TABLE OF CONTENTS

I. ACKNOWLEDGEMENTS ....................................................... 1
II. ABSTRACT ............................................................................. 2
III. BACKGROUND AND RATIONALE ......................................... 3
IV. OVERVIEW
   A. INTRODUCTION ............................................................... 4
   B. ENVIRONMENT ............................................................... 6
   C. STRUCTURE CHART ......................................................... 7
V. FILE ORGANIZATION .......................................................... 8
VI. DETAILED PROGRAM DESCRIPTIONS ..................................... 14
   A. main.c ............................................................................... 15
   B. crform.c ............................................................................ 17
   C. shoform.c ........................................................................... 23
   D. prform.c ............................................................................ 30
   E. upform.c ............................................................................ 34
   F. helpfile.c .......................................................................... 39
VII. CONCLUSIONS .................................................................... 41
VIII. BIBLIOGRAPHY ............................................................... 42

APPENDIX A

Program listings ........................................................................ 43

USER'S GUIDE .......................................................................... 88
ACKNOWLEDGEMENTS

The cooperation of the entire staff of the Directorate for Management Information Systems (DMIS) of the Corpus Christi Army Depot (CCAD) is to be commended. My supervisors' cooperation was evident in their efforts to help me derive a project idea that would be worthy graduate work. Other cooperative efforts may have simply been made by those who gave up their sparse terminal time so that I could use a terminal.

More specifically, I would like to acknowledge four individuals. Don Miles, Team Leader for Information Resource Management, deserves much appreciation for his leadership, intuitive suggestions and most definitely for his encouraging words during periods of nervous gitters and self doubts.

Carter Ames, a fellow worker and graduate student, is highly commended for his knowledge of the "C" programming language, which he also taught himself just a short time before I tried to master it. His patience was steadfast through every consultation I had with him concerning the quirks of the language that were not divulged by the text.

I sincerely appreciate the efforts of Kay Christensen. She recently joined the group of co-op graduate students of DMIS. She most willingly helped me input this text on the VAX to practice her new skills at the "vi" editor.

Not the least of all, I want to thank Priscilla Milam, who was in the midst of work on her graduate project at the same time. She lent a sympathetic ear to me during many periods of frustration.
ABSTRACT

This project involved the design and implementation of a system of programs to automate the processing of Department of the Army forms. One function of the system is the design and definition of a form layout. Once a designer has answered a series of questions to define the layout of a particular form, the form description created is used to prompt users for actual data entry to that form. The data entered in the preceding entry process is formatted to print. When a form is inserted into a printer the data will print, properly aligned, on the form.

The system of programs was implemented and tested on a Vax 11/730 with UNIX Operating System version 4.1. The programs were coded in the "C" programming language.
BACKGROUND AND RATIONALE

The "forms manager" system was developed to satisfy the desire of the management divisions at Corpus Christi Army Depot (CCAD) to automate forms processing. The VAX Unix System was chosen as the system to perform this particular processing function because it had been established as the office automation system.

Before the development of the "forms manager" system, proper completion of the forms depended on the skills of the secretaries and their abilities to consistently type input information in the appropriate lines and columns. Remembering the proper print positions for several hundred forms was viewed as a difficult, if not impossible feat.

Attempts have been made to automate forms processing by designing separate programs to handle each form. There are two such programs in existence at CCAD now. Again, because of the number of forms to be automated, writing separate programs for each form was viewed as an insurmountable task.

Therefore, this effort was begun to design a system that could universally describe any form and interpret the description so that the process would culminate in a complete, properly aligned printed form.
INTRODUCTION

The completed system consists of five subsystems. The first of these subsystems is the form description system. The second subsystem is the user input-data system and the third subsystem is the print-preparation system. The fourth subsystem exists to allow changes to be made to the form description file and is therefore referred to as the update subsystem. An on-line "HELP" facility is provided to the user as the fifth subsystem.

In the forms description system a user (designer) will be required to define a form based on the field information requirements of its layout. The designer will use a set of attributes that were developed to distinguish between different input requirements to describe each field of a form. Each of these "form fields" will have a unique record present in a file named after the form. The "form field" records are created by the designer using the form description subsystem.

Once a description file for a form has been created and tested, any user may make use of the form description to enter data for their specific use. This task can be accomplished using the input-data subsystem. The system interprets the form description file and displays the prompt messages established for the various "form fields" onto the terminal screen. Depending on the specific "form field" requirements, a user may or may not enter data after the prompts. Data entered is prepared for output to another file that will be used in the print-preparation subsystem. Data preparation techniques such as left- or right-justification are performed.

The third subsystem, the print-preparation system, arranges the data records entered in the input-data system into records of common print lines.
The system fills the record with spaces between the data of the "form fields" on a form. Then the form, once inserted in a printer, is printed.

The update subsystem provides the designer with a method to make changes to an existing form description. Additions, deletions and changes can be made to perfect a form description. A search technique used by each subfunction searches on print positions to check for the pre-existence of records. This provides integrity of data because it prevents such inconsistencies as record duplication and records with overlapping print positions.

The fifth and final subsystem is the on-line "HELP" facility provided to the user as a training facility. The "HELP" facility can also be used to obtain helpful hints and reminders.

Each of the subsystems can be selected from the main menu screen. Each subsystem has adequate error checking throughout to catch user errors and to enable the user to make corrections at the time of data entry.

The actual detailed discussion of each of these subsystems will be presented with the program listings.
ENVIRONMENT

The equipment used to develop and test the programs was the VAX 11/730 minicomputer. This equipment is designated as the "office automation system" at CCAD and is apparently in widespread use throughout the majority of the Department of the Army. Therefore, the use of the "forms manager" system is not restricted simply to the local CCAD. Consequently, when a form description is completed at one Army installation, it could be shared with other installations that are also making use of the "forms manager" automated system.

Another point that encourages a larger user base is that the programs were written in the "C" compiler language. With the availability of the "C" compiler with the UNIX Operating System and many other operating system environments, "C" was determined to be a most practical choice of languages. It provides a necessary amount of transportability. The standardization of the "C" language enables any computer that has a "C" compiler to run the "forms manager" system.
There are three files created and processed in the "forms manager" system.

The first file is created in the form description subsystem and is referred to throughout the writeup as the form description file. This file is named by the form designer during the processing of the form description subsystem. It can be named using from 1 to 10 characters. The chosen characters should form some representative, meaningful file name.

The first record in the file of the form description is what is known as the list pointer. Once the file has been read into an array during further processing, this numeric value signifies the element of the array to be processed first. After updating the file, this value may not be the first physical array element, so that is the reason for its inclusion. The "C" data structure for this record is:

```c
int lptr;
```

The second record in this file is a flag whose function is to set lines of print/inch to print on the printer. The standard is 6 lines of print/inch but the designer may set the flag to indicate 8 lines of print/inch. This was found to be a necessary enhancement after the examination of over fifty forms. Some form layouts did not seem to line up as well at 6 lines of print/inch. Therefore 8 lines of print/inch was added as the only other option. The "C" data structure for this record is:

```c
int nlines;
```

The records that follow these first two records are those that each describe the specifications for a single "form field". The "form field" descriptors are usually entered into the file in the order they will be processed.
However, each has two pointer values that can be used to alter the logical processing order. The provision for pointers within the records affects a linked list.

The record format for the "form field" descriptors is described below. field-type code—up to 10 characters permitted classifies "form fields" according to the type of action that will be required of the user. Choices are "noinp", "ctrl", "nfield", and "cont" field-types.

next-field pointer—a numeric element that identifies the subscript of the array element of the next logical field to be processed. When this file is read later by the input-data subsystem, it is read in as an array of 200 maximum elements. Then this pointer value becomes meaningful. The last element of the array will contain a "-1" as this value.

alternate-next-field pointer—a numeric data element that can identify an alternate processing path to the one specified by the next-field pointer. An alternate path may be taken when a user enters <return> in response to a prompt during the input-data stage instead of legitimate data. It has another function when used in conjunction with a "ctrl" block of repeated fields. When a "form field" is the last of a block of repeated "form fields", a "-1" in this pointer will indicate so.

prompt—a character string terminated by a "\\" symbol. This will be the message that is displayed to the user to prompt for input or give
special instruction during the input-data stage.

justification flag--a one-character flag primarily used to signal left- or right- justification of the data entered in the input-data stage into its allotted space. An "l" or an "r" is permissible for this entry. The flag may instead indicate the presence of a line-feed after a "noinp" prompt message when used in a record of that "field-type code". A "y" or "n" is permissible for this function. A "/" is inserted by the program in this element for a record of field-type "ctrl".

line number--a numeric value that identifies the horizontal line of print of a "form field". For a record of field-type code "ctrl", this element is redefined as the value of the increment between the lines of print of the "form fields" in a block of repeated fields. For a record of field-type code "noinp" which should not have data to print, there will be a 0 assigned as this value.

starting column number--a numeric value representing the first vertical print position that the "form field" may print in. For a record of field-type code "ctrl", this element is redefined as the maximum number of lines a block of "form fields" needs to be repeatedly prompted for and printed. For a record of field-type code "noinp", which should not have data to print, there will be a 0 assigned as this value.

ending column number--a numeric value representing the last vertical print position to which the "form field" may print. For
records of field-type codes "ctrl" and "noinp" this data element is unused and is therefore always assigned a value of 0.

Presented below is a replica of the "C" data structure used in the programs to define the elements just described:

```c
struct rec
{
    char ftoode[10];
    int nxt;
    int alt;
    char [prompt[80];
    char jflag;
    int ln;
    int bcol;
    int ecol;
} ft;
```

Another file is constructed by the Input-data subsystem. This file is named after the form description file but with the letters "temp" appended to the end. The flag for the number of lines of print/inch is carried over to this file and is present as the first record.

All records which follow the first record are the same format. They are the formatted data records that the user entered when he was prompted for input. Each record, however, also contains several preliminary elements. Each contains its designated line of print and starting column of print, brought over from the form description file. Then there is a value representing the length of the allotted print space for data. This value is calculated by subtracting the starting column of print from the ending column of print. Next in each data record is the data entered by the user at each prompt. The data will exist left or right-justified in its space. Right-justified data will have one peculiarity. The first position will contain an "*" instead of a padding blank. This technique was implemented because of the
"C" language's character string manipulation characteristics. The "C" compiler will assign the contents of the input buffer to a character string variable until it encounters the first blank character. The blank is interpreted as a field separator. Therefore, the "*" masked in the first position serves to trick the compiler to continue reading as it assigns the string its data. Also a "|" character is placed in the last position of the string so that blank spaces can occur naturally throughout the data. A special format was designed so that the "fscanf" statement (file read) would read a string of characters up to the occurrence of the specified character, "|".

mentioned above.

The "C" data structures for this file is written out from the input-data subsystem as:

```c
int nlines;
int line;
int beg;
int wmlen;
char record[90];
```

It is read into the print-preparation subsystem as:

```c
int nlines;
struct prt
{
    int ln;
    int beg;
    int mlen;
    char record[90];
};
```

The third and final file produced by the "forms manager" system is a result of the print-preparation subsystem. This file replicates the data as it is actually printed on the form. The output of the print-preparation subsystem is printed on the form but as an enhancement to this system, it is also written to a file. This file will be named after the form description file but with the letters "out" appended to the end. One advantage of
having a saved copy of the print for the form is to enable a user to make small changes to adjust alignment of print within allotted spaces. Also if a particular form is used frequently and minor changes are all that will be required at each use, then a user may copy this saved file elsewhere and simply make those few changes in the new file.

This file consists of an array of structures. The structure consists of a character array. It is defined in "C" as:

```c
struct prout
{
    char outrec[102];
};
struct prout out[100];
```
The following section presents each of the programs and subroutines of the "forms manager" system in some detail. Each of the programs was compiled into separate object modules. Then all object modules were linked and loaded to produce one load module known as "formswriter". In fact, "formswriter" is what a user must type to begin using the system of programs.

One common characteristic of all of the programs is the data file that each uses. This is evidenced by the same "include" statement in all the programs. The following data structures found in the file "dataf.h" have also been made available to all of the programs.

```c
struct rec
{
    char ftcode[10];
    int nxt;
    int alt;
    char fprompt[80];
    char jflag;
    int ln;
    int bcol;
    int ecol;
    } ft;

char fname[10], tname[14], e;
int eflag, i, nlines, sub, flag, lptr;
int found, b, j, xln, xbcol, xecol, sw;
```
The first file of programs is "main.c". It begins with a module called "main". This program begins by calling the subroutine "clearscr" which clears a portion of the terminal screen by outputting 13 "newline" characters.

Next a title screen appears that says "WELCOME TO THE FORMS WRITER PROGRAM". A user is instructed to enter a <return> to continue processing. This technique is used quite often in these programs to provide the user time to read the screen.

The main menu screen appears next with six selections. The first five selections are the actual subsystems described earlier. The sixth selection exists to allow the user to exit the main menu and thus the system. The statement "Please enter the number of the function you wish to perform:" is displayed and the system waits for a response from the user. The following illustrates the main menu screen display:

1. Create the initial form description
2. Input data for use of a form
3. Run the routine to set up the input for printing
4. Run the routine to add, delete or change field descriptors for an existing form description file.
5. Print the 'HELP' file for user information
6. Quit this program

Please enter the number of the function you wish to perform:
Once the user has responded, the subroutine "func" is called. The only parameter passed to "func" is the value that the user chose from the main menu screen.

A case structure is the construct for this subroutine. The next subprogram called will depend on the choice made by the user from the main menu screen.

Case of "1" calls "crec", the routine that creates a form description.

Case of "2" calls "readin", the routine that prompts the user to input specific data based on a functional form description.

Case of "3" calls "prec", the routine that prepares the data entered by the user during the execution of the program "readin", for printing.

Case of "4" calls "update", the routine that can be used to make additions, deletions and changes to an existing form description file.

Case of "5" calls "helpf", the routine that processes several "help" files to display to the user on the terminal screen.

Case of "6" does not call a routine but exits the system of programs.

A default message has been established that will display "I don't understand" when the user has entered an unacceptable choice.

A listing for Main.c begins on page 43 in Appendix A.
This file of programs has the subroutine "crec" as its main process. "Crec" begins by calling the subroutine, "clearscr", which clears a portion of the terminal screen by outputting 13 "newline" characters.

The question "Input the name of the form" appears next, and the user's input is assigned to the variable "fname". The user is given a chance to correct this input when the question "Is this correct? ('y' or 'n')" is displayed. These two questions have been placed, as have most of the questions, in a "do-while" loop so that a user can continue entering this information until satisfied that it is correct.

The form name assigned by the user names a file for this form's description. The file is opened next in the "append mode".

The next section which begins with the question "Is this a re-start of the creation of a form description?", was added to the program for two reasons. First of all, it has been incorporated as a protection for the user. Most of the users of this system at CCAD will be operating a terminal that is connected to the Vax via modems and telephone lines. If interruption of communication or static in communication occurs during the execution of this program, the user's work will have been periodically saved. Therefore, all is not lost.

The second reason for providing the re-start capability is to enable the user to postpone execution of this program before the form description file is complete. The user may begin again later from a preserved point.

This enhancement is accomplished by the following activities:

After a completed form description record is written to the I/O buffers, a counter variable named "incr" is updated. When the counter reaches the value
"5", the buffer that has been accumulating the records is flushed or written to the form description file. The file pointer value is determined and saved by being output to another file named "keeppointer". Also a total of the number of records processed thus far (in multiples of five) is written out to "keeppointer". These two values, the file pointer value and the total number of records processed, are read from the "keeppointer" file when the response to the "re-start" question is "y".

The form description is positioned at the value of the file pointer. The total that was saved will be used to start the "creation loop" from a point other than the beginning of the process.

If the user answers "n" to the "re-start" question, the question below will appear:

"What should the printer be set to print, 8-lines per inch or 6-lines per inch for this form? (answer '6' or '8')"

The value "nlines", will be used later to change the printer capabilities. It was provided to handle unusual print spacing. When a processing session is not a "re-start", the pointer to the first logical record, "lptr", will be set to 0. The "nlines" variable and the "lptr" variable will be written to the form description file as the first two records.

Next, whether a processing session is a "re-start" or not, the question "Input the total number of fields needed to describe the form, top to bottom, left to right" will appear. The answer will be read into "num".

The "creation loop" is the main processing loop for the creation of individual "form field" descriptor records. It is iterated from 1 or the value of "tot" on a "re-start", to "num" times. It is performed using the "while" construct.

For each iteration the same questions will appear requiring responses from the user.
"Input the appropriate field-type code for this field."
The user's response is read in as a string into "ft.ftcode". An error
correction technique is incorporated.

"Input the number of the field to follow this field."
The user's response is read in as a numeric value into "ft.nxt". This is the
next-field pointer for processing. An error correction technique is incorpor-
ated.

"Input the number of the field that may follow this field as an alternate
to the "next-field" entered above".
The user's response is read in as a numeric value into "ft.alt". This is the
alternate-next-field pointer for processing. An error correction technique
is incorporated.

If the user entered "noinp", "nfld" or "ctrl" as the field-type code,
the question below will appear:

"Input the prompt and any special instructions that will be issued for
this field."
The user's response will be sent as "ft.fprompt" to a subroutine called
"read_line1". "Read_line1" is explained on page 21.

If the user entered "cont" as the field-type code, the prompt message
"ft.fprompt" will contain ":\", "|" and "\0" in its first three positions.
This will cause a ":" to appear as a prompt to the user during the input-
data stage for a "cont" field-type.
If the user entered an erroneous field-type code, error trapping provided will ask the user "Please re-enter a valid field-type code."

Once again if the user entered "cont" or "nfld" as the field-type code the questions below will appear:

"Will this field be printed left or right-justified?"
The user's response will be read in as a character into "ft.jflag". Only an "l" or an "r" will be accepted.

"What horizontal line will this field begin to print on?"
The user's response will be read in as a numeric value into "ft.lin".

"What vertical column will this field begin to print in?"
The user's response will be read in as a numeric value into "ft.bcol".

"What is the last vertical column that this field may print to?"
The user's response will be read in as a numeric value into "ft.ecol".

An error correction technique is incorporated for the three preceding questions.

If the user entered "noinp" as the field-type code, the program will assign zeros to "ft.lin", "ft.bcol" and "ft.ecol", the print positions. Then the following question will display:

"Will this field need a line-feed after its prompt is displayed to the user during input?"
The user's response will be read in as a character into "ft.jflag". This causes a line-feed to be inserted after the message prompt for a "noinp" field is displayed during the Input-data stage. An error correction technique is incorporated.

If the user entered "ctrl" as the field-type code, the program will assign a "/" character to "ft.jflag". Then the following questions will be displayed:

"Input the increment to be used between lines of print."
The user's response will be read in as a numeric value into "ft.in". This value will be used to add to the print line numbers of the fields in a block of repeated fields. An error correction technique is incorporated.

"Input the number of possible times that this block of field descriptors may be needed for."

The user's response is read in as a numeric value into "ft.bool". The block of repeated fields will be repeated for this number of times. An error correction technique is incorporated.

The variable "ft.icol" is assigned a value of 0 for the "ctrl" field-type.

When all of these questions have been answered to create a "form field" descriptor record, the record is written to the I/O buffers using the "fprintf" statement.

The "incr" variable is tested for a value equal to 5. This variable was discussed earlier during the explanation of the "re-start" question.

If "incr" does equal 5, then the buffers are flushed to the form description file. The file pointer and the total number of fields created are determined and written to a file called "keeppointer". The value for "incr" is reset to 1.

If "incr" does not equal 5 on any iteration, it will be incremented by 1.

At the end of each "creation loop" iteration, a variable "a" which serves as a loop counter is also incremented by 1.

The form description file is closed at the termination of the loop.

"Read line1" is a subroutine that is used to accept the user's input for a prompt message. It gets one character at a time until it encounters a "newline" character. At that time it substitutes the "newline" character with a "\n" character. A "null" character is added to the end.
"ClearScr" is a subroutine that uses a "for" loop to print 13 "newlines" on the terminal screen. This affects a screen clear.

A listing for Crform.c begins on page 45 in Appendix A.
SHOFORM.C

The main process for this file of programs is "readin". This subroutine
begins by asking the user to "Input the name of the form". The user's input
is assigned to the variable "fname". A file by the name given is opened.

The first record read is assigned to "Iptr", the first logical record
to be processed, or the list-pointer.

The second record is read into "nlines" and will stand for the number
of lines of print/inch for which the printer should be set.

A variable to serve as a subscript into an array, "sub", is initialized
to 0.

The form description file is read ("fscanf") into an array of structures
until the end-of-file occurs. When reading the string characters from the file
that are meant to be the prompt message, ("fprompt"), a special format is em-
ployed. This format "%[^|]" reads the characters up to the "|" character into
the character array, fprompt. A "while" construct is used to perform this read
process with the value of "sub" being incremented as a last step of each iter-
ation.

The form description file is closed and a new file is named and opened, a-
waiting activity. The new file is named after the form description file but
with "temp" appended to the end.

Next, the question below is posed to the user.

"Is this a re-start for the input of the data for this form?"

Only "y" or "n" are acceptable answers to this question.

If the user answers "y" to the re-start question a file called "timeout"
is opened and a new value for "sub" is read. This enhancement was made in case
the user had a previously interrupted input session, whether by choice or
not. The value for "sub" is written at the end of each iteration of the
"processing loop", to the "timeout" file.
If the user answers "n" to the re-start question, "sub" will be assigned the value of "lpotr". Also the value of "nlines" read in earlier is written to the new file with "temp" at the end of its name.

The "processing loop" is the next function to encounter. A "while" construct is employed to control the loop. The loop is processed by the value of "sub". "Sub" takes the value of the next field pointer or alternate field pointer at the end of each iteration. When a "-1" from one of those pointers is assigned to "sub", the processing loop terminates.

Within the processing loop, each array element's code, ("ft.ftocode"), is tested for its specific type using an "if-else if-else" construct.

When a field-type code of "noinp" is encountered the element "jflag" is tested for "y" or "n". If "jflag" is "y", the element's prompt message is displayed on the terminal screen with a "line-feed". Otherwise the prompt message is displayed without a "line-feed" after it. "Sub" gets the value of this element's next-field pointer, affecting the traversal of a linked list. This new value for "sub" is written out to the file named "timeout". This technique was employed to keep track of the array element at which to begin on a "re-start". Now the new element subscripted by "sub" will be processed.

When a field-type code of "ctrl" is encountered a message may appear such as: "The following block of repeated "form fields" will be repeated 20 times".

The value "inc" is set for the number of lines to increment between lines of print for the block of repeated "form fields". A processing flag "flag" is set to 1. "Block" will be set to the number of times the block of repeated "form fields" should be repeated. Next "sub" gets the value of this element's next-field pointer, affecting the traversal of a linked list. This new value for "sub" is assigned to "saveptr" and then written out to the file named "timeout". Now the new array element subscripted by "sub" will be processed.
When a field-type code of "nfld" is encountered, the value for "flag" is
tested. If "flag" is equal to 1, then it signals to the process that this
"form field" must be the first of a repeated block of "form fields".
The variables "block", "flag" and "incr", set in a preceding "ctrl" form field,
will be used to direct processing. With this particular processing, a
"for" loop is started to repeat the block's processing until the value of
"block". The variable "loopf" is initialized to 0 within this loop. The user
will see a message "inputting line 1" (or whatever line they are inputting).
This is to help the user keep track of input progress. A "while" construct
controls another loop that processes each form field in the block. It displays
each field in the block. It displays each prompt message and waits for user
input. The user's input is sent to a subroutine "read_line2" which is explained
later on page 28. An error correction technique is incorporated for this input.

If the user's response to the prompt message was not a <return>, the
length of the input is calculated using a "while" construct until a "|" char-
acter is encountered. Then the "form field" maximum length is calculated by
subtracting the element's starting print column from its ending print column.
These values are sent to a justification subroutine, "left-justify" or "right-
justify".

The variable "lout" is calculated by adding this element's line of print
number to the variable "incr". The data record for this element is written
out using the "wr_rec" subroutine that is explained on page 28. This element's
alternate-next field pointer is examined next. If it is equal to a "-1" it
signifies that this is the last "form field" in this block. Then the "block"
variable is tested to see if the number of iterations of the block of repeated
form fields has been performed. If the necessary number of iterations have
been made, "sub" gets the value of this element's next-field pointer. The new
value of "sub" is written to the file, "timeout". "Loopf" is assigned "1" to break out of the "while" loop. Now the new array element subscripted by "sub" will be processed.

However, if the block has not been repeated for the necessary number of iterations, then "loopf" is set to a "1" to break out of the "while" loop. "Sub" is assigned the value of "saveptr" which is the subscript that was saved for the first "form field" of the block. The new value of "sub" is written to the file, "timeout". "Incr" is incremented by the value of "inc". Now the new array element subscripted by "sub" will be processed.

Unless, if the alternate-next-field pointer for this element did not equal "-1", then processing should still take place with the block of repeated "form fields" and "sub" gets the value of this element's next-field pointer. The new value of "sub" is written out to the file, "timeout". Now the new array element subscripted by "sub" will be processed.

Yet still, going back to the entry of data to the prompt, if the user had entered a <return>, "sub" gets the value of this element's alternate-next-field pointer. The new value for "sub" is written to the file, "timeout". Now the new array element subscripted by "sub" will be processed.

At the end of the processing of a block of repeated "form fields", "flag" will be reset to 0.

If "flag" was set to zero when a "nfld" was encountered, then the following processing will take place for a "nfld" that is not part of a block of repeated "form fields".

First the element's prompt message will display on the terminal and wait for the user's input. The user's input will be sent to the "read_line2" subroutine explained on page 28. If the user's response to the prompt was not just a <return>, the length of the response will be determined using a
"for" loop. Next the maximum length of this "form field" is calculated for "mlen" by subtracting this element's starting column of print from its ending column of print. These values are sent to a justification routine, "left_justify" or "right_justify". Then the completed data record is sent to the "wr_rec" subroutine which is explained on page 28.

Whether the user had responded to the prompt message with a <return> or not, "sub" will get the value of this element's next-field pointer. Then the new value of "sub" is written out to the file, "timeout". Now the new array element subscripted by "sub" will be processed.

This completes the discussion of what can happen when a "nfld" is encountered as a field-type code.

When a field-type code "cont" is encountered, a ":" displays on the terminal as the prompt. Since this is a continuation field, the information entered should be an extension of some previous subject. The data entered by the user is sent to a subroutine called "read_line2" which is explained on page 28. An error correction technique is incorporated. If the user response was not a <return>, then the length of the response is calculated using a "for" loop. The maximum length of this "form field", "mlen", is calculated by subtracting this element's starting print column from its ending print column.

These values are sent to a justification routine, either "left_justify" or "right_justify". The completed data record is sent to the subroutine, "wr_rec" which is explained on page 28. "Sub" will receive the value of this element's next-field pointer. The new value for "sub" is written to the file, "timeout". Now the new array element subscripted by "sub" will be processed.

