is transferred to the Schema-Data processor.

The Schema-Data processor is one of the most sophisticated processors in the entire system. The schema data entry allows a data item or aggregate to be defined for the current record. To do this the processor must allow for an almost infinite number of data description possibilities, while at the same time making sure they conform to all the guidelines. Included in this task is the job of insuring the inclusion of all required information.

An interactive approach is of primary importance to facilitate the implementation of this processor. The system first displays a brief set of instructions on how the data should be entered. The general format is level,fieldname,COBOL description. This format adheres to the main guideline since the picture string must conform to the rules of ANSI COBOL. After the data is entered, a check is made to be sure that the three elements are present. If not, an error message is printed and control is returned to the input stage of the processor; otherwise, control is shifted to a subprocessor. The subprocessor unstrings the data into three registers that correspond to the component parts of the data entry. The second part is then stored in the threaded array for the current record. Meanwhile the processor places each element into a readable format. There is an indentation of seven spaces and then the two digit level number is followed by a space and the fieldname. The COBOL description of the field name starts in column 41 leaving
at least one space between the largest possible fieldname and the description. The field description is then stored in the editing buffer and control returns to the parent processor. Upon return to the parent, control is shifted to the input stage once again. If a period is entered instead of actual data, control passes to the last portion of the record section, the DDL Editor. If a period is not entered the above process is repeated.

When the DDL Editor is invoked, the user is asked if a display of the buffer contents is desired. If none is desired, the contents are written into the output file and the editing memory is cleared. If the user wishes, the contents are displayed in their entirety. The user can then change any of the items displayed which include the record name thru its field descriptions. As in the DMCL editor, the user enters the line number of the erroneous line, a comma, and then makes the correction. One major difference arises with the DDL editor, however. Because of the manner in which field names and descriptions must be stored and assembled, special care has to be taken to make this editor slightly more intelligent. In doing so, one requirement was the placement of a semi-colon imposed between the level, field, and picture clause. With this, the editor is able to identify these special cases and update them accurately. The editor also checks for indented commands and will of course indent the corrected command. When all the corrections are made, the user simply enters a period and then control returns to the beginning of the editor, when he is asked if a display is again desired. At the completion of the editing package control is transferred to the Record Name processor.
The last section of the DDL generator is the set section. This section describes a set in the schema. The description is used by DBCS to store and find occurrences of records within specified sets.

The first statement processor is the Set Name processor. It specifies a generic name for all occurrences of the set type within the Data-Base. The processor displays a request for the desired set name. After the name is input, the command statement is assembled in the form of "SET NAME IS set-name". If a period is entered, control is transferred to the Sub-Schema processor; otherwise, the statement is stored in the editing buffer and control shifts to the MODE processor.

Mode clauses are used to specify how a set is to be manipulated. Therefore, the processor is very straightforward in its approach. Since there are only two possible modes: Linked to Prior, or Not Linked to Prior, they were coded as 1 and 2 respectfully. The user inputs his choice when asked to do so, to generate the appropriate command. The Mode clause is indented seven spaces and then stored in buffer memory allocated to the editor. Control now passes to the Order processor.

The Order command specifies the insertion point of a member record occurrence inside of a set occurrence which thereby defines the logical order of linkage. The insertion order can be first, which places the record immediately after the owner record of the set, or last, thus making a new record the last in a set. Other choices may be next/prior which simply insert records as they are stored in
the Data-Base. Always Next and Always Prior can be equivalent to the first two depending on the current record. Other possible order commands include: sorted by record name, Data-Base key, or simply sorted by duplicates. Since there are three possible duplicates conditions: first, last, and not allowed, the total number of possible order commands stands at nine. These options were crypted as 1 thru 9 as described previously and are displayed along with a request to enter the order clause for member records. The data entry is checked for reliability. If acceptable, the appropriate order command is assembled and stored in the editing buffer. Control then passes to the owner command processor.

The owner command specifies the owner record of a set or that the owner of the set is the Data-Base itself. To assemble this statement, the system requires the entry of the record name or the word 'System'. When such an entry is made, the processor searches the record array created in the Record name processor for the particular record name entered. When the record name is found, or the word System is entered, the statement processor assembles a command of the form "OWNER IS record-name". Should the record name not be found, an error message is printed and control transferred to the data entry point of the processor. After the command is built, it is placed in the editing buffer and control is handed over to the Membership command processor.

The membership command is used to specify how occurrences of a record type can be added or deleted from a particular set. Since each record type, that's a part of a set occurrence, could be added or removed differently; there must be one membership command per record type in the set.
Possible types of membership include Mandatory Automatic, Mandatory Manual, Optional Automatic, and Optional Manual. Mandatory/Optional specify whether or not a record can be deleted by means of a DML remove command. Since records are not deleted from a Data-Base during the initial loading, there is little importance associated with these options. Automatic/Manual refers to how a record can be added to various sets. If the manual branch is selected, records can only be added by use of the DML Insert command. In the other case, records are implanted automatically by the Data-Base system.

The Membership command processor initially asks the user to enter the member record name. As in the Owner processor, the record is checked to be sure of its existence. After verifying the record name, the processor displays the four possible insertion types, coded as 1 thru 4, and asks for the one desired to be applied towards the current record. Selection made, the processor strings the correct statement together and proceeds to deposit it in the editing buffer. Since the type of member insertion determines how the loader will inevitably add records to sets in the Data-Base, this information must be transmitted to subroutine Loader. The processor accomplishes this by storing the type of insertion mode in an array that is linked together with the previously mentioned threaded arrays. Control now passes to the Linkage processor.

The linkage command causes each record occurrence to possess a pointer to its owner. This command is optional and, if omitted the records are not linked directly to their owners. The processor asks the user if the Linked to Owner
phrase is desired. If so the command statement "LINKED TO OWNER" is indented seven spaces and then placed in the editing buffer memory. Control then passes to the Ascending/Descending statement processor, as if the linked phrase had not been desired.

Ascending/Descending refers to the way in which sorted sets are ordered. There are four possibilities that make up this command which include: Ascending, Descending, Ascending within a range, and Descending within a range. Beside ordering the set, this processor also specifies what should be done should a duplicate occur.

The four possibilities were coded as 1 thru 4 and are displayed along with a request to enter the sort order. In the event one of the first two choices is selected, the user is asked to enter the key data names to be sorted, and separated by commas. These data names are then strung into a command statement of the form "ASCENDING KEY IS data-name1, data-name2, ..., data-nameN". The statement is saved in the memory allocated to the editor. Control is handed over to the Duplicate subprocessor.

Had one of the latter two possibilities been selected, the user would still have been asked for the key data names; but the general command would have appeared as "ASCENDING RANGE KEY IS data-name1, ..., data-nameN". The word Range specifies that all keys are also treated as range keys. Once again the command would be stored and control shifted to the subprocessor.

The three possible duplicate conditions first, last, and not allowed were encoded as 1 thru 3 respectfully. When
the user makes a choice the appropriate command is stored in the editor. Control then moves on to the last command processor in this section of subroutine DDL which is the Set Occurrence processor.

The set occurrence command is used to define the guidelines by which the correct occurrence of a set is selected during the automatic insertion of a member record. This occurs during the execution of a DML Store command. This command processor allows for two types of selection commands, as well as the optional Using and Alias clauses. Current of Set and Location Mode of Owner, the two primary choices were coded as 1 and 2. The user simply enters his choice and the processor automatically assembles the command and stores it in the DDL editor.

The subprocessor allows for the entry of data names and aliases for these data names. It is very straightforward and simply asks for the desired data names separated by commas. The names are assembled into a clause of the form: "USING data-name1, ..., data-nameN". The subprocessor then asks for an alias clause if desired. The data is entered in the form of: data-name, then its alias. If a period is entered, this portion is skipped. The clauses are stored as part of the parent commands in the editing buffer. Control is then transferred to the DDL editor.

At this point the DDL editor is invoked again and, as in the first implementation, the user can look at and change any part of the buffer. A detailed discussion will be preempted since its operation was fully explained earlier. At its completion, program control returns to the Member processor.
When control is finally given to the Sub Schema processor, subroutine DDL is almost at an end. All user/system conversations are ended and the processor performs all the remaining tasks. Normally the Sub Schema is used to define subportions of the entire Data-Base; but since this system is only concerned with loading the base, there is no need for such refinement. In the sub schema, as in the schema, there are areas, records, sets and a sub schema name. These items must be defined in some manner, even if identical to the schema. This is the approach taken in the development of this processor. It simply assigns the name "MASTER" to the sub schema and then writes the sub schema name command into the output file. In much the same way, it writes the "AREA SECTION" command followed by "COPY ALL AREAS", which does exactly that. This process is repeated for the Record Section and Set Section. After the set section is completed, the last command "END-SCHEMA." is written into the output file and control is returned to the executor.
Loader Generator

The final subsystem invoked is of course subroutine loader. It is herein that all the data received thus far is compiled into a working Cobol loading program. As with the other two subroutines, Loader is conversational. It does have one variance from the other two, however. While the DMCL and DDL subroutines deposit their results to the Scmfilddl data file, Loader writes its output into the DBLOADCBL file.

This subroutine represents the most intricate of the three subroutines. Although Loader can only manipulate the "Good" and "Bad" cases described earlier, it approaches the aura of the day in computer systems development like the writing of a program hitherto performed by humans alone. The Loader generator is not a turn key routine, however, and it still relies on the other two subroutines as well as a considerable amount of human intervention.

In accordance with the principles of Cobol, the Loader generator can be divided into four main sections. The sections are: Identification Section, Environment Section, Data Section, and the Procedure Section. In addition to these four main sections, the generator has its own editor and help routines. When the executor initiates this subroutine, control is placed in the Identification Section.

The identification section is responsible for establishing all necessary commands for the identification division of a Cobol program. Included in this are the Division Name, Program Identification, and the Author of the program. Since all these items are constants, no user involvement is
required. The processor writes the command "Identification Division." into the output file and then transfers control to the Program ID processor.

The Program ID is the only required segment of the Identification Division. It must be written in the Cobol A margin and must end with a period. In order to meet these requirements and at the same time automate this procedure, the name "LOADER" is given to the program ID; therefore the following command is written into the output file while, control passes to the Author processor.

The author entry is normally used to indicate the programmer's name as its caption implies. Since the author of this thesis developed the system which actually writes the program, his name is supplied here. This statement also starts in margin A and appears as "AUTHOR GLENN D. WATT JR.". After the command is written into the output file, control passes to the next section, which is the Environment Section.

The Environment Section has a dual purpose. First it defines the type of computer being used to compile the program; the second function is to relate program files to specific computer input-output units. This system does not get involved with the first function, as it is not of primary concern, but does home-in on the second part of the Environment Division This part is called the Input-Output Section and it is handled thru the use of three command processors.

The first of these processors, the I-O processor simply writes the command, "INPUT-OUTPUT SECTION." into the output
file. It then relinquishes control to the File Control processor. File control is required whenever a file to be processed is associated with a computer input-output device. As in the first processor, it simply writes the appropriate command, "FILE-CONTROL." into the output file. After completing this task, control passes to the final processor in the Environment Section, the Select processor.

User interaction is reestablished with the Select command processor. The select command is used to assign a data file to a particular device. For example, if a traditional file system consisting of four data files were to be placed in a Data-Base, those files would each need to be assigned in the Data-Base Loading Program (DBLP). To accomplish this assigning, the data file name must be entered when the system asks for it. After the data file name has become input, the select command is partially assembled in the form of "SELECT data-file-name ASSIGN TO DSK". This basically is assigning the file to a disk and will then begin to further define the memory allocations for Cobol. This portion is indented seven spaces and then written into the output file. The processor then assembles the last half of the command, "RESERVE NO ALTERNATE AREAS PROCESSING MODE IS ASCII.". This part specifies how each file should be recorded sequentially, on the disk using the ASCII format. This part is indented as the first and written into the output file.

The processor is not yet complete however. The data file names are needed throughout the rest of subroutine Loader. Therefore, the names are stored in a one-dimensional array for easy access at a later point in time. Control then returns to the beginning of this processor to
accept the next data name. The looping procedure continues until a period is entered instead of a data name. When this occurs, control is transferred to the next section, the Data Section.

The Data Section has a number of functions among which are the definition of the computer memory for the program and the assigning of data names to all such memory areas. It indicates the required size of each storage area and describes the data to be stored therein. The data section processors also define the relationships between files, records, and field names. They indicate the grouping of certain data and the independence of other fields.

The first processor to gain control is the Data-Div/ File Section processor. Since Cobol requires a division name and section name, this processor provides just that. It first writes the command "DATA DIVISION", starting in margin A, into the output file. Next the processor generates the command "FILE SECTION". The statement is then indented seven spaces and also written into the output file. Control now passes to the File processor.

The File processor is used to assemble the FD command statement for the loading program. Since Cobol requires that each file mentioned in the Environment Division have a file description in the Data Division that agrees exactly in name, this processor performs the task automatically. It initially strings the first data file name from the file name array together with the appropriate syntax to form the last half of the command statement. The processor then assembles the front end syntax and appends the other half to it. The command is stored in the editing buffer in the form
"FD file-name LABEL RECORDS ARE STANDARD VALUE OF ID IS file-nameDAT.". Control then moves to the Record command statement processor.

The Record processor is used to define and allocate memory for the various record types within the file previously defined. The processor starts by displaying a request for the traditional record name that corresponds to the current data file, also listed. When the name is entered, the processor begins assembling the command statement. An "01" level number is prefixed and a period suffixed to the record name. The complete statement is then deposited in the editing buffer. Control is passed to the Field Description processor at the completion of the storage procedure.

Each record must have its fields described completely. Every data element must be defined as to nature: numeric, alphabetic, alphanumeric, and size. To accommodate the voluminous number of possibilities, the user is permitted to enter the required information in a manner similar to the DDL's Field Description processor. The general format is level number, comma, field-name, comma, and picture description. All entries are checked to assure the inclusion of all three items. Should an error be made, a message is printed and control backed up to the "field" data entry point of the processor. Otherwise, the processor unstrings the data into three parts and then reassembles them with margins separating the triad. The finished command statement is then loaded into the editor's memory. Control now backs up to the "field" data entry position in the processor for the entrance of another data element name and its description. Only when a period is entered does control escape from this
infinite loop. When the period is entered control is channeled to the Loader editor.

The editor in subroutine Loader is in many ways identical to its compatriot, the DDL editor. They both examine the line being replaced, and they format the correction in a similar manner. Unlike the DDL editor, the Loader editor does not require two types of data separators. This is due to the type of command statements that pass thru the editor. Only record descriptions and special insertion command statements are edited. Therefore the editor need only check for two kinds of input. At the completion of the editing cycle, control returns to either the Field Description processor or the Insert Condition subprocessor.

When control returns to the Field Description processor from the editor, the processor poses a question to the user. The question to be answered is whether or not the current file has another record type. If the answer is affirmative, control is rerouted to the beginning of the Record Name processor. In the negative, state control passes to one of two locations. If another file has been specified but its records not yet defined, control will be transferred to the File processor; otherwise control is relinquished to the final section of subroutine Loader, the Procedure Section.

The Procedure Section of subroutine Loader generates the COBOL code for the Procedure Division of a loading program. This section develops the procedural commands that activate the computer; in fact, this division may be said to be "the program", as most people think of it. Because of the nature of the Procedure Division, there are very few guidelines by which to operate automatically. Basically
only six rules apply to this division.

1) "The program segment is introduced by its name as follows:
PROCEDURE DIVISION.
This must be written at the A margin and followed immediately by a period."^4

2) "The principle subdivision of the procedure division is the paragraph. This is a program segment of at least one sentence, introduced by a name. This the paragraph name or procedure name, is written at the A margin and followed by a period and at least one space."^5

3) "Sentences exist within paragraphs. A sentence in COBOL, is one or more commands terminated by a period. Sentences must be written to the right of the B margin."^6

4) "Before a file may be used for either input or output purposes, it must be opened."^7

5) "All input and output files must be closed after processing has been completed, but before the programming is terminated."^8

6) "Because of its terminal effect the STOP RUN statement can only be used as a final statement in the sequence in which it appears; otherwise, the

---


5. Ibid

6. Ibid, p. 75.

7. Ibid, p. 78.

succeeding statement will never be executed.°9

After fulfilling these needs the procedure section must simulate the logic of a programmer and construct a working loader program. Since the loading of a Data-Base only requires the use of five DML commands, table 5.1, the near total automation of this section seemed likely.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND</td>
<td>(locate a record)</td>
</tr>
<tr>
<td>STORE</td>
<td>(a record into a Data-Base)</td>
</tr>
<tr>
<td>INSERT</td>
<td>(a record into a set)</td>
</tr>
<tr>
<td>OPEN</td>
<td>(part of the Data-Base to process)</td>
</tr>
<tr>
<td>CLOSE</td>
<td>(part of the Data-Base)</td>
</tr>
</tbody>
</table>

Table 5.1

Control is passed from the data section to the Procedure processor in this section. The processor assembles the statement that fulfills the first rule of the Cobol division. It then writes the command into the output file and turns the program control over to the A-Marker processor.

The A-Marker is nothing more than a paragraph label. It is placed at the beginning of the loader program for readability. The processor assembles this label as follows: "AMARK." and then writes it out to the output file. Control then passes to the Open processor.

With the initializing of this processor, the complex job of creativity begins. The processor starts off by writing the command, "OPEN ALL USAGE MODE IS EXCLUSIVE UPDATE", into the output file. This command will open the Data-Base for storing data. The processor now needs to open the traditional data files for input purposes alone. After

constructing the first half of the command, "OPEN INPUT", the processor begins retrieving the traditional file names that were stored earlier. The names are threaded together which form the files to be opened. The first half, field names, and the last part are then assembled and written to the output file.

At this point we can begin to code the specific statements that transfer the traditional data to the Data-Base. This is accomplished thru the Transfer processor. It initially writes a paragraph label, into the output file. Next it assembles a Cobol READ command statement for the first file and writes it out. The command appears as: "READ file-name AT END GO TO AMARK(n)". With this general read command the traditional file is read one record at a time and at the end, Loader Program Control (LPC) is transferred to the next paragraph. This generalized form allows for the reading of any file by simply using it at the appropriate time.

From here the processor displays the field names of the Data-Base record that applies to the traditional record in the hitherto described file. As each field name is displayed the user inputs the corresponding traditional field name. In this way the interface between traditional and Data-Base files are established. With each set of field names the following command is generated and written into the output file: "MOVE tfield TO dbfield.". When each of the Data-Base record fields are accounted for, the processor unthreads the Duplicates Condition, stored in subroutine DDL. If no duplicates are allowed for the current record type, control is transferred to a subprocessor for this condition.
In the subprocessor, two commands are added to the loader program being generated. The first command is a DDL Find command for the current record to be stored. Should a duplicate not be there, the DBCS would return an error code. Here is where the second command comes in. It tests for the error condition; and, if none is found, a duplicate exists. When this occurs, LPC is diverted around the store command for that instance.

After handling the duplicate condition, one way or the other, the transfer processor unthreads the set membership indicator. If the record is not an automatic member of each set, it is affiliated with; control passes to yet another subprocessor. This subprocessor allows the condition setting, for set insertion. The system displays a request for the entering of the specific Cobol code, and DDL instructions that will insert the record into the desired sets. The user can enter as many lines of coding as needed to accomplish this task. Since the automation of this portion extends beyond the scope of this thesis, the validation of the statements entered is left to the user. The subprocessor will however, store each response in the editing buffer until a period is entered. At this point, the editor is invoked and the user can review what was entered. After approval is given, the commands are written into the output file while control returns to the parent processor.

As control returns to the Transfer processor, a check is made to see if another file has yet to have its records described. If this is the case, the Mark Routine processor is invoked, which assembles a paragraph name. Each paragraph name is identical except for the last character, a number.
Thus the first paragraph is labeled "AMARK1", the second "AMARK2" and so on down the line. After writing the command into the output file, control returns to the Transfer processor.

When the last file has been described, control goes to the "GO TO" processor. This processor simply writes a Cobol 'GO TO' command into the output file which will cause the LPC to loop until the Data-Base is loaded.

At the completion of that processor, control passes to the End of File processor. During the execution of this processor, all the general housekeeping commands are written to finish off the loader program. The first of these commands to be placed at the end, is the last paragraph name of the Amark variety. The Amark paragraph name is needed to provide a place for the LPC to go, after the last traditional file has been exhausted.

Now the Data-Base files need to be closed to fulfill rule 5, stated earlier. This is accomplished through the DDL "CLOSE ALL" command, which closes every file in the Data-Base. The command is written into the loader program, leaving only the traditional files to be closed. Since there is no global close command in Cobol, each file, previously opened, must be closed by name. To do this the file array is accessed one final time as the names are assembled with the close command. The finished command statement is then added to the loader program.

At this point only one rule needs to be applied before the loader program can be called complete. The command is, of course, "STOP RUN". The processor writes this command
into the output file and relinquishes control to the operating system of the host computer. The loading program, as well as the DMCL and DDL of the Data-Base, are now complete.
ANALYSIS, SUMMARY and CONCLUSIONS

The "Loading System", as noted before, only handles the "Good" and "Bad" types of loading problems. Clearly such complex loading procedures, as the concatenation of two or more traditional fields to form a single Data-Base field, are not provided in subroutine Loader. Other, equally difficult problems are not addressed either. After due analysis an estimated 12 to 24 man-months would be required to develop the additional processors necessary to handle any loading problem.

The present system supports the major premise of the thesis. The verification comes from the evaluation criteria, set down earlier. In each of the evaluation areas this system either met or surpassed an equivalent manual system.

Under the area of tangible benefits there were three main points of comparison noted: Creation Time, Cost, and Manpower Requirements. The creation time was tested by supplying several individuals, both experienced and inexperienced, with a specific Data-Base schema, and associated traditional files. The Computer Data-Base and subsequent loader program had to be created and entered into the computer, using the traditional manual process. The development and programming process was timed from beginning to end. The procedure was then repeated using the ILS. The times varied between different Data-Base schemas, but remained standard between the manual and interactive systems. A typical test example took an entire 8 hour day to write and enter the DMCL, DDL, and loading program. When the same example was applied to the ILS, completion took approximately 2 hours. A 70% reduction in creation time was common.
The cost of operating this system, verses the cost of operating an editor, or similar procedure to create the DMCL, DDL and loading program is negligible. In some cases an editor is less expensive to operate while in others the ILS has considerably less overhead. Since the two methods are so close, neither is considered superior in terms of cost.

The manpower requirements for both the manual and the interactive system are the same on a single project basis. When many Data-Bases are considered concurrently the manpower requirements change drastically. While additional personnel are required, at the rate of approximately one per project under the old system; the interactive system only requires an increase of one per six projects. Clearly as the number of concurrent projects increase the benefits compound themselves.

Under the intangible benefits the interactive system really begins to shine. In terms of user convenience, the ILS provides a fool-proof method of DMCL, DDL and loader generation as well as an on line reference library for help. In addition, the system contains editors that allow the user to change anything that he, or the system, generates during specified portions of the design process. One drawback to the system does remain, however. At this point in time complex Data-Base structures can't be realized through the ILS. These structures can be entered using the manual method.

In terms of ease of use, the system always displays a brief set of instructions for the user. Along with the "Help" routines and data entry verification, the user need know nothing about the Loading System. A working knowledge
of the CODASYL DBMS and ANSI COBOL is required, as they are in the manual system.

To summarize the operation of the system consider the following example. An employee reference Data-Base is to be established with all it's data coming from two traditional files (Figure 6.1).

Traditional Personnel Record

| SKILL-T | TRAINING-T | EMP-NO-T |

Traditional Payroll Record

| EMP-NO-T | GROSS-T | NET-T | FIT-T | FICA-T |

Data-Base Employee Reference Record.

| EMP-NO | SKILL | TRAINING | GROSS | NET | FIT | FICA |

Figure 6.1

Since the Data-Base contains only one record type the DMCL GENERATOR might create the following Data-Base DMCL by using the interactive method described earlier.

IMAGES BY COMMAND.

NOTE ALL.

INTERCEPT BIND.

JOURNAL JRN1.

RECORDS-PER-PAGE 35.

ASSIGN EMPLOYEE-AREA TO FILE 01.

BACKUP BEFORE IMAGES
Control would then pass through the Executor and into the DDL generator. This generator might add the following to the DMCL to form the entire schema.

SCHEMA NAME IS EXAMPLE.

AREA NAME IS EMPLOYEE-AREA.
RECORD NAME IS EMPLOYEE-REFERENCE-RECORD
LOCATION MODE CALC USING EMP-NO
DUPLICATES NOT ALLOWED
WITHIN EMPLOYEE-AREA.

02 EMP-NO   PIC 9(5).
02 SKILL    PIC X(5).
02 TRAINING PIC X(5).
02 GROSS    PIC 999V99.
02 NET      PIC 999V99.
02 FIT      PIC 999V99.
02 FICA     PIC 99V99.

SET NAME IS EMPLOYEE-SET
MODE IS CHAIN LINKED TO PRIOR
ORDER IS SORTED DUPLICATES NOT ALLOWED
OWNER IS SYSTEM
MEMBER IS EMPLOYEE-REFERENCE-RECORD
SET SELECTION CURRENT.
SUB-SCHEMA-NAME IS MASTER.

AREA SECTION.
COPY ALL AREAS.

RECORD SECTION.
COPY ALL RECORDS.

SET SECTION.
COPY ALL SETS.

END-SCHEMA.

After the schema is completed control passes through the executor and into the Loader Generator. Using the interactive approach described in chapter 5 the first three divisions of the loading program are created. The loader for this example would appear as follows.

IDENTIFICATION DIVISION.
PROGRAM-ID. LOADER.
AUTHOR. GLENN D WATT JR.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT PERSONNEL-FILE ASSIGN TO DSK ASSIGN
NO ALTERNATE AREAS PROCESSING MODE IS ASCII.
SELECT PAYROLL-FILE ASSIGN TO DSK RESERVE
NO ALTERNATE AREAS PROCESSING MODE IS ASCII.

DATA DIVISION.

FILE SECTION.
FD PERSONNEL-FILE LABEL RECORDS ARE STANDARD
VALUE OF ID IS "PERFILDAT".
01 PERSONNEL-RECORD.
   02 SKILL-T     PIC X(5).
   02 TRAINING-T  PIC X(5).
   02 EMP-NO-T    PIC 9(5).

