COOKBOOK

Instructions for Implementing the Software Tools Package
(As distributed by the Software Tools Users Group)

Prepared by:

Debbie Scherrer
Advanced Systems Research Group
Computer Science and Mathematics Department
Lawrence Berkeley Laboratory
Berkeley, CA 94720

January 1981

LBID 098

TABLE OF CONTENTS

Part 1

Part 2

SUMMARY OF CONTENTS OF TAPE

1. TOOLS

ar	archive file maintainer
cat	concatenate and print text files
ch	change text patterns
comm	print lines common to two files
cpress	compress input files
crt	copy files to terminal
crypt	encrypt and decrypt standard input
date	print the date and time
dc	desk calculator
detab	convert tabs to spaces
diff	isolate differences between files
echo	echo command line arguments
ed	editor
edin	in-core editor
entab	convert spaces to tabs and spaces
expand	uncompress input files
fb	search blocks of lines for text patterns
field	manipulate fields of data
find	search a file for text patterns
format	format text
includ	file inclusion preprocessor
kwic	prepare lines for keyword-in-context index
lam	laminate files
ll	print line lengths
macro	general-purpose macro processor
mcol	multicolumn formatting
mv	move (rename) a file
os	convert backspaces into multiple lines
pl	print specified lines/pages in a file
pr	print file
ratfor	Ratfor preprocessor
rev	reverse lines
rm	remove (delete) files
roff	[see 'format']
sedit	stream editor
sh	command line interpreter
show	show all characters in a file
sort	sort and/or merge text files
spell	locate spelling errors
split	split file into pieces
tail	print last lines of a file
tee	copy input to output and named files
tr	character transliteration
tsort	topologically sort symbols
uniq	strip adjacent repeated lines from a file
unrot	unrotate lines prepared by kwic
wc	count lines, words, and characters in files
xref	make a cross reference of symbols

2. SUBROUTINES AND PRIMITIVES

(* indicates that the implementation of the routine is system-dependent
# indicates that the routine may, in some cases, be system-dependent)

definitions	standard Ratfor definitions

File Manipulation

#amove	move (rename) filel to file2
*close	close (detach) a file
*create	create a new file (or overwrite an existing one)
*gettyp	get type of file (character or binary)
*isatty	determine if file is a teletype/CRT device
#mkuniq	generate unique file name
*open	open an existing file for reading, writing, or both
*remove	remove a file from the file system

I/O

fcopy	copy file in to file out
*flush	flush output buffer for file 'fd'
getc	read character from standard input
*getch	read character from file
#getlin	get next line from file
*note	determine current file position
#prompt	prompt user for input
putc	write character to standard output
*putch	write character to file
putdec	write integer n in field width >=w
putint	write integer n onto file fd in field width >=w
#putlin	output a line onto a given file
putstr	write str onto file fd in field width >=w
*readf	read from an opened file
*remark	print single-line message
*seek	move read/write pointer
*writef	write to an opened file

Process Control

*spawn	execute subtask

String Manipulation

addset	put c in array(j) if it fits, increment j
addstr	add string s to str(j) if it fits, increment j
clower	fold c to lower case
concat	concatenate 2 strings together
ctoc	copy string-to-string
ctoi	convert string at in(i) to integer, increment i
ctomn	translate ascii control character to mnemonic
cupper	convert character to upper case
equal	compare strl to str2; return YES if equal
esc	map array(i) into escaped character, if appropriate
fold	convert string to lower case
gctoi	generalized character-to-integer conversion
getwrd	get non-blank word from in(i) into out, increment i

gitoc	generalized integer-to-character conversion
index	find character c in string str
itoc	convert integer to character string
length	compute length of string
lower	convert string to lower case
mntoc	ascii mneumonic to character
scopy	copy string at from(i) to to(j)
sdrop	drop characters from a string
skipbl	skip blanks and tabs at str(i)
stake	take characters from a string
stcopy	copy string at from(i) to to(j); increment j
strcmp	compare 2 strings
strim	trim trailing blanks and tabs from a string
substr	take a substring from a string
type	determine type of character
upper	convert string to upper case

Pattern Matching

amatch	look for pattern matching regular expression
getpat	encode regular expression for pattern matching
makpat	encode regular expression for pattern matching
match	match pattern anywhere on line

Command Line Handling

*delarg	delete command line argument number 'n'
*getarg	get command line arguments
gfnarg	get next filename argument
query	print command usage information

Dynamic Storage Allocation

dsfree	free a block of dynamic storage
dsget	obtain a block of dynamic storage
dsinit	initialize dynamic storage

Symbol Table Manipulation

delete	remove a symbol from symbol table
enter	place symbol in symbol table
lookup	get string associated with name from hash table
mktabl	make a symbol table
rmtabl	remove a symbol table
sctabl	scan all symbols in a symbol table

Date Manipulation

fmtdat	convert date to character string
*getnow	get current date and time
wkday	get day-of-week corresponding to month-day-year

Error Handling

cant	print 'name: can't open' and terminate execution
error	print single-line message and terminate execution

Miscellaneous

*endst	close all open files and terminate program execution
*initst	initialize all standard files and common variables

3. ADDITIONAL TOOLS AND LIBRARY ROUTINES

As assortment of tools and library routines including:

1) Alternate versions of tools included earlier on the tape
2) Tools requiring additional primitives
3) Experimental tools and routines
4) Other tools and routines not yet accepted as part of the basic package

4. COMPLETE DOCUMENTATION FOR TOOLS AND LIBRARY ROUTINES

5. PRIMERS

edit	editor
ratfor	ratfor preprocessor

6. SPELLING DICTIONARY

GUIDELINES FOR INSTALLING SOFTWARE TOOLS

Introduction

The purpose of this document is to provide a checkout scenario for installing an enhanced version of the Addison-Wesley Software Tools package developed by B. W. Kernighan and P. J. Plauger in conjunction with their book "Software Tools". Accompanying this document is a tape providing ratfor source code and documentation for enhanced versions of the original tools, as well as additional useful tools and a UNIX-like shell. (Unix is a registered trademark of Bell Labs...) This manual assumes you have read and understand the 'Software Tools' book and are at least vaguely familiar with the UNIX operating system concepts.

One of the purposes of the "Software Tools" experiment is to provide users of a multitude of operating systems with a portable set of common program development tools. These tools are made 'portable' via two mechanisms:

1. All source is written in ratfor, a fortran preprocessor language which is directly translatable into fortran.
2. Most system-dependent quantities are pushed down into "primitive" function calls, which are left up to the person in charge of bringing up the tools to implement.

This documentation is designed to assist the implementor of the ratfor preprocessor and primitives to bring up her version with as much ease as possible.

The first section of the manual contains step-by-step instructions for dealing with each of the files on the tape. The files have been arranged to allow you to develop your primitives in a reasonable order, while bringing up some useful tools at the same time. Along with a description of each of the files is a list of the primitives you'll need to develop to implement the file, a list of primitives you've already developed which the tool will need, and suggestions for implementation of the tool.

The second section of this manual gives detailed specifications for the design of your system-dependent primitives. Following these specifications as closely as possible when you write your primitives will help you bring up the tools with fewer problems.

Implementation Issues

The most difficult problems facing the software tools implementor are: character sets, passing command argument strings to a running program, random access to files, and (if the shell is desired) execution of subtasks.

Character sets: The main purpose of the tools is to provide a rational environment in which to do program development. We feel that accomplishing this requires a 128 character set as a minimum. However, the tools may be installed with a restricted character set if there is no alternative. If this is the case, we urge the implementor to at least develop some sort of escape conventions.

Passing command line arguments: Every system has a different (and invariably inadequate) way of accomplishing this. Often arguments are 'gratuitously' folded to a single case. Some systems even forbid "uninstalled" programs from reading their own arguments. Since the ability to read command line arguments is vital to tool utility, this problem will have to be faced early. In the absolute worst case, the arguments can be prompted for by the running program.

Random Access to Files: This capability is necessary only for running the editor. If random access is not possible, an in-core version of the editor is provided.

Execution of subtasks: The command line interpreter (the 'shell') will need to be able to spawn subtasks. Almost every system has the ability to roll executing programs in and out; however, many of them do not give the user easy access to this capability. To run the shell the implementor must devise some (perhaps devious) method of causing the execution of a desired task.

Format of the tape

There are 16 files on the tape. File 1 contains this document, which describes the remainder of the files.

If you look at the tape files you'll notice that most of the source code contains archive headers and trailers (that is, lines that begin with "#-h-" and "#-t-" respectively). We maintain all our sources with the archiver, making each routine a member of the source file. This file is in turn combined with the documentation and common block files to make one large working archive for each tool. Thus, each tool on the tape is an archived file containing the documentation, common blocks, and source code. These archives generally have the format:

```
main archive
      tool.doc        (documentation)
      cblock1         (common blocks)
      cblock2
      ...
      tool.r          (source code)
         rtn1         (each routine is a
         rtn2            sub-achive member)
         ...
```

Each archive header contains the file name, its size in characters, and the last date and time the member was changed. When you bring up the archiver on File 10, you can continue to maintain your source in this format.

ACKNOWLEDGEMENTS

The tools distributed on this tape represent a compilation of some of the most useful tools available from the University of Arizona, Georgia Institute of Technology, and Lawrence Berkeley Laboratory. I would like to especially thank Allen Akin (GT) and David Hanson (U. of Arizona) for their help in preparing these tools for distribution.

Depending upon the services your system provides, you can expect to spend anywhere from one week to several months developing the primitives for your tools. Good luck!

COPY (in Fortran)

DESCRIPTION

'Copy' represents the IO routines extracted from the ratfor bootstrap for easier testing. Copy reads an input file, converts the input characters to ascii strings via the routine 'inmap', converts them back to local format via 'outmap', and copies them to an output file. (Don't worry about the somewhat abstruse Fortran in which this tool is written; this is the output from the ratfor preprocessor and one generally never has to look at it.)

Copy also makes a simple call to "remark", a routine which receives a hollerith character string and sends it to the user's terminal.

CHANGES YOU MIGHT HAVE TO MAKE

The input file is defined as unit 5 and output as unit 6; change these in the READ and WRITE statements if they are different for your system. The end-of-file test is one commonly used, but not supported on all systems.

On some systems, the routine 'putch' needs an extra blank character at the beginning of each line when writing to certain devices such as terminals or printers. If your system has this 'feature', modify the write statement.

Look at the routine "remark". You'll most likely have to change the WRITE statement to print hollerith characters in whatever manner your system demands. Don't worry about finding the end of the hollerith array; simply print 20 or so characters. Later on, you will rewrite remark to handle strings gracefully.

You should attempt to run the alphabet (upper and lower cases if your system allows both), the digits, and all your special characters through copy to make sure they emerge as they should. If they don't, inspect the 'inmap'-'outmap' routines, which convert from a fortran character in 1H format to an integer representation of the ascii characters.

After you can properly read from and write to the users terminal, attempt to associate logical units 5 and 6 with physical file names. If your command language supports this, use it for now. Otherwise, rummage through your Fortran manual for terms like ASSIGN or OPEN, and modify copy accordingly for testing with disk files.

NEW PRIMITIVES TO WRITE

(inmap/outmap)

OTHER PRIMITIVES USED

None

ROUTINES NEEDED FROM OTHER TOOLS

None

RATFOR BOOTSTRAP

DESCRIPTION

The ratfor bootstrap, in fortran. The bootstrap contains most of the ratfor capabilities except for the 'include'. You'll use this bootstrap version for creating some simple tools and for developing your system primitives. The complete ratfor compiler will come later, after you can directly access files and perform more powerful IO.

Don't be overly concerned with the slowness of the bootstrap, which uses Fortran IO. When you implement the full ratfor, you will use your own, more efficient primitives which will speed up the processing.

CHANGES YOU MIGHT HAVE TO MAKE

Whatever changes you might have to make are determined by what your fortran compiler will or will not accept...

NEW PRIMITIVES TO WRITE

None

OTHER PRIMITIVES USED

There are dummy primitives provided in the bootstrap.

ROUTINES NEEDED FROM OTHER TOOLS

Combine the bootstrap with getch, putch, remark, inmap, and outmap from File 1.

SYMBOL DEFINITIONS, LIBRARY ROUTINES, AND TEMPORARY PRIMITIVES

DESCRIPTION

The fourth file contains the general symbol definitions, some generally useful library routines, and a set of temporary primitives which you can use to assist in developing your own primitives. They will be useful as a test program for the bootstrap and as a teaching aid to help you learn the ratfor language.

CHANGES YOU MIGHT HAVE TO MAKE

Take a look at the symbol definitions. Comments in it point you to symbols that might have to change for your system (e.g. FILENAMESIZE).

It is the "real" versions of the primitives that you will have to implement on your system. The rest of this manual is designed to simplify that procedure. However, right now, just try to get File 4 to run through the bootstrap and your fortran compiler.

NEW PRIMITIVES TO WRITE

You'll have to change the version of "remark" on this file to whatever you made it do on File 1.

OTHER PRIMITIVES USED

None

ROUTINES NEEDED FROM OTHER TOOLS

Use the 'inmap'/'outmap' pair from COPY.

READING COMMAND ARGUMENTS
ECHO and GETARG

DESCRIPTION

Now the real work begins. The tool 'echo' does nothing more than read the command line arguments passed to it and print them on the standard output (hopefully your terminal). However, for this tool you will have to implement 'getarg'. Find the design specifications for 'getarg' and 'delarg' in Section 2 of this manual and read them carefully.

CHANGES YOU MIGHT HAVE TO MAKE

The documentation for echo (and all the tools) precedes the source code. Remove it and store it someplace convenient. (All the documentation is also provided as a user's manual on File 14.)

Before compiling echo, you'll have to copy the general symbol definitions onto the front of the file.

The temporary primitives provide a version of getarg which prompts the user for the command line arguments. If this is the ONLY way you can implement getarg, then it will have to do. First test "echo" with these temporary versions, before attempting to create your own.

As you create your primitives, initialization routines will probably be necessary. We have called ours 'initst' and 'endst', and we have included them in the temporary primitives on File 4. We have also written the DRIVER macro to automaically call them for us. (If you can have your system automatically do the initialization, so much the better.) As you develop your 'getarg' and other primitives, insert into the initst routine any initialization which must be done to allow them to run.

Then, when writing your own getarg, look at the temporary primitives, especially 'makarg' and 'initst'. Makarg picks up the arguments and puts them in an array which 'getarg' subsequently reads. All you need do is change 'makarg' to pick up the command line from your system, convert it to ascii if necessary, and store it in the array. Then the rest of the code can remain unchanged.

Also, while looking at initst, notice that it calls getarg to look for file substitutions for the STDIN, STDOUT, and ERROUT files, which are generally the user's terminal. The files a user desires to substitute for are given as command line arguments preceded by a special flag. The flags for reassigned files are:

```
<infile
>outfile
>>outfile  (for appending)
```

```
?errfile
??errfile  (for appending)
```

where 'infile', 'outfile', and 'errfile' would be replaced with the name of the file desired. You should be aware of these file substitution capabilities, although they won't be completely operational until you bring up the file manipulation primitives.

NEW PRIMITIVES TO WRITE

Getarg - pick up command line arguments
delarg - delete argument number 'n'
(or, if possible, simply change 'makarg')

OTHER PRIMITIVES USED

Initst, endst, putch, getch, remark

ROUTINES NEEDED FROM OTHER TOOLS

The symbols file and library routines from file 4

THE CAT TOOL FOR TESTING THE FILE PRIMITIVES
Open, Create, Close
Getch, Getlin, Putch, Putlin, Remark

DESCRIPTION

Now is the time to begin developing your file manipulation primitives. The 'cat' (i.e. concatenate = copy) tool is provided for testing your versions.

This is the most critical step in the development of the tools at your site. The file primitives provide a mechanism for attaching to files from within running programs. Many operating systems already provide these utilities. In this case you simply need to design ratfor interfaces to them. However, more likely is the possibility that your operating system provides few or none of the capabilities you will need.

First, attempt to get cat to run using the temporary primitives open, create, close, getch, putch, and remark. Look at the code for these primitives to get an idea of what they should be doing. Notice that 'open' and 'create' simply set up a particular fortran unit for reading or writing. They assume you have assigned (in some manner) a file to these particular units. When you write your own primitives, you must be able to associate a file name with an IO channel, and set it up for reading and/or writing from within a running program.

Read the design specifications (Section 2 of this manual) for open, create, close, getch, and putch. Then sit down and carefully think through exactly what your versions will have to do. For instance, if you want to be able to handle local character sets as well as ascii, you will probably have to do your own block IO. Remember, too, that you will need a certain amount of random IO capabilities when you bring up the editor so you might have a glance at the descriptions for seek and note.

You will also have to teach 'remark' to find the end of a hollerith array. If your system provides this capability, fine. Otherwise, have 'remark' look for a period (.) as the end marker for the string. All hollerith arrays in the tools source code end with a dot.

You will probably have to set some limit to the maximum number of files which can be open at a time. 10 - 15 seems to be a good range.

CHANGES YOU MIGHT HAVE TO MAKE

Extract the documentation, and copy the general symbols file onto the front of the source code.

Insert into 'initst' any initialization that must be done. Also, insert into 'endst' the code necessary to close any files that have been opened. It's best to do this even if your system automatically closes all files at the end of a job.

You will most certainly have a number of symbol definitions to create for your primitives. You might like to keep these separate from the general definitions, on a file which can be "included" when necessary.

You will need to do quite a lot of testing on your primitives. Make sure reads on empty files work correctly; make sure you can create a file that already exists; and, test all the boundary conditions you can think of.

Plan on at least one or two major rewrites of your primitives in the future. Don't try for great efficiency right now, just get something that works.

NEW PRIMITIVES TO WRITE

Open - open an existing file for reading, writing, or both
create - create a new file or overwrite an existing one
close - close (detach) a file
getch - read a character from a file
putch - write a character onto a file
remark - write a hollerith string to ERROUT

OTHER PRIMITIVES USED

Initst
endst
getarg

ROUTINES NEEDED FROM OTHER TOOLS

You can use any of the library routines provided on file 4.
inmap and outmap from file 2

FILE INSERTION
INCLUD Tool

DESCRIPTION

You are actually now ready to bring up the ratfor in ratfor compiler itself. However, your task will be easier if you implement the 'includ' tool first so that you can easily include the common blocks needed in the preprocessor source code.

This file will also be a good test for your primitives.

CHANGES YOU MIGHT HAVE TO MAKE

Remove the documentation and copy the general symbol definitions file to the beginning of includ before compiling it.

Adjust the symbol MAXOFILES in the general symbol definitions to match the maximum number of opened files allowed by your primitives.

NEW PRIMITIVES TO WRITE

None

OTHER PRIMITIVES USED

open
close
getlin
remark
getarg
initst
endst

ROUTINES NEEDED FROM OTHER TOOLS

Library routines from file 4

RATFOR PREPROCESSOR

DESCRIPTION

File 8 contains the symbol definitions, included common blocks, and source code for the ratfor preprocessor. This version is one developed from the original by David Hanson of the University of Arizona and enhanced by Joe Sventek and Debbie Scherrer of Lawrence Berkeley Laboratory and Allen Akin of the Georgia Institute of Technology. It includes a hash table for searching through definitions, plus a full macro processor, the 'string' declaration, long variable names, and a few other goodies. If you've brought up 'includ', extract the common blocks onto a file named 'commons'. If you don't have 'includ', you'll have to insert the common blocks by hand.

CHANGES YOU MIGHT HAVE TO MAKE

This version of ratfor automatically opens and includes the file containing the general symbol definitions. Set the definition STDEFNS in the source code to the name of the file you are using for your symbols. (If the file resides on a particular directory, don't forget to include that in the filename.) For example,

```
define(STDENFS,"ratdef")
```

(The quotes must be included.) If you don't want this feature, set the definition to:

```
define(STDEFNS,"")
```

Take a final look at 'remark' in your primitives to make sure you've taught it to look for a period at the end of a quoted hollerith string.

The major problems you will probably run into are character sets. If you can't pass the braces '{' and '}' in, you can use '[' and ']' respectively. If your fortran compiler can only process upper case characters, set the definition UPPERC this way:

```
define(UPPERC,)
```

Look at 'inmap' and 'outmap' for any other character problems you run into.

NEW PRIMITIVES TO WRITE

None

OTHER PRIMITIVES USED

open
close
getarg
getch
putch

```
     remark
     initst
     endst
```

ROUTINES NEEDED FROM OTHER TOOLS

```
     Library routines
```

IN-CORE ED

DESCRIPTION

At this point you are ready to bring up most of the rest of the tools. There are still a few more primitives to write, but the order in which they are done is not critical.

If your system has not provided you with an editor of any use, you might want to bring up the in-core editor now, to assist you in implementing the rest of the tools. You'll find it on the first part of file 11. The in-core editor does not require any more primitives than those you already have.

TEXT FORMATTING

DESCRIPTION

This is the source code for the text formatter (often nicknamed 'roff'). Although you are now ready to bring up many of the other tools, it might be advantageous to implement the text formatter first, so that it can process the documentation provided on File 14. Once again, extract all the common block files and put them on files with the names indicated.

You might want to use File 14 for testing.

CHANGES YOU MIGHT HAVE TO MAKE

This implementation of the formatter can either put out a line-feed character (control-l) to indicate the beginning of a page, or it can count lines as the version in the book does. Decide which would be more appropriate for the devices you will be printing output on, and then, if you desire the control-l, set the following definition:

```
define(PAGECONTROL,)
```

If this definition doesn't appear, format will count lines.

This formatter has a mechanism which allows the user to have the formatter stop before printing each page so that a new sheet of paper can be inserted. The formatter will attempt to open a channel to the user's teletype using the definitions of TERMINAL_IN and TERMINAL_OUT (in the general symbols file) as the names of the input and output channels respectively. Make sure you have set these to the appropriate file names.

The primitive 'flush' will be needed to send a line to the user's teletype while suppressing the carriage return/line feed sequence. If you can't implement flush, simply send a NEWLINE to the output file (which will actually cause the buffer to be flushed but won't suppress the cr/lf.)

If possible, implement the primitive 'getnow', used to pick up the current date and insert it into any header or footer where a percent sign (%) occurs.

You'll eventually want to implement the tool 'lpr' (system-dependent and thus not provided on the tape). 'Lpr' is a combination of the tools 'os' (overstrike) and 'detab', plus some sort of mechanism to spool a file for printing. It might also have to do some carriage control to make sure the formatter output aligns on page boundaries.

NEW PRIMITIVES TO WRITE

flush - flush output buffer

getnow - get current date and time (optional)

OTHER PRIMITIVES USED

getarg
open
close
create
getch
putch
remark
initst
endst

ROUTINES NEEDED FROM OTHER TOOLS

The library routines

FILE ARCHIVING

DESCRIPTION

This is the source code for 'ar', the file archiving tool. The archiver is an extremely useful tool for maintaining source code, documentation, and files of files. It also does quite a bit of IO so will be a lovely test for your primitives.

Two versions of the archiver have been included in this file. The first was written by Allen Akin at Georgia Tech. It delimits archive members by preceding each with a header of the format:

```
#-h- filename size type date time
```

and following each with a trailer exactly the same as the header except beginning with "#-t-". Archives are searched by comparing headers and trailers. The size of the file (in characters), date and time are kept only for the user's convenience. All files on this tape are maintained by this archiver.

The second version is an enhanced version of the archiver described in "Software Tools". It separates archive members with the same header as the Akin version, but relies upon the size given to locate the end of the member.

The first version has the advantage that one can edit archive files directly without destroying their integrity. The second version has the advantage that it can be adapted to be used with binary files. Choose whichever you wish.

CHANGES YOU MIGHT HAVE TO MAKE

The primitive 'mkuniq' is needed to generate a scratch file name unique to the process. This is needed to avoid conflicts when several users are logged in under the same account or directory. The archiver passes a string of characters to 'mkuniq', which in turn might append to them the process ID or some other unique identifier. If you cannot pick up a process ID or cannot generate unique file names in any way, or if multiple users aren't a problem on your system, simply have 'mkuniq' return the character string passed to it.

The primitive 'remove' is used to delete the scratch file after it has been used. Read the specifications in Section 2 for implementation details.

The library routine 'amove' is used to copy the archive scratch file back to the original after all changes have been made. It is currently implemented as a copy-remove operation, but if your system provides a renaming feature you should use that instead.

You might have a look at the routine 'gettyp'. 'Gettyp' is a function which determines a files's type--local character, ascii, or binary. This information is stored in the archive header only for the user's convenience. In its current form, 'gettyp' returns the file type LOCAL. If your system has a way for you to determine a file's type, you might want to teach 'gettyp' to return the correct information. ('Gettyp' is also needed by the shell, if you intend to bring that up.)

NEW PRIMITIVES TO WRITE

remove - remove a file from the file system
amove - move (rename) a file (optional)
mkuniq - get scratch file name (optional)
gettyp - determine a file's type (character or binary) (Optional)

OTHER PRIMITIVES USED

getarg
getnow (optional)
open
create
close
getch
putch
remark
initst
endst

ROUTINES NEEDED FROM OTHER TOOLS

The library routines

The Editor

DESCRIPTION

Two versions of the editor have been included on this file. The first version is the in-core editor from the "Software Tools" book. It is provided for those unfortunates who cannot implement random IO on their systems. If you can implement random IO, choose the second version.

CHANGES YOU MIGHT HAVE TO MAKE

You'll have to implement the two random IO primitives--seek and note. Read their descriptions carefully. Two words have been allotted for the address returned by 'note'. You may not need this many but several systems do so space has been allowed for them.

You will probably be interested in fine-tuning the editor a bit for your own system. On the random access version of the editor, look at the routines 'setb' and 'getb'--they pick up and store information in the line pointer array. Four items of information are kept about each line: pointer to next line, pointer to last line, mark (for global commands), and seek address (2 words). Each piece of information is kept in a separate word, but you might like to pack them into fewer bits. If you do this, adjust the symbol 'BUFENT', which sets the number of words needed for each line.

You'll probably want to adjust the symbol MAXBUF, which determines the maximum length of the line pointer array.

NEW PRIMITIVES TO WRITE

note - determine current file position
seek - move read/write pointer to position specified

OTHER PRIMITIVES USED

getarg
getch
open
create
putch
remark
remove
mkuniq
initst
endst

ROUTINES NEEDED FROM OTHER TOOLS

The library routines

OTHER TOOLS

DESCRIPTION

Here are most of the rest of the tools, each included as a separate member of an archive.

Note that a few of the tools require common blocks and definitions already provided for other tools on the tape. We've included them twice, but make sure that if you've made any changes to the previous ones, you change these copies as well.

CHANGES YOU MIGHT HAVE TO MAKE

If you've implemented your primitives properly, all these tools should come up with few problems.

Finish up any primitives you haven't written, reading the design specifications in Section 2.

NEW PRIMITIVES TO WRITE

whatever you haven't completed (except 'spawn')

OTHER PRIMITIVES USED

getarg
getch
putch
remark
open
close
create
initst
getnow
isatty
endst
remove
mkuniq

ROUTINES NEEDED FROM OTHER TOOLS

The library routines

The Shell

DESCRIPTION

Ah, here is the piece-de-resistence: the UNIX-like shell.

CHANGES YOU MIGHT HAVE TO MAKE

You'll have to implement the primitive 'spawn'. Read the description in Secion 2 very carefully. You may have to alter your original version of 'getarg' (or 'makarg') so that it can read arguments passed via 'spawn'. If you cannot implement background processes, disable the 'doampr' routine.

Have a look at the library routine 'prompt'. It is used to output a string (such as '% ') to the user's terminal, suppressing the carriage-return/line-feed sequence. It then reads input from the user. 'Prompt' expects to be able to write to the user's terminal via the channel descriptor passed to it. If this cannot be done on your system, adjust prompt to open a separate channel to the teletype.

If you haven't already implemented 'gettyp', try it again now. The shell uses 'gettyp' to determine whether a command is a binary executable file or character script file containing further commands. If you absolutely cannot find a way to tell character files from executable code, then the user will have to explicitly execute shell scripts by saying:

```
% sh scriptname args ...
```

You might also want to look at the routine 'loccom', which searches a series of directories when attempting to locate commands. You might want to adjust it for your system.

NEW PRIMITIVES TO WRITE

spawn - execute a subtask
gettyp - determine type of file (character or binary)
prompt - issue prompt to user and read input

OTHER PRIMITIVES USED

close
create
delarg
endst
getarg
getch, getlin
initst
open
putch, putlin, remark
remove
mkuniq

ROUTINES NEEDED FROM OTHER TOOLS
    The library routines

Documentation

DESCRIPTION

Here is the input source for the software tools programmer's manual, in a format designed to be sent to the text formatter.

Notice that it is an archived file. To produce the documentation for, say, ratfor, the user would specify:

```
ar p manual ratfor | format
```

(Or, "ar p manual ratfor | format | crt")

To print the entire manual, the user might say:

```
ar p manual | format | lpr
```

(where 'lpr' is a combination of 'os' and 'detab', plus whatever is necessary to spool a file to the printer).

CHANGES YOU MIGHT HAVE TO MAKE

Change anything you'd like.

Optional Tools

DESCRIPTION

This section contains tools which may require additional, special-purpose primitives, or which have been submitted for distribution without extensive testing or alteration. These tools have been included on the tape exactly as submitted. Each is included as a member of the archive.

CHANGES YOU MIGHT HAVE TO MAKE

????

NEW PRIMITIVES TO WRITE

Hopefully, each tool will provide not only documentation for the tool itself, but also instructions for writing any necessary new primitives.

OTHER PRIMITIVES USED

probably all of them

ROUTINES NEEDED FROM OTHER TOOLS

Who knows...

Spelling Dictionary

DESCRIPTION

Here are about 42,000 words for you. We use the dictionary for our 'spell' tool, but it's also useful for game shows, cross-word puzzles, etc.

The dictionary is in sort order (of course), all lower case, and with one word per line.

CHANGES YOU MIGHT HAVE TO MAKE

Add to it all you like.

SPECIFICATIONS FOR SYSTEM-DEPENDENT PRIMITIVES

This part of the cookbook contains detailed specifications to be used in the design and implementation of the software tools system-dependent primitives.

OVERVIEW OF SOFTWARE TOOLS PRIMITIVES

(The '#' indicates that, on some systems, the routine may be written in terms of the other primitives.)

FILE ACCESS

```
        open - open an existing file for reading, writing, or both
        create - create a new file (or overwrite an existing one)
        close - close (detach) a file
        remove - remove a file from the file system
       #amove - move (rename) file1 to file2
        isatty - determine if file is a teletype/CRT device
        gettyp - determine type of file (character or binary)
```

I/O

```
        getch - read character from file
       #getlin - read next line from file
        putch - write character to file
       #putlin - write a line to a file
       #prompt - write to/read from teletype; suppress cr/lf
        remark - print single-line message
        seek - move read/write pointer
        note - determine file position of next record to be read/written
        readf - read 'n' bytes/words from file
        writef - write 'n' bytes/words to file
        flush - flush output buffer
```

MISCELLANEOUS

```
        getarg - get command line arguments
        delarg - delete command line argument 'n'
        initst - initialize all standard files and common variables
        endst - close all open files and terminate program
        mkuniq - generate unique file name
        getnow - get current date and time
        spawn - execute subtask
```

NAME

amove - move (rename) file1 to file2

SYNOPSIS

```
stat = amove (name1, name2)

character name1(FILENAMESIZE), name2(FILENAMESIZE)
integer stat returned as OK/ERR
```

DESCRIPTION

Amove moves the contents of the file specified by name1 to the file specified by name2. It is essentially a renaming of the file. If the file could be moved properly, OK is returned. If there were problems either creating the new file or deleting the old one, ERR is returned.

Both file names are ascii character strings terminated with an EOS marker.

The files need not be connected to the running program to be renamed.

IMPLEMENTATION

Amove primarily exists to change the name of a file, such as when moving the archive scratch file back to the original. If possible, this should be implemented with a "rename" primitive in the local operating system. If this capability isn't available, amove could be implemented as a copy/delete combination.

Amoves from different devices will most likely have to be implemented as copy/remove operations.

If the system supports naming conventions for devices such as TTYs, then amoving a file to a TTY should copy the file to the TTY and then remove it.

SEE ALSO

fcopy, remove

DIAGNOSTICS

None

NAME

close - close (detach) a file

SYNOPSIS

```
call close (fd)

filedes fd
```

DESCRIPTION

Close disassociates file descriptor "fd" from the opened file to which it refers. If "fd" is the only descriptor referring to that file, all pending I/O is completed and the file is closed. If "fd" does not refer to an opened file, close simply returns. "fd" is an internal file descriptor as returned from an open or create call.

IMPLEMENTATION

Close breaks the connection between the program and a file accessed via open or create. If necessary, the file's write buffer is flushed and the end of the file is marked so that subsequent reads will find an EOF. If a file has been opened multiple times (that is, more than one internal descriptor has been assigned to a file), care is taken that multiple closes will not damage the file.

SEE ALSO

open, create, endst

DIAGNOSTICS

If the file descriptor is in error, the routine simply returns.

NAME

create - create a new file (or overwrite an existing one)

SYNOPSIS

```
fd = create (name, access)

character name(FILENAMESIZE)
integer access
filedes fd - returned as a file descriptor/ERR
```

DESCRIPTION

Create creates a new file whose name is contained in "name" and then opens it for I/O according to the value of "mode", as if open had been called (see "open"). If the file already exists, it is truncated and prepared for overwriting.

If the creation succeeded, create returns a file descriptor which is used to refer to the file in subsequent I/O calls. If the file could not be created, ERR is returned.

IMPLEMENTATION

Create creates a new file from within a running program and connects the external name of the file to an internal identifier which is then usable in subsequent subroutine calls. If the file already exists, the old version is removed or truncated to 0 length and overwritten. All other functions are similar to open.

On some systems a default character type (ASCII or LOCAL) is assigned to a newly-created file.

SEE ALSO

open, close

DIAGNOSTICS

The function returns ERR if the file could not be created or if there are already too many files open.

NAME

delarg - delete command line argument number 'n'

SYNOPSIS

```
call delarg (n)

integer n
```

DESCRIPTION

Delarg deletes the "n"th command line argument, if it exists. After a successful call to delarg, calls to getarg behave as though the deleted argument had never been specified.

IMPLEMENTATION

Delarg works in conjunction with 'getarg'. It generally re-orders indices to an array holding the command line arguments.

SEE ALSO

getarg, initst

DIAGNOSTICS

If argument 'n' does not exist, delarg simply returns.

NAME

endst - close all open files and terminate program execution

SYNOPSIS

```
call endst
```

DESCRIPTION

Normally called at the end of any ratfor program or program which uses the software tools primitives. Closes all open files and terminates program execution.

On many systems a call to endst is made automatically, either by the system or by specifically inserting the call into code processed by the ratfor preprocessor.

IMPLEMENTATION

Any open files are closed. If any files have been opened multiple times (that is, they have more than one internal descriptor assigned to them), care is taken that multiple closes do not damage the file.

SEE ALSO

close, initst

DIAGNOSTICS

None

NAME

flush - flush output buffer for file 'fd'

SYNOPSIS

```
call flush (fd)

filedes fd
```

DESCRIPTION

Flush assures that any remaining characters in the output buffer of the file specified by "fd" are sent out to the file. It is useful for sending lines to a teletype-like device without requiring a NEWLINE character, and also for flushing buffers after calls to "writef".

IMPLEMENTATION

It is expected that most software tools installations will employ some form of buffered I/O. Flush is intended to define the buffer-clearing operation that takes place before file closing, and to provide a means of insuring that output directed to a terminal has appeared on that terminal (e.g. before obtaining some input after a prompt). On systems with unbuffered I/O, flush is a no-op.

SEE ALSO

prompt, writef, putch, putlin

DIAGNOSTICS

None

NAME

getarg - get command line arguments

SYNOPSIS

```
stat = getarg (n, array, maxsize)

integer n, maxsize
character array(ARB)
integer stat returned as length/EOF
```

DESCRIPTION

Getarg returns the "n"th argument to the current program in the array "arg", one character per array element. The argument is terminated by an EOS marker. 'Maxsize' is passed as the maximum number of characters array is prepared to deal with (including the EOS); getarg truncates the argument if necessary. Getarg returns the length of the argument in "arg" (excluding the EOS), or EOF if "n" specified a non-existent argument.

On some systems, if "n" is zero, the name of the current program is returned in "arg" and, if "n" is -1, the function returns the number of arguments on the command line.

Also, on some systems, command line arguments can only be passed in a single case (upper or lower). On these systems an escape mechanism may be necessary to indicate case when specifying arguments.

IMPLEMENTATION

The implementation of 'getarg' may be quite different on different operating systems. Some systems allow only upper case (or lower case) on the command line; they may limit size; they may not even provide access at all without considerable contortions.

When implementing 'getarg', the designer should keep in mind that a 'delarg' will also be needed. One possible design would be to create a routine 'makarg', which would pick up the arguments from the system, convert them to ascii strings, handle any upper-lower case escape conventions, and store them in an array. 'Getarg' could then access this array, stripping off any quoted strings surrounding the arguments, and passing them along to the user. 'Delarg' could also access this array when removing reference to arguments.

If it is absolutely impossible to pick up command line arguments from the system, 'makarg' could be taught to prompt the user for them.

If the shell is implemented, 'getarg' (or perhaps 'markarg') will have to be altered to read arguments as passed from the shell.

SEE ALSO

initst, delarg

DIAGNOSTICS

None

NAME

getnow - determine current date and time

SYNOPSIS

```
subroutine getnow (now)
integer now (7)
```

DESCRIPTION

'Getnow' is used to query the operating system for the current date and time. The requested information is returned in a seven-word integer array, where: word 1 contains the year (e.g. 1980); word 2 contains the month (e.g. 9); word 3 contains the day (e.g. 25); word 4 contains the hour (e.g. 13); word 5 contains the minute (e.g. 39); word 6 contains the second (e.g. 14); word 7 contains the millisecond (e.g. 397).

The information returned by 'getnow' may be used as-is or further useful processing may be done by 'fmtdat' or 'wkday'.

IMPLEMENTATION

Operating systems generally have some mechanism for picking up the current date and time. If yours has one, use it.

Getnow is not critical to the implementation of the tools and can be left as a stub if the operating system cannot supply the needed information.

ARGUMENTS MODIFIED

now

BUGS

Some systems cannot obtain all the time information described. Array elements that cannot be filled default to zero.

SEE ALSO

fmtdat (2), wkday (2), date (1)

NAME

getch - read character from file

SYNOPSIS

```
c = getch (c, fd)

character c
filedes fd
```

DESCRIPTION

Getch reads the next character from the file specified by fd. The character is returned in ascii format both as the functional return and in the parameter c. If the end of a line has been encountered, NEWLINE is returned. If the end of the file has been encountered, EOF is returned.

IMPLEMENTATION

Interspersed calls to getch and getlin work properly. A common implementation is to have getlin work by repeated calls to getch.

If the input file is not ascii, characters are mapped into their corresponding ascii format via a routine called "inmap".

Getch is able to recognize an end-of-file marker from either a terminal or a file.

SEE ALSO

getc, getlin, putch, putlin, readf, writef

DIAGNOSTICS

None

NAME

getlin - get next line from file

SYNOPSIS

```
stat = getlin (line, fd)

character line(MAXLINE)
filedes fd
integer stat returned as length/EOF
```

DESCRIPTION

Getlin reads the next line from the file opened on file descriptor "fd" into the ascii character array "line". Characters are copied until a NEWLINE character (or other end-of-record marker) is found or until MAXLINE characters have been copied. A NEWLINE character is returned whenever an end-of-line marker has been sensed and the entire string is termined with an EOS.

If the line is longer than MAXLINE characters, some systems truncate the line to MAXLINE, while other systems return the remainder of the line in the next call to getlin.

Getlin returns EOF when it encounters an end-of-file, and otherwise returns the line length (excluding the EOS).

Interspersed calls to getlin and getch are allowed and work properly.

IMPLEMENTATION

Getlin reads characters either directly from a file or from an internal buffer set up when the file was opened. When an end-or-record is encountered (by whatever means the system does that sort of thing), a NEWLINE character is returned by getlin. If the file contains characters in a representation other than ascii, the characters are mapped (via inmap) into their internal ascii representation.

Getlin generally assumes a maximum size of the array line passed to it (MAXLINE). If the input line exceeds the limit, either truncate the line or return the rest of it in subsequent calls to getlin.

Getlin and getch are compatible; that is, interspersed calls to getlin and getch are allowed and work properly. A common implementation is to have getlin call getch until a NEWLINE character is found (or MAXLINE is reached).

Getlin is able to recognize end-of-file marks from both terminals and files.

SEE ALSO

getch, putch, putlin

DIAGNOSTICS

None

NAME

gettyp - get type of file (character or binary)

SYNOPSIS

```
type = gettyp (name)

character name(FILENAMESIZE)
integer type returned as ASCII, LOCAL, BINARY
```

DESCRIPTION

'Gettyp' determines whether the file specified by 'name' is ascii characters, local characters (if different from ascii), or binary. The type is returned as ASCII, LOCAL, or BINARY in the functional call.

IMPLEMENTATION

A file's type may be determined by locating system information about the file, or 'gettyp' might have to actually open the file and read part of it, making a reasonable 'guess' as to its flavor.

The shell uses 'gettyp' to determine whether a command verb given by the user represents a script file or an executable tool. If the file turns out to be a character (i.e. script) file, the shell then spawns itself with the file as input. Thus, if 'gettyp' could not be reliably implemented on a particular system, the user would have to specifically execute her script files by:

```
% sh script ...
```

'Gettyp' may also be called by the archiver to store a file's type in the archive header (for informational purposes only).

This primitive is not considered finalized. Most likely, another primitive will be specified which is used to pick up an assortment of information about a file. 'Gettyp' is being used temporarily until the final version is specified.

SEE ALSO

DIAGNOSTICS

ERR is returned if the file does not exist

NAME

initst - initialize all standard files and common variables needed for the software tools primitives

SYNOPSIS

call initst

DESCRIPTION

This routine is generally the first routine called by any program desiring to use the software tools primitives. It opens STDIN, STDOUT, and ERROUT files, performing any file substitutions indicated on the command line. It also prepares the list of arguments needed by getarg and sets up any buffers, variables, etc. needed by the software tools primitives.

On many systems, the calls to 'initst' and 'endst' are done automatically either by having the ratfor preprocessor insert them into the code, or by having the system itself call them before executing the user's program.

IMPLEMENTATION

'Initst' initializes any common blocks, variables, buffers, arrays, or whatever is necessary to allow the other software tools primitives to operate. It may also have to retrieve (via 'makarg') the list of command arguments passed to the program, if this is not automatically available from the operating system.

'Initst' is also responsible for parsing the command line to determine if there have been any file substitutions for STDIN, STDOUT, or ERROUT. The appropriate files (either the user's terminal or the substitutions) are then opened and properly positioned. Arrangements are made so that 'getarg' won't pick up standard file substitution flags on subsequent calls (probably by a call to 'delarg').

SEE ALSO

endst, getarg, delarg

DIAGNOSTICS

If initst cannot function for some reason, the program generally aborts (possibly without an error message since the standard error file may not have been opened).

NAME

isatty - determine if file is a teletype/CRT device

SYNOPSIS

```
stat = isatty (fd)

filedes fd
integer stat returned as YES/NO
```

DESCRIPTION

This function returns YES if the file specified by 'fd' is a teletype-like device, otherwise it returns NO. 'Fd' is a file descriptor returned by a call to open or create.

IMPLEMENTATION

When a file is opened, a flag is usually set indicating what device the file is associated with. This function generally reads that flag. Other implementations are possible, depending upon the operating system involved.

'Isatty' is generally used by the tools to determine whether they should issue prompts or not.

SEE ALSO

open, create

DIAGNOSTICS

NO is returned if 'fd' is in error.

NAME

mkuniq - generate unique file name

SYNOPSIS

```
len = mkuniq (seed, name)

character seed(ARB), name(FILENAMESIZE)
integer len returned as length/ERR
```

DESCRIPTION

Mkuniq generates a "unique" file name from a given seed string. This name is intended for use in subsequent calls to create and open. "Len" is returned as the number of characters in "name", not including the EOS marker. If there was some problem in creating the name, ERR is returned.

Mkuniq is generally used for generating scratch file names, such as those needed by the editor and archiver.

On single-user systems or others where the unique naming of scratch files is not important, mkuniq simply returns "seed". More sophisticated versions may construct a file name in a special directory, use process ids or time-and-date to insure uniqueness.

IMPLEMENTATION

'Mkuniq' is used to avoid conflicts which occur when more than one user is logged in under a single user or directory name. The optimum implementation would be to return an absolutely unique file name based on 'seed'. However, on most systems this is impossible. Another solution would be to append (or prepend) some sort of process identifier to 'seed', thus making the file name at least unique to the calling process. To avoid privilege violations it might also be necessary to choose a specific directory for all scratch files, with appropriate privileges being assigned to it.

On some systems, 'seed' is limited to a certain number of characters.

On single-user systems, systems with local files, or other circumstances where unique file names are not a problem, 'mkuniq' can simply return the 'seed' as 'name'.

SEE ALSO

DIAGNOSTICS

If a file name could not be generated, ERR is returned.

NAME

note - determine current file position

SYNOPSIS

```
stat = note (offset, fd)

integer offset(2)
filedes fd
integer stat returned as OK/ERR
```

DESCRIPTION

Note determines the current value of a file's read/write pointer. The argument "offset" is a 2-word integer array that will receive the information. Offset is maintained untouched by the user and passed to "seek" when desiring to return to that particular location in the file.

Note is usually used as the file is being written, picking up the pointer to the end of the file before each record is inserted there.

On text files (e.g. those created by calls to putch, putlin), note is guaranteed to work at line boundaries only. However, it should work anywhere on a file created by calls to writef.

IMPLEMENTATION

Note is compatible with whatever implementation is chosen for seek and the opening of files at READWRITE access.

Offset is a two-word integer in which is stored a character count, word address, block and record address, or whatever is appropriate for the local operating system. Note should be taught to return BEGINNING_OF_FILE and END_OF_FILE where appropriate.

In the editor, note is called to locate the end of file for subsequent writes.

SEE ALSO

seek, readf, writef

DIAGNOSTICS

None

NAME

open - open an existing file for reading, writing, or both

SYNOPSIS

```
fd = open (name, access)

character name(FILENAMESIZE)
integer access
filedes fd - returned as file descriptor/ERR
```

DESCRIPTION

Open opens the file whose name is contained in "name" for I/O according to the value of "mode", which may be READ, WRITE, READWRITE, or APPEND. If the file exists and can be opened according to "mode", open returns a file descriptor. If the file cannot be opened, ERR is returned.

After a file is opened, it is positioned at the beginning, unless APPEND access is requested, in which case the file is prepared for extension.

Opening the same file for reading more than once is permissible and works correctly. However, on many systems a file may be opened only once in WRITE, APPEND, or READWRITE mode.

There is generally a limit to the number of files that can be opened simultaneously. This number is specified by the definition MAXOFILES in the general symbol definition file.

IMPLEMENTATION

Open attaches an existing file to a running program and associates the external file name with an internal identifier which is then usable by the program. The file is opened for I/O according to the value of "mode", where mode may be READ, WRITE, READWRITE, or APPEND. "Name" is passed as an ascii character array, stored one character per array element. The access modes READ, WRITE, READWRITE, and APPEND are global symbols defined in the standard definitions file.

Open does whatever manipulations are necessary to allow reading and/or writing to the file. An internal descriptor (usually an integer) is assigned to the file and subsequently used when calling other primitives such as close, getch, putch, getlin, and putlin.

'Open' should be able to open a channel to the teletype in responce to the filenames defined by TERMINAL_IN and TERMINAL_OUT. It also might be taught to respond to other device names where appropriate.

Open may have to set up an internal I/O buffer for the file. It may also have to determine the file's type (teletype, character file, binary file). Information about the file's type and teletype characteristics (yes or no) is generally maintained and made available to the user via "isatty" and possibly other file characteristics primitives.

Open is sometimes taught to read characters of ascii type as well as local character type (if not ascii). Translation of characters from local to ascii is done when the characters are passed to getch and getlin.

Opening a fresh instance of an already opened file is permissible and does not affect the position of the file as accessed by subsequent or previous calls.

There is generally a limit to the maximum number of files open at any one time. 10-15 is a common range.

READWRITE access may cause problems, or even be impossible on many systems. The only tool which needs this access is the editor. If necessary, READWRITE access may be implemented by opening the file twice--once at READ and once at WRITE access.

SEE ALSO

create, close, remove, getch, putch, readf, writef, seek, note, isatty

DIAGNOSTICS

Open returns ERR if the file does not exist, if one of the necessary directories (if any) does not exist or is unreadable, if the file is not readable/writeable, or if too many files are open.

NAME

prompt - prompt user for input

SYNOPSIS

```
call prompt (str, buf, fd)

character str(ARB), buf(MAXLINE)
filedes fd
```

DESCRIPTION

Prompt determines if "fd" refers to a teletype-like device and, if so, writes the prompt string "str" to the TTY, and flushes its output buffer to insure the prompt is printed. A line of input is then read from fd by "getlin".

No carriage return/line feed sequence is done unless specified by a NEWLINE in the prompt string.

IMPLEMENTATION

The version of 'prompt' on the tape is essentially:

```
          if (isatty(fd) == YES)
               {
               call putlin(str, fd)
               call flush (fd)
               stat = getlin (buf, fd)
               }
```

Note that prompt expects to be able to read from and write to 'fd'. If this is not possible on your system, modify prompt to open a separate channel to the teletype for the write.

SEE ALSO

putlin, remark, flush, isatty

DIAGNOSTICS

None

NAME

putch - write character to file

SYNOPSIS

```
call putch (c, fd)

character c
filedes fd
```

DESCRIPTION

Putch writes the character c onto the file specified by file descriptor "fd". If c is the NEWLINE character, the appropriate action is taken to indicate the end of the record on the file. The character is assumed to be in ascii format; however, if the output file is not of ascii type, the necessary conversion is done.

IMPLEMENTATION

Interspersed calls to putch and putlin work properly. One implementation is to have putlin perform repeated calls to putch.

If the output file is not ascii, characters are mapped into their corresponding format via the routine outmap.

SEE ALSO

putc, putlin, getch, getlin, readf, writef

DIAGNOSTICS

None

NAME

putlin - output a line onto a given file

SYNOPSIS

```
call putlin (line, fd)

character line(ARB)
filedes fd
```

DESCRIPTION

Putlin writes the characters in "line" to the file opened on file descriptor "fd". If a NEWLINE character is located, appropriate action is taken to indicate the end-of-record in whatever format is necessary for the local operating system. If no NEWLINE character is specified, no carriage return (or end-of-record) is assumed. This probably means that the output buffer will not be flushed.

Any necessary character translation is done if the output file is not of ascii type.

IMPLEMENTATION

Putlin should write the line onto the file and, if a NEWLINE is encountered, do whatever is necessary to indicate to the local operating system that a record has been generated. If the output file is to contain characters in a representation other than ascii, the characters are mapped (via outmap) into their proper representation.

Putlin and putch are compatible; that is, interspersed calls to putlin and putch are allowed and work properly. A common implementation is to have putlin call putch until an EOS marker is found.

SEE ALSO

putch, prompt, remark, getch, getlin

DIAGNOSTICS

None

NAME

readf - read from an opened file

SYNOPSIS

```
count = readf (buf, n, fd)

character buf(ARB)    or   integer buf(ARB)
integer n
filedes fd
integer count returned as count/EOF
```

DESCRIPTION

Readf reads "n" bytes (or words) from the file opened on file descriptor "fd" into the array "buf". The bytes (or words) are placed in "buf" one per array element. Readf is the typical way of doing binary reads on files. Whether buf is declared an integer or a character array is dependent upon which is most appropriate for the host operating system.

Readf returns the number of bytes/words actually read. In most cases, this is equal to "n". However, it may be less if an EOF has been encountered or if "fd" specified a device such as a terminal where less than "n" bytes were input.

IMPLEMENTATION

Readf is the typical way of implementing binary I/O. Do whatever is necessary on your system to allow users to get at the file directly.

If reasonable, design readf to work properly in conjunction with getch and getlin.

SEE ALSO

writef, getch, putch

DIAGNOSTICS

None

NAME

remark - print single-line message

SYNOPSIS

```
call remark (message)

integer message - message is a hollerith array
```

DESCRIPTION

Remark writes the message onto the standard error file ERROUT. A NEWLINE is always generated, even though one may not appear in the message. The message array is generally a Fortran hollerith string in the format generated by the Ratfor quoted string capability. On some systems it may be necessary to indicate the end of the message with a period ".". For example,

```
call remark ("this is a warning message.")
```

The escape character "@" may be used to output a period (e.g. @.) and on some systems, the escape sequences "@t" and "@n" and "@b" may be used to output a TAB, NEWLINE, and BACKSPACE respectively.

IMPLEMENTATION

Remark is very similar to error except it returns after printing, instead of stopping. It expects its argument to be a hollerith string which is produced by the ratfor quoted string capability. If your system has no way of determining the end of hollerith strings, you might have to require users to include a termination character such as a ".". (All the quoted strings in the software tools source code do terminate with a dot.)

Remark is similar to the following, except the message string is hollerith rather than character:

```
call putlin (message, ERROUT)
call putch  (NEWLINE, ERROUT)
```

SEE ALSO

error, putlin, putch, prompt

DIAGNOSTICS

None

NAME

remove - remove a file from the file system

SYNOPSIS

```
stat = remove (filename)

character filename(FILENAMESIZE)
integer stat returned as OK/ERR
```

DESCRIPTION

From within a running program, remove (or delete) the file specified by "name" from the file system. "Name" is an ascii character array. The file need not be opened to be removed.

If the file exists and can be removed, OK is returned. If the file does not exist or cannot be removed for some other reason, the function returns ERR.

IMPLEMENTATION

The file to be removed need not be connected to the running program. However, if it is, remove closes the file before removing it.

SEE ALSO

open, close, create

DIAGNOSTICS

If the file does not exist the routine returns ERR.

NAME

seek - move read/write pointer

SYNOPSIS

```
call seek (offset, fd)

integer offset(2)
filedes fd
integer stat returned as OK/ERR
```

DESCRIPTION

Seek moves the read/write pointer of the file specified by "fd" to a (previously identified) spot specified by "offset". "Offset" must have been set by a call to "note", or its first element must be set to one of the constants END_OF_FILE or BEGINNING_OF_FILE (definitions available in the standard symbols file).

Once the file is positioned by a call to seek, reading can be done using the standard I/O calls (getch, getlin, readf).

Seek can also be used for seeking to the end of a file and performing a write (thus extending the file).

Rewriting in place may not be allowed on some systems.

IMPLEMENTATION

Seek depends heavily upon the peculiarities of the operating system. It can generally be used on files opened at READWRITE access.

The offset units are chosen to be whatever is most appropriate for the system involved: characters, words, records, block numbers, line numbers, etc. Two words have been allotted for "offset" although some systems may not need that much.

On some systems READWRITE access may have to be implemented by opening the file twice, once at READ and once at WRITE access.

'Seek' should be made compatible with the standard reading and writing routines.

SEE ALSO

note

DIAGNOSTICS

None

NAME

spawn - execute subtask

SYNOPSIS

```
stat = spawn(command, args, desc, waitflg)

character command(ARB), args(ARB), desc(ARB), waitflg
integer stat returned as OK/ERR
```

DESCRIPTION

Spawn is called to cause execution of a subtask. 'Command' is an ascii character array giving the (file)name of the task to be executed.

'Args' is an ascii character array giving the command line arguments to be passed to the subtask. The arguments are separated by blanks and the entire string is terminated with an EOS marker.

'Desc' is returned as a character array containing an ID for the spawned subtask. This ID may be passed to the 'pstat', 'suspnd', 'resume', and 'kill' process control tools (if implemented).

'Waitflg' is a flag indicating whether or not spawn should return before execution of the task is completed. If WAIT is passed, spawn does not return until execution of the task has completed (the situation for normal commands). If NOWAIT is passed, spawn begins execution of the task and immediately returns (for use with real pipes). If BACKGROUND is passed, spawn executes the task as a background process and immediately returns.

If the task cannot be executed, spawn returns ERR; otherwise it returns OK.

Spawned tasks must be properly taught to read their command line arguments in whatever manner spawn sends them.

IMPLEMENTATION

Spawn is, by far, the most difficult primitive to implement. A few of the major obstacles which must be overcome are:

1. Does the target operating system permit a running process to spawn a subprocess? If it provides a multi-user, interactive environment, it most certainly does, but it may not be common knowledge as to how to do it. For example, the following DEC implementations have been done by the LBL group:

    a) RSX11M - a loadable pseudo-driver is used to stuff MCR commands into MCR's queue, via qio requests.

   b) IAS - the TCS macros provided by the operating system for custom CLI construction are used. The only interface is from assembly language, so that is the language used.

   c) VMS - the sys$creprc system service, which is callable from any supported language, is used. In fact, the entire spawn primitive is written in ratfor.

2. Once one has determined how to spawn the process, it is necessary to determine how to control it. If the operating system does not provide any synchronization methods, then they must be implemented.

3. Finally, one must determine how to communicate the arguments and environment information to the subprocess. This generally entails an exploration of the system provided interprocess-communication mechanisms, and often requires the invention of better ones.

SEE ALSO

DIAGNOSTICS

A message 'Cannot spawn process' is printed if that situation occurs.

NAME

writef - write to an opened file

SYNOPSIS

```
count = writef (buf, n, fd)

character buf(ARB)   or   integer buf(ARB)
integer n
filedes fd
integer count returned as count/ERR
```

DESCRIPTION

Writef writes "n" bytes from the array "buf" to the file opened on file descriptor "fd". Writef is the typical way of doing binary writes to files. Whether buf is declared an integer or a character array is dependent upon which is most appropriate for the host operating system.

Writef returns the number of bytes/words actually written. In most cases, this is equal to "n". If, however, a write error occurs, writef returns ERR.

IMPLEMENTATION

Writef is the typical way of implementing binary I/O. Do whatever is necessary on your system to allow users to get at the file directly.

If reasonable, design writef to work properly in conjunction with putch and putlin.

SEE ALSO

readf, putch, putlin

DIAGNOSTICS

None