If the user's response to this "cont" field's prompt is a <return>, "sub" will get the value of this element's alternate-next-field pointer. The value for "sub" is written to the file, "timeout". Now the new array element
subscripted by "sub" will be processed.

When a field-type "del" is encountered, "sub" gets the value of this element's next-field pointer. The new value for "sub" is written to the file, "timeout". Now the new array element subscripted by "sub" will be processed.

When an erroneous field-type code is encountered, the run will be aborted by assigning a "-1" to "sub".

The first subroutine to be explained is "wr_rec". Its function is to write each data record in the following format to the file named with "temp" at its end.

line of print  ("line")
starting column of print  ("beg")
maximum print length  ("wmlen")
data record  ("record[]")

The character array "record[]" is written one character at a time until a "|" character is encountered. Then a "newline" is written to indicate end-of-record. The file is closed each time a record has been written. This was employed to assure the user that the record is actually written to the file. If the file is not closed each time, each record is simply written to the I/O buffer until the buffer is filled. Without this enhancement, if an interruption occurs during processing the user may lose all of those records not physically written to the file yet.

Instead, using this technique, the user will have written out records up to the last record added.

"Read_line2" is a subroutine that reads a character at a time from the terminal buffer. The characters will be designated as the data the user entered to a prompt.

The next subroutine is "left_justify", which takes care of left-justification of data entered by the user in response to a prompt. Blanks are
added from one position past the value of the length of the data, up to one position before the value of the maximum length of the field. In the position of the maximum length, a "|" character is inserted. Then from one position past the value of the length of the data, up to the 101st position, more blanks are inserted.

The last subroutine of this file of programs is "right_justify". For this right-justification routine, a special technique had to be developed. An "*" has to be placed in the first position of the data field because the "C" compiler interprets a blank space as a field separator when reading a string. The necessity of having blank characters padded to the left of the actual data, made it imperative to have a "non-blank" character in the first position of the string to trick the compiler to read further for the string.

Next the data is transferred one character at a time into the right-end of another character array. A "|" character is inserted in the position whose value is the maximum field length. Blank characters are inserted between the "*" and the first position of data. The new character array is replicated one character at a time over the original array. The blanks fill out the character array from the "|" character to the 101st position.

A listing for Shoform.c begins on page 53 in Appendix A.
PRFORM.C

This file of programs uses data structures other than those in "dataf.h". Its main processing routine is "prec". "Prec" begins by displaying:

"Input the name of the form."

The user's response is assigned to "fname". An error correction technique is incorporated.

Then the characters of the variable "fname" are assigned one at a time to the variable "tname", and the letters "temp" are concatenated to the end of "tname". The file with this constructed name is opened and read sequentially into an array of structures, "pr". This file is the same file that was output from the "shoform.c" program.

A subroutine called "sort_rtn" is referenced. "Sort_rtn" is explained on page 33.

Once the input array has been sorted, a new larger array of structures, "out", is initialized a character at a time with blanks. The last character of each record in the array is initialized with a "|" character.

For each element, the part of the input array that contains the data record array is examined. If in the first position of the data record array, an "#" is present (indicating right-justification), the "#" will be replaced by a blank character.

The next section uses a "for" loop to place a "|" character in the first position of those "out" array elements whose subscripts are less than the line of print of the first "pr" array element. A flag called "flag" is set to 1 to be tested later.

The next section organizes each of the "pr" records into lines of print, by examining each "pr" record's line of print. This is accomplished by assigning those records with common lines of print to the same larger "out"
array element. Also for lines of print that have no records
set to print on them, a "|" character is placed in the first position of the
"out" array elements.

Next, for those lines whose "out" array elements subscripts are greater
than the greatest line number of the "pr" records, "|" characters are also
placed in the first positions.

When the internal processing just discussed is complete, the user will
be given the following instructions:

"Please insert the form to be printed into the printer now."
"Make sure to align the form at the upper left corner."
"When you are ready to print the form, enter <return>.

When the user has entered a <return>, the subroutine "getterm" is called.
This is the last activity of the main process of the "prform.o" program,
though it is an important activity.

The subroutine "getterm" uses a call to "getenv" (get environment) to
produce the terminal type in use. Then a "case" structure of the two terminal
types at CCAD is employed.

The first option, constant "v", is used for the VT100 terminal manufac-
tured by DEC. The sequence of octal characters unique to the VT100 to open
the DIABLO printer port is presented next. When it is time to print the "out"
array, it will be printed on the printer and not the terminal screen. If the
"nlines" variable has a value of "6", the number of lines to print on a page
will be 66 ("limit"). Another subroutine, "printout", is called. "Printout"
is explained on page 32.

If the "nlines" variable has a value of "8", the number of lines to
print on a page will be 88 ("limit"). A sequence of octal characters is used
next to switch the DIABLO printer to print 8 lines per inch. The subroutine
"printout" is then called. "Printout" is explained on page 32. Upon return
from "printout", a system call is made to switch the printer back to print 6 lines per inch.

When printing has been accomplished for either of the lines per inch settings, the printer port is closed with another unique set of octal control characters.

The second option, constant "t", is used for the TVI950 terminal manufactured by TELEVIDEO. The sequence of octal characters unique to the TVI950 to open the DIABLO printer port is presented next. When it is time to print the "out" array, it will be printed on the printer and not the terminal. If the "nlines" variable has a value of "6", the number of lines to print on a page will be 66 ("limit"). The subroutine, "printout" is called. "Printout" is explained below.

If the "nlines" variable has a value of "8", the number of lines to print on a page will be 88 ("limit"). A sequence of octal characters is used next to switch the DIABLO printer to print 8 lines per inch. The subroutine "printout" is called next. "Printout" is explained below. Upon return from "printout", a system call is made to switch the printer back to print 6 lines per inch.

When printing has been accomplished for either of the lines per inch settings, the printer port is closed with another unique set of octal control characters.

The subroutine "printout", creates and opens a file named after the form with the letters "out" appended to the end. This file will be used to capture a copy of the formatted print for a form as it is also printed on the printer.

A "for" loop is used to traverse the elements of the "out" array. It will print characters to the printer and to the file just discussed. If a "|" is found in the first position of an "out" element, then a "newline" is output for
that element. Otherwise, all of the characters in an element up to the occurrence of a "|" character are output one at a time. The "|" character is changed to a "newline" character when it is encountered here also. The processing loop is terminated at the value of "limit".

The subroutine "sort_rtn", is an internal exchange sort. This sort was chosen over faster more complex sorts because the records to be sorted will most usually be in manageable numbers. The sort chosen is known as the "Shell sort". "The sort works by comparing elements at interval distances that decrease for each pass until this distance is equal to 1 for the last pass."*

The first distance is "d = sub/2" where "sub" is the number of records. Successive distances are calculated as "d = d \( \frac{d}{2} \)"

\[
\text{sub} \quad \text{sub - 1}
\]

"The progression through the list is from the top, one element at a time, until an exchange occurs or until the end of the list is reached. When an exchange does occur, a check is made to see if the exchange will be propagated back through the list by backing up at intervals of the current value of 'd' until no exchange occurs: then the procedure resumes at the point at which the exchange started. This process moves records that are greatly out of order into position with fewer compares and exchanges; thus it is faster."*

A listing for Prform.c begins on page 63 in Appendix A.

UPFORM.C

This file of programs can be executed to permit updates to an existing form
description file. Its main processing routine is "update". When "update"
begins, the user specifies the same name of the form used originally in the
creation of the form description. That file is opened and the list-pointer, "lptr", is read. Each form field descriptor record is then read sequentially
into an array.

The variable, "j", which will be used later to traverse through the array,
is assigned the value of the list-pointer, "lptr". A flag called "found" is
initialized to 0. The variable, "b", also to be used later as a previous record
indicator, is set to an initial value of "-1".

Next, within a "do loop", the following message and set of choices is dis-
played:

"Please enter the letter of the action you wish to perform on the form de-
scription file you have selected."

"'a' for add a field"

"'d' for delete a field"

"'c' for change a field"

"'q' for quit and return to main menu screen"

If an "a" is selected, the subroutine "add_field" is called. A flag
"eflag" is reset to a value of 1. "Add_field" is explained on page 36.

If a "d" is selected, the subroutine "srch_rtn" is called after a value
of 0 is assigned to "sw". "Sw" is a switch used by the "srch_rtn". "Srch_rtn"
is explained on page 35. Then "del_field", explained on page 36, is called. As
after the "add_field" subroutine, the flag "eflag" is reset to a value of 1.

If a "c" is selected, the switch "sw" is set to 0 and "srch_rtn" is
called. "Srch_rtn" is explained on page 35. "Change_rtn", explained on
page 37 is called next, just before "eflag" is once again reset to 1.

If a "q" is selected, "eflag" is simply set to a value of 0, which will cause an exit from the "do-loop" used for the selection of update actions.

If a character other than those permitted is entered as a selection, an error correction technique is executed.

When all actions have been performed on the form description file, the file is rewritten. The rewrite operation occurs from "lptr" and "nlines", to the last form field descriptor. The file is closed.

The "srch_rtn" subroutine is the first subroutine to be explained. It begins by assigning a variable "k" the value of "lptr". If the switch "sw" enters this routine with a value of 0, the user will be asked to provide the information to the three following requests:

"Please input the line of print for this field." (xln)
"Please input the starting print position for this filed." (xbcol)
"Please input the ending print position for this field." (xecol)

Then regardless of its previous value, "sw" is set to 0. A "do-loop" is used to search through the array. Searching is performed by trying to match the field values of the current array element to the search arguments "xln", "xbcol" and "xecol".

If a record matches all three search arguments, the record that was searched for is assumed to be found and "found" is set to 1.

If a record matches the search arguments for the line of print ("xln") but incurs an overlap with its other print positions, a message will appears. The message warns the user of overlap and sets the value of "found" to 3. Otherwise, the next record is retrieved by following next-field pointers. Comparisons are made until a next-field value of a "-1" is encountered or the requested record is found. When the record is
not found, "found" will be set to 2.

The subroutine "add_field" appears next in the listing. The first variable "j" is set to the current value of "sub". The user is taken through the necessary prompts to build a new form field descriptor record.

Depending on the field-type code entered for the new form field, the user will be offered the appropriate requests for information to continue building the rest of the form field descriptor record.

Finally "sw" is set to 1 for the subroutine "srch_rtn". This switch will be used to indicate to the search routine that the search arguments "xln", "xbcol" and "xecol" have already been assigned.

Still within the "add_field" subroutine, "srch_rtn" is called if the field-type code of the record to be added is not "noinp". Then if "found" has a value of 1, the user is warned of a duplicate record and is not permitted to add that record.

If "found" is equal to 2, the user will be asked to provide the information concerning the location of record insertion. For instance, if the new record should be the first record to be processed of the form fields, the list-pointer will have this record's subscript value as its new value. Otherwise, if the record should be inserted between two existing records, the user must provide the subscript value of this record's predecessor. The program will adjust next-field pointers accordingly.

