



DOS Library for PL/M-86

The ACME Software Deli

Abstract

A history of the ACME Software Deli and why a library for DOS was created

Owner

R. Michael Gillmore

Foreword

The ACME Menu application was first written in 1986, then went dormant a couple of years later with the release and subsequent use of Microsoft Windows 3.1. In the DOS days, everything was done at the command line, with LOTS of repetitive typing. If you were not a typist (I'm not, really) it was tedious indeed.

Move forward to 2019. I retired from software engineering, though the timing was not of my own choosing. I wanted to improve the applications that I had written all those years ago, so I found a DOS look-alike that would run on Windows, and got to work.

My first challenge was to find references that were still available in book form. With the help of a couple of world-wide web sites, I was able to recover two of my favorites: [The Peter Norton Programmer's Guide to the IBM PC & PS/2](#) by Peter Norton and [Advanced MS-DOS](#) by Ray Duncan.

Next on the list was finding a copy of the long since abandoned PL/M-86 compiler. Thanks to the Vetus Software site, another problem solved. Now came the real work. I remembered that there was no operating system API available, so I had to write my own. The earlier version was mostly implemented in x86 assembler, with some of the higher level functions in PL/M-86. The problem was that it meant LOTS of work any time the memory model was changed.

Not wanting to go down that path again, I took advