A File System Extension to Micro-PL/CS

by

James Archer, Jr.

TR79-366

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by a grant from the NSF MCS77-08198.

A File System Extension to Micro-PL/CS

James Archer, Jr.
Department of Computer Science
Cornell University

Micro-PL/CS is a version of PL/CS developed for interactive use with a dedicated microprocessor. A file system extension is proposed to give PL/CS a simple, but extremely powerful file system capability. The system allows for the creation and manipulation of files for sequential, random, or keyed access (or any combination) in an unrestricted manner. Essential to the capability is a set of built-in functions and pseudo-variables which allow file manipulation without syntactic complication.

PL/CS is an instructional dialect of PL/I designed to support the contemporary view of programming and to assist in teaching the development of correct programs. It is defined by selecting a subset of features from PL/I, and then restricting the manner in which the features can be used [3]. The original PL/CS implementation was an error-correcting batch compiler based on PL/C [2, 4].

Micro-PL/CS is a new system developed for use on a standalone microprocessor. Typically, a system hardware configuration consists of a processor (conceptually standalone, but possibly time-shared), a CRT terminal connected to it by a high-speed link, and random access secondary storage (floppy disk). Micro-PL/CS is based upon a syntax-directed synthesizer (SDS) which actively assists the user in the structured development of correct programs [6]. Program development using the SDS relies heavily on the speed and flexibility of the CRT display.

Micro-PL/CS provides a uniquely hospitable environment for program development. From the user's point of view, programs simply exist. The version being executed is identical to the permanent version, since all changes to programs are automatically permanent unless specifically "undone" [5]. The system appears to execute program text directly. In fact, programs are stored as directly interpretable syntax trees specifically for the purpose of allowing free movement from program entry and modification to execution and back. An optional trace facility moves the CRT cursor through the program text as it is being executed. Execution errors return to the user with the cursor at the offending program element. The user is then free to perform any system function, including changing this (or any other program), altering the values of active variables, or terminating the session; all without losing the current program state. Any program statement can also be executed in "immediate mode".

The addition of a general file processing capability to Micro-PL/CS serves two purposes: it makes Micro-PL/CS useful in a wider variety of instructional situtations, for a larger set of problems, and it makes Micro-PL/CS an attractive system for the development of a personalized computing facility. The objective is to define a permanent file system which will be both simple enough for use by the beginning student and yet powerful enough to be attractive to a more demanding user.

The dominant criterion is <u>simplicity of use</u>. PL/CS is a deliberately simple language; it has a total of fourteen (including END and null) distinct statement types [3]. The PL/CS design philosophy requires that the number of new statement types and constructs for the file system extension be kept to a minimum. Only one statement, DELETE, has been added. The ability to reference files (the FILE phrase) has been added to GET and PUT, and a "start new record" capability, NEXT (similar to PL/I SKIP), has been added to the GET statement (PL/CS GET does not have SKIP as an option). NEXT is used in place of SKIP to increase the syntactic separation of GET and PUT to facilitate PL/CS error repair.

In order to achieve a complete file system without the plethora of PL/I statements (OPEN, CLOSE, READ, WRITE, etc.) and phrases (KEYTO, KEYFROM, GENKEY, etc.), a small, but powerful, set of pseudo-variables and built-in functions has been employed. In a direct departure from PL/I usage, the functions/pseudo-variables directly control much of the file processing. In the cases of COUNT, KEY, and REC, the file may actually be <u>modified by assignment to the pseudo-variable</u>. Although all potential error conditions can be detected before they occur, reliance is placed on the interactive nature of the system to allow the user to recover from unanticipated conditions.

PL/CS does not currently support structures. Without structures, the standard PL/I record-oriented constructs are particularly unattractive. Further, considerable expertise, use of sophisticated storage allocation facilities, and attention to detail are required to process records with variable numbers of items each of variable length in PL/I. PL/I attempts to provide an <u>efficient</u> file capability; Micro-PL/CS requires an <u>effective</u> system independent of implementation efficiency. Efficiency of implementation has intentionally been excluded from the file extension design criteria.

<u>System Overview</u>

Each user has an independent file system -- there is no provision for sharing or communication between users or for the processing of "compatible" media for interchange with other systems. A file system is a collection of files.

1) A <u>file</u> is a named sequence of records. The file-name is an arbitrary-length string of printable characters. In actual language statements, the file-name may be any character expression. If the file-name is a constant, the value of the constant is the actual file name; if the file-name is a variable or expression, the <u>value</u> of the variable or expression is the actual file name. An uninitialized character string variable used as a file-name variable is an error; the name of the variable is <u>not</u> assumed to be the name of the file (as is the case in PL/I). File names are global -- known to all programs run in the user's system. These values do not conflict with identifiers in any program.

2) A <u>record</u> is a sequence of items. Each record is numbered by its sequential position within its file. The number of a record changes whenever a record is added to or deleted from a lower-numbered position in the file. Optionally, each record can be named by a "key-value". The key-value is any legal PL/CS character string value not starting with HIGH or LOW; the key-value is unique for the file. The key-value of a record cannot be changed (though the same contents can be rewritten with a new key).

3) <u>Items</u> are distinct objects in "LIST" format. If an "EDIT" format output statement is used, all of the output is a single item. The

PL/I "DATA" format is not supported.

4) The current record pointer is the record number of the current position in the file.

5) The current item pointer is the number of the next item in the current record of the file.

The basis for the file system extension to Micro-PL/CS is the Unrestricted File Organization (UFO) underlying it. UFO supports direct-access (by key-value or record number) and sequential file operations. Sequential operations may span multiple records of a file, or apply within a single record. Any file may be accessed or modified by any of the available mechanisms. No feature depends on the file declaration; there are no file variables. In particular, the following characteristics of UFO files are noteworthy:

1) Some records are keyed, others are not.

2) Records with keys are in key-value order (following the standard rules for character string comparison).

3) Record numbers are sequential.

4) The number of items in a record can vary from record to record

5) Records may be inserted at/deleted from/modified at any position in the file.

Consider the following section of a typical file selected to illustrate the basic characteristics detailed above:

Key	Number	Contents
ace	15	item 1▯ item 2▯ item 3▮
	16	item 1▯ item 2▮
lob	17	item 1▮
serve	18	▮
smash	19	item 1▯ ...▯ item n▮

If a record were written with a key-value of "forehand", it would be inserted as record 17, causing the record with the key "lob" to be numbered 18, etc. Unkeyed records are considered to be subsidiary to the immediately preceding keyed record (if any exists) for the purpose of record placement.

Built-In Functions and Pseudo-Variables

The following set of functions/pseudo-variables are used with the "standard" PL/C GET and PUT statements (summarized below) to control file operations. The functions/pseudo-variables for a particular file are tightly coupled: assigning to any of the pseudo-variables can change the values of any of the other functions for that file. A summary of these interactions follows the descriptions. A set of Micro-PL/CS examples is given to show the power and simplicity of the facility provided. The facilities are compared with those provided by PL/I.

Four functions/pseudo-variables are provided to perform record positioning in the file: REC, REMAIN, KEY, and FIND. A brief definition of each follows:

REC(file-name) (function and pseudo-variable)

Returns the current record number (from 1 to the size of the file). The value returned for an unreferenced file is 0; for an undefined file -1.

Assignment to the pseudo-variable causes the current record pointer to be set to the numbered record. Values may be assigned which arbitrarily extend the file; negative values are errors.

REMAIN(file-name) (function)

Returns the number of records until end-of-file is encountered on the file. A 0 value indicates that the current record is the last record in the file. For a file that has not been read, REMAIN(file-name) is the number of records in the file; -1 if the file is not present. For an existing file, REMAIN(file-name)+REC(file-name) is always equal to the number of records.