The value of "sub" is incremented by 1 so that the additional record will be written to the form description file as the file is rewritten.

"Del_field" is the next subroutine to occur. If "found" enters this routine with a value of 1, indicating the requested record to delete was found, then the first four positions of the record's field-type code ("ftocode") are
changed to 'd', 'e', 'l', and '\0' respectively. If "found" is a value other than 1, the user is told that the field was not found and cannot be deleted.

The "read_line" subroutine is called when the user is building a new form field descriptor. It is used to accept the string of data the user enters for the prompt message ("prompt"). This subroutine examines the string until it encounters a "newline" character.

The "change_rtn" is the last subroutine included in this file of programs. If "found" enters this routine as a 1, then the record to be changed has been found. Many things will happen, beginning with the display of the record to be changed. Otherwise, the user is told that the record has not been found and cannot be changed.

Then a list of the parts of a record that may be changed is presented in a "do-loop". The "do-loop" enables the user to make more than one change to a record, though only one change per iteration. The list of parts to change are numbered from 1 - 8, with 9 ending the change session entirely. The number entered as a selection drives a "case" structure for different activities.

Case of "1" permits the user to change the field-type code. A warning for this change is presented. A different set of information requests and assignments are given for each of the possible field-type codes.

Case of "2" permits the user to change the value of the record's next-field pointer. A warning for this change is presented.

Case of "3" permits the user to change the value of the record's alternate-next-field pointer. A warning for this change is presented.

Case of "4" permits the user to enter a new prompt message for the record. It calls the subroutine "read_line" to read the string for the prompt.

Case of "5" permits the user to change the justification of the record's data. Only an "l" or an "r" is permitted to be entered.

Case of "6" permits the user to change the line of print for the record. A
warning for this change is presented.

Case of "7" permits the user to change the starting column of print for the record. A warning for the change is presented.

Case of "8" permits the user to change the ending column of print for the record. A warning for this change is presented.

A listing for Upform.c begins on page 70 in Appendix A.
HELPFILE.C

This file of programs presents several "HELP" files for viewing by users. In the main processing routine "helpf", a title screen appears then waits for the user to enter a <return> to continue.

Next a screen called the "HELP" menu is displayed. Five choices to view different "HELP" files are given, the sixth selection being the choice to exit the "HELP" facility.

The user will see the instruction:

"PLEASE ENTER THE NUMBER OF THE SUBJECT YOU WISH TO STUDY."

Here is a sample of the "HELP" menu screen:

1. DISPLAY INFORMATION TO EXPLAIN CREATION OF FORM DESCRIPTION PROCESS
2. DISPLAY INFORMATION TO EXPLAIN THE DATA-INPUT PROCESS
3. DISPLAY INFORMATION TO EXPLAIN THE PRINT-PREPARATION PROCESS
4. DISPLAY INFORMATION TO EXPLAIN THE UPDATE OF A FORM DESCRIPTION PROCESS
5. DISPLAY INFORMATION TO USE 'VI EDITOR' TO MAKE CHANGES TO FORM DESCRIPTION
6. QUIT OR EXIT THE HELP FACILITY

PLEASE ENTER THE NUMBER OF THE SUBJECT YOU WISH TO STUDY
At the entry of a selection number, the subroutine "func1" is called. "Func1" performs two functions. First, using a "case" structure, a file name is constructed by concatenating the letter "h" with the number entered as the choice. For instance, if number "1" was chosen from the "HELP" menu screen, the file name "h1" will be constructed.

Secondly, "func1" will open that file whose name was constructed and display a screenful at a time. Basically, 80 characters or up to the first occurrence of a "newline", are read from the file. A "newline" is changed to a "null" character. The line of characters is now ready to be displayed on the screen. This process is repeated until 21 lines are displayed on the screen. Then the user must enter a <return> to view another screenful of the file. Encountering end-of-file for a particular "HELP" file causes that file to be closed. The user will be returned to the "HELP" menu screen to view other "HELP" files. The user may exit the process with a selection of "6".

A listing for Helpfile.c begins on page 84 in Appendix A.
CONCLUSIONS

In the project proposal five desirable end results were stated.

The first result desired was to create a system of programs that were user-friendly. This result was achieved. Any user should be able to design and use a form description using the comprehensive users' manual written for the system.

The proposed project was to permit a typist to enter data to a form without concern for proper spacing and alignment. This result was achieved. A form description will need to be established and fine-tuned to achieve precise form printing. However, when a form description is ready for use, a typist should be free of the responsibility of aligning the data entered into proper spaces.

The system of programs was to include provisions for adding, deleting or changing records in an existing form description. This result was achieved. There is an entire set of programs that allows the user to perform such activities. The updates are available as one of the selections from the main menu screen.

Adequate error checking and input re-try methods were to be incorporated in the system of programs. This result was achieved. At each plausible occurrence of input from the user, steps were taken to remind the user to check his entry. At other times, data was checked by the program for invalid responses.

The final result expected was for the author to attain a greater understanding of the "C" programming language. This result was achieved. At the onset of this project, the author had not read about or made use of this language. Self-instruction was performed by reference and trial and error. The author now has a greater understanding of the "C" programming language. This can be verified by reviewing the coding used to program the five major modules involved in the Forms Manager System.
BIBLIOGRAPHY

The sources used for reference in this project include:


3. Miles, Donald B. "Program for automation of Disposition Form". Corpus Christi, TX: Corpus Christi Army Depot, 1983.


42