FD PAYROLL-FILE LABEL RECORDS ARE STANDARD
VALUE OF ID IS "PAYFILDAT".
01 PAYROLL-RECORD.
   02 EMP-NO-T    PIC 9(5).
   02 GROSS-T     PIC 999V99.
   02 NET-T       PIC 999V99.
   02 FIT-T       PIC 999V99.
   02 FICA-T      PIC 99V99.

The Loader generator, after creating the Identification, Environment and Data divisions of the COBOL loading program, begins the unthreading process. During this process the traditional fields are associated with the Data-Base fields (Figure 6.2).

![Diagram](image-url)
After each logical linkage is established the COBOL code is generated for the Procedure Division of the loading program. The code for this example is the following.

PROCEDURE DIVISION.
   AMARK.
       OPEN ALL USAGE-MODE EXCLUSIVE UPDATE.
       OPEN INPUT PERFIL,PAYFIL.
   AMARK1.
       READ PERFIL AT END GO TO AMARK2.
   AMARK2
       READ PAYFIL AT END GO TO E-O-F.

       MOVE EMP-NO-T TO EMP-NO-DB.
       MOVE SKILL-T TO SKILL-DB.
       MOVE TRAINING-T TO TRAINING-DB.
       MOVE GROSS-T TO GROSS-DB.
       MOVE NET-T to NET-DB.
       move FIT-T TO FIT-DB.
       MOVE FICA-T TO FICA-DB.

At this point one of the optional processors, the duplicates processor, is invoked since the schema of this example specified no duplicates allowed. The following code is therefore generated.

   FIND PERSONNEL-RECORD RECORD.
   IF ERROR-STATUS NOT 0326 GO TO SKIP1.
   STORE PERSONNEL-RECORD.
   SKIP1.
From here the rest of the program generation is a simple process. The end of the program would appear as follows.

E-O-F.
CLOSE ALL.
CLOSE PERFIL,PAYFIL.
STOP-RUN.

In conclusion the ILS provides a much easier and more economical way to develop a computer Data-Base and loading program. Several area are recognized for further development. These areas include:

1) The "Bad" loading situation.
2) Two levels of operation - one for beginners and one for the more experienced.
3) The development of slightly more intelligent editors that would allow the user to stop the system at any point and resume at a later date.
4) The capacity to self execute the finished schema and loading program.
LIST OF REFERENCES


EXECUTOR

SUBROUTINE DMCL

SUBROUTINE DDL

SUBROUTINE LOADER

Appendix A
VITA

Born on July 15, 1955, in Philadelphia Pa., I was the son of Mr. and Mrs. Glenn D. Watt. At the age of six I moved to Denver, Pa. and was enrolled in the Cocalico School District. During my high school years my horizons were broadened through extensive travel which included the continental United States as well as Canada, Mexico, Finland and the Soviet Union. I graduated from Cocalico High School in June 1973.

I immediately entered Kutztown State College as a Secondary Education Math Major. While at Kutztown I was involved in numerous activities including: President of the Inter Varsity Christian Fellowship, Assistant Director of a dormitory, and a member of the college Jazz Band. During my senior year I also enrolled in Air Force R.O.T.C. I graduated from Kutztown in May 1977.

I then embarked on a graduate program in Computer Science at Lehigh University. After my first year of graduate work was completed I received my commission to 2nd Lt. U.S. Air Force in May 1978.

I began my first computer related occupation as a Data-Base analyst for the Naval Air Development Center in Warminster Pa. After completing my project, with accommodations, I moved on to my present occupation; Adjunct Professor of Data Processing at Northampton County Area Community College, Bethlehem Pa.

I have also written software for the Texas Instruments' PPX-52 program which has been published in the Texas Instruments PPX-52 Program Library.