KEY(file-name) (function and pseudo-variable)

Returns the character string value of the key for the current record in the file, file-name. If the current record is unkeyed, KEY returns the key of the most closely preceding keyed record, or LOW if no preceding keyed record exists.

Assignment to the pseudo-variable sets the current record pointer to the record with the assigned key-value. Assignment of a key-value that did not previously exist causes a null record with that key to be created.

FIND(file-name, key-value) (function)

Sets the current record pointer at the first record whose key is greater than or equal to key-value and returns the character-string value of that key. If no record exists with a key-value greater than or equal to key-value, HIGH is returned and the current file pointer is set to "end of file".

The REC function locates records on the basis of absolute record number. Using the REC pseudo-variable, it is possible to move freely back and forth within the file, or to add records to the end. Through the use of REC in conjunction with GET and PUT, it is easily possible to perform random, unkeyed file operations.

REMAIN is primarily useful as a means of controlling simple sequential operations without the need for ON conditions. The ability to <u>predict</u> end-of-file makes simpler and more elegant programming possible. The program knows when the last record is being processed, not just when it has tried to read more records than are available. Extensive use of REMAIN is made in the examples below. It should be noted that the PL/CS "ON ENDFILE GOTO ..." construction has been retained only for terminal input. <u>It is not possible to predict the last record of terminal input</u> without the assistance of the user. The user would be forced to specially designate the last line, an unreasonable requirement.

The KEY pseudo-variable is the sole method for placing keys in a file. Assignment to the KEY pseudo-variable positions to the correct place in the file and creates a null keyed record if no previous record existed with that key. The KEY function provides a way of determining the key of the current record, regardless of what processing methodology was used to reach the record.

The FIND function provides a way to locate a specific key or the place in the file where a specific key or a group of keys would be placed. This location capability is effective even if the key found does not resemble the key searched for. This is in contrast to the corresponding PL/I GENKEY facility which requires that the key string searched for be a prefix of the string actually found. It is possible to test for the existence of a particular key by a comparison of the form:

```
IF (FIND(file, key) = key) ...
```

The corresponding PL/I operation would require the use of the ON KEY construction.

The UFO is based on records containing variable numbers of items. In order to easily take advantage of this facility, two function/pseudo-variables, COUNT (no relation to the PL/I function of the same name) and ITEM, are provided. Brief descriptions follow:

COUNT(file-name) (function and pseudo-variable)

> Returns the number of items in the current record for the file-name.
>
> Assignment to the pseudo-variable changes the number of items in the current record, either truncating or extending the record with null items.

ITEM(file-name) (function and pseudo-variable)

> Returns the current item number in the current record of the file. The first value for any record is 1.
>
> Assignment to the pseudo-variable moves the current item pointer to the specified position within the record. It is an error to assign a value to ITEM(file-name) that is greater than COUNT(file-name) or less than 1.

The COUNT pseudo-variable allows the user to set the size of the record to any value that he would like, without the requirement that he have data to fill each item. The COUNT function allows the user to check the number of items, making it easily possible to process files with variable numbers of items. The ITEM pseudo-variable allows the user to move around within the record at will; the function allows him to check where he is. ITEM and COUNT make it entirely reasonable for the novice user to deal with variable numbers of items.

Program Statements

Brief descriptions of the syntax and semantics of the GET, PUT and DELETE statements follow. Independent of input form, PL/CS always prints out program statements in a canonical order; that order is reflected by the order of the optional phrases in the prototype statements.

```
PUT  [FILE(file-name)]  [SKIP[(n)]]   |LIST(list)       | ;
                                      |EDIT(list)(list)|
```

1. If file-name is new, this constitutes an implicit declaration of a new file-name and creates a new file. If the file-name is a constant, the value of the constant is the actual file name; if the file-name is a variable or expression, the <u>value</u> of the variable or expression is the actual file name. An uninitialized character string variable used as a file-name variable is an error; the name of the variable is <u>not</u> assumed to be the name of the file.

2. If file-name is known, statement applies to existing file. If the first statement or pseudo-variable referencing the file is a PUT, the current file pointer is initialized to filesize+1, and the current item pointer to 0.

3. If the SKIP phrase is omitted, items are written starting at the end of the current record, i.e. at REC(file-name), COUNT(file-name)+1.

4. If the SKIP phrase is specified:

   a. If the SKIP count, n, is positive, n records will be written, numbered REC(file-name)+1 ... REC(file-name)+n. Any items transmitted will be to REC(file-name)+n. As a result, the preceding n-1 records will be null.

   b. If the SKIP count is zero, output begins with ITEM(file-name), updating exisiting fields or extending the record as needed.

   c. Negative SKIP count is an error.

5. If FILE phrase is omitted, the statement refers to the standard system output, i.e. the terminal. For compatibility with PL/C, the name "SYSPRINT" is reserved for the standard system output.

6. Both the LIST and EDIT phrases can be omitted. This construction can be used to create null records or to truncate or extend the current record.

```
GET    [FILE(file-name)]    [NEXT]    |LIST(list)      | ;
                                      |EDIT(list)(list)|
```

1. If FILE phrase is omitted, GET applies to the standard input, i.e. the user's terminal. For compatibility with PL/C, the file-name "SYSIN" is reserved for the standard input. If the first statement or pseudo-variable referencing the file is a GET, the current file pointer is initialized to 0.

2. If NEXT is present, REC(file-name) is incremented and ITEM(file-name) is set to 1 prior to data transmission.

3. LIST format items are read starting with REC(file-name), ITEM(file-name). Record boundaries are ignored. Null records are ignored when searching for the next item of the data list.

4. EDIT format input is limited to the current item; ITEM(file-name) is incremented by 1.

```
DELETE FILE(file-name) [RECORD[(n)]];
```

1. If RECORD phrase is present, deletes the number of records specified (one is the default) starting with REC(file-name). REC(file-name) remains unchanged, but refers to the first record following those deleted. The number of records to be deleted must be positive.

2. If RECORD phrase is absent, this statement deletes the entire file.

Compatibility of GET and PUT

For instructional purposes (and general user convenience), it is desirable for Micro-PL/CS constructs to resemble their PL/CS, PL/C, and PL/I counterparts wherever possible. The new facilities provided by the Micro-PL/CS file system extension are clearly not compatible with preceding systems. Unfortunately, compatibility is a problem even for the simplest sequential operations. The Micro-PL/CS file system is built on the premise that what is placed in a file by a PUT statement can be simply and straightforwardly retrieved by a GET statement. The PL/I GET LIST and PUT LIST statements do not support this symmetry. Character strings are output to be human-readable, i.e. without quotes; character strings must be enclosed in quotes on input for the data to be uniquely readable. For similar reasons, user terminal input uses the same rules as PL/I, although error-correction procedures can relax the requirements considerably. The system must be able to detect the difference between "raw" input and already processed items.

The user can create files using the text entry facilities of Micro-PL/CS. This process makes use of a conceptually distinct (and simpler) SDS for data files.

Through interaction with the SDS, the user can directly enter items, group them into records, and insert keys. The user will need to alert the system if he is not entering standard LIST format input. Further, any input that is entered in EDIT format will have to be read by an EDIT statement or processed internally as a character string. In the Micro-PL/CS environment, the LIST format is more efficient and more flexible, making it relatively unattractive to adhere to the EDIT format for other than printing purposes. LIST format data entry has the further advantage of providing an SDS to assist in the entry of syntactically correct data items.

Summary of File Interactions

The effects of each of the file system statements, pseudo-variables and the FIND built-in function are summarized below. Each table entry represents the effect of the execution of the left-hand column construct on the column header.

	KEY	REC	ITEM	errors
KEY	=	corresponding record number	1	HIGH or LOW
REC	key of closest <= record	=	1	argument < 0
ITEM	unchanged	unchanged	=	argument <= 0 or > COUNT
COUNT	unchanged	unchanged	COUNT+1	argument < 0
FIND	key of closest >= record	corresponding record number	1	none
GET	key of closest <= record	current record pointer	updated	attempt to read past end of file
PUT	unchanged	current plus SKIP count	updated	none
DELETE	key of closest <= record	unchanged	1	records to be deleted <= 0

The value of REMAIN(file-name) will always be the total file size minus REC(file-name). Error conditions are always referred to the operator; no function values are changed by errors.

File System Structure

The file system directory ("<files>") is itself readable as a file keyed by file name. In addition to "standard" file information (e.g. file size, file type, time of last reference and modification), the record corresponding to each file contains a field for each of the keys in the file itself. The system provides the ability to read program files (SDS syntax trees) with each record being a line of source text in display form. Neither the file system directory nor the program files can be written by the user.

Processing Examples

The following examples are intended to illustrate the functions and statements described above in stylized processing situations. These examples were chosen to be indicative of file processing tasks which might be required by the PL/CS user community. User-variables are presented in lower-case; PL/CS statements, functions and pseudo-variables in upper-case.

1) Sequential file print, variable length records. Null records cause blank lines to be printed.

```
DO WHILE (REMAIN(source) > 0);
    /** access next record and start new print line */
        GET FILE(source) NEXT;
        PUT SKIP;

    /** print each of the list items */
        repeat = COUNT(source);
        DO i = 1 TO repeat BY 1;
            GET FILE(source) LIST(str);
            PUT LIST(str);
            END;
    END;
```

2) Print section of given file between the supplied key-values start and stop. Print the key, and the first, second and tenth items.

```
/** determine filename, start and stop keys */
    GET LIST(indexed, start, stop);

/** find the first record to be printed */
    start = FIND(indexed,start);

/** print key and items 1, 2, 10 for records with key <= stop */
    DO WHILE ((KEY(indexed) <= stop) & (REMAIN(indexed) > 0);
        GET FILE(indexed) NEXT LIST(a, b);
        ITEM(indexed) = 10;
        GET FILE(indexed) LIST(c);
        PUT SKIP LIST(KEY(indexed), a, b, c);
        END;
```

3) Read student keys and grades from terminal and add to file.

```
ON ENDFILE GOTO stop_processing;
...
DO WHILE ('1'b);
    GET LIST(student, grade);
    /** get student record -- key errors ignored */
        KEY(CS632) = student;
        PUT FILE(CS632) LIST(grade);
    END;
```

or, to intercept errors

```
    ...
    /** get student record if it exists */
       IF (FIND(CS632,student) = student)
           THEN PUT FILE(CS632) LIST(grade);
           ELSE PUT SKIP LIST('***student not found', student);
```

4) Print file system directory.

```
DO WHILE (REMAIN('<files>') > 0);
    GET FILE('<files>') NEXT LIST(filename, filesize);
    PUT SKIP LIST(filename, filesize);
    END;
```

5) Print all companies with credit balances.

```
DO WHILE (REMAIN('A/R') > 0);
    GET FILE('A/R') NEXT LIST(name, balance);
    IF (balance < 0)
        THEN PUT SKIP LIST(KEY('A/R'), name, balance);
    END;
```

6) Merge two files; add field from f2 to records in f1.

```
/** add items from f2 to records in f1 */
    DO WHILE ((REMAIN(f1) > 0) & (REMAIN(f2) > 0));
        GET FILE(f1) NEXT;
        GET FILE(f2) LIST(d);
        PUT FILE(f1) LIST(d);
        END;

/** mark unmatched records from f1 with "missing" field */
    DO WHILE (REMAIN(f1) > 0);
        GET FILE(f1) NEXT;
        PUT FILE(f1) LIST('missing');
        END;
```

Note that it would also be possible to make the records consist entirely of the first three items of f1 and the first from f2 by a replacement of the form:

```
...
GET FILE(f1) NEXT;
GET FILE(f2) LIST(d);
COUNT(f1) = 3;
PUT FILE(f1) LIST(d);
```

or to have the first item replace the fourth item of f1:

```
...
GET FILE(f1) NEXT LIST(a, b, c);
GET FILE(f2) LIST(d);
PUT FILE(f1) SKIP(0) LIST(d);
```

7) Temporary storage for intermediate results.

```
DCL x(10,20) FLOAT;

PUT FILE('x') LIST(x);
...
GET FILE('x') LIST(x);
...
DELETE FILE('x');
```

or

```
KEY(temp) = 'x';
PUT FILE(temp) SKIP LIST(x);
...
KEY(temp) = 'x';
GET FILE(temp) LIST(x);
...
KEY(temp) = 'x';
DELETE FILE(temp) RECORD;
```

Comparison with PL/I

The file processing capabilities provided by Micro-PL/CS are syntactically and algorithmically simpler for the user than those provided by PL/I. Syntactically, PL/I provides a statement or statement phrase to handle each variation of function desired. The syntax reflects the data management philosophy of large, batch operating systems which allow (and frequently require) the user to make myriad detailed decisions about file allocation, representation and processing. For the single-user, interactive environment, this "flexibility" is an obstacle rather than an opportunity.

All PL/I file processing (other than the simplest sequential) is done using record-oriented constructs, most commonly in conjunction with PL/I structures. The user has three major options for this type of processing:

1) Predetermine data format, type and length. This allows fixed-length records and access to all processing capabilities. Over-allocation is commonly used to allow room for growth and modification. Mistakes in allocation are rectified by "conversion" programs which expand the file records to meet the newly perceived requirements.

2) Use BASED or CONTROLLED storage to allocate structures of differing sizes to meet differing needs. Except where efficiency is vital, only the most experienced (or ambitious) user will choose this method over 1) above.

3) Use varying length character strings for all data transmission. The actual records can be superimposed (using DEFINED or BASED). Alternatively, the record can be "structured" by GET/PUT STRING operations. Logically, this is the closest alternative to the PL/CS solution.

Algorithmically, Micro-PL/CS also provides significant simplification. For the novice (or casual) user, this simplification is very important. One of the primary areas of potential usefulness of a personal computing system lies in its file capabilities; the complexity of the user's files should be governed by the task that he wishes to accomplish, not by the difficulty of fitting the task to an efficient implementation.

Micro-PL/CS simplifies algorithms by its increased range of permissible file operations. The intent is for Micro-PL/CS to allow any file processing operation which can be simply and unambiguously stated. Records can be expanded, truncated, modified, overwritten or deleted at will. Files can be created or destroyed under program control without knowledge of system structure or a separate control language. All user files are capable of supporting all operations at all times.

The simplified handling of exceptional conditions makes realistic processing much more straightforward. The interactive nature of Micro-PL/CS itself aids in handling exceptional conditions. Just as importantly, it is possible to detect any possible "exceptional" condition before it occurs. One of the reasons why real PL/I file processing programs frequently have complicated ON units is that most exceptional conditions cannot be predicted; _PL/I makes it impossible to detect errors in advance_. The most commonly encountered examples of this are the ENDFILE and KEY conditions.

The following two PL/I examples are approximately comparable with two of the Micro-PL/CS examples above (the numbers have been retained). Some simplifications have been made to make the PL/I programs tractable. Both PL/I examples use fixed-length records. Situations for which no simple PL/I solution exists are noted in comments rather than attempting a "solution". Declarations are included only where they are deemed important to the understanding of program operation. Although PL/I programs would actually exist entirely in upper case, the upper/lower case distinction used in PL/CS has been retained for ease of comparison and improved readability.

2) Print section of given file between the supplied key-values start and stop. Print the key, and the first, second and tenth items.

```
DECLARE indexed FILE RECORD SEQUENTIAL ENVIRONMENT(VSAM GENKEY);
DECLARE field(10) ...;  /* assumed fixed length records */

/* intercept "invalid" start values */
    ON KEY(indexed)
        BEGIN;
        /* give up -- code to emulate FIND too complicated */
           PUT SKIP LIST('*** the key ', ONKEY(indexed),
                            ' is not a valid starting point');
        STOP;
        END;

/* detect end of file in case stop value too large */
    ON ENDFILE(indexed) GO TO endprinting;

/* determine filename, start and stop keys */
    GET LIST(filename, start, stop);
    OPEN FILE(indexed) TITLE(filename) INPUT;

/* find the first record to be printed */
    READ FILE(indexed) KEY(start) INTO(field);
    /* cannot get key of first record; use start */
        indexedkey = start;

/* print items 1, 2, 10 of each record with key <= stop */
    DO WHILE(indexedkey <= stop);
        PUT SKIP LIST(indexedkey, field(1), field(2), field(10));
        READ FILE(indexed) KEYTO(indexedkey) INTO(field);
        END;

endprinting:
    CLOSE FILE(indexed);
```

6) Merge two files; add field from f2 to records in f1.

```
DECLARE (f1, f2) FILE RECORD SEQUENTIAL;
DECLARE 1 rec1,       /* assumed fixed length record */
          2 a ...,
          2 b ...,
          2 c ...,
            ...
DECLARE 1 rec2,
          2 d ...,
            ...

/* detect end of file conditions on f1 and f2 */
    ON ENDFILE(f1) GO TO endprocess;
    ON ENDFILE(f2) GO TO markmissing;
```

```
/* add items from f2 to f1 */
    OPEN FILE(f1) UPDATE, FILE(f2) INPUT;
    DO WHILE('1'b);
        READ FILE(f1) INTO(rec1);
        READ FILE(f2) INTO(rec2);
        rec1.c = rec2.d;
        REWRITE FILE(f1) FROM(rec1);
        END;

markmissing:;
/* mark unmatched records from f1 with "missing" field */
    DO WHILE('1'B);
        rec1.c = 'missing';
        REWRITE FILE(f1) FROM(rec1);
        READ FILE(f1) INTO(rec1);
        END;

endprocess:
    CLOSE FILE(f1), FILE(f2);
```

Summary

A file system suitable for use in a highly interactive environment has been described and examples have been presented. The facilities provided allow for a wide variety of file formats and access patterns in a general manner. The emphasis has been on simplicity of use without limitation of function. Many of the provided capabilities are available in "production" file systems only at great expense and programmer ingenuity. Using these capabilities, a novice user should be able to construct and utilize file structures which would otherwise be beyond his grasp. The presentation has dealt entirely with facilities and their associated language constructs. Implementation is discussed in a separate report [1].

Acknowledgements

R. Conway and D. Krafft collaborated in the discussions that led to this file system and contributed significantly to its design.

References

1. Archer, J. E. Jr., "Implementation of an Unrestricted File Organization for Micro-PL/CS", Technical Report 79-367, Department of Computer Science, Cornell University.

2. Conway, R. W., A Primer on Disciplined Programming, Winthrop Publishing Co., 1978.

3. Conway, R. W. and R. L. Constable, "PL/CS -- A Disciplined Subset of PL/I", Technical Report 76-293, Department of Computer Science, Cornell University.

4. Conway, R. W. and D. Gries, Introduction to Programming, 3rd ed. Winthrop Publishing Co., 1979.

5. Sandewall, E., "Programming in an Interactive Environment: the LISP Experience", Computing Surveys, March 1978.

6. Teitlebaum, R. T., "The Cornell Program Synthesizer, a Micro-Processor Implementation of PL/CS", Technical Report 79- , Department of Computer Science, Cornell University.

7. OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, IBM GC33-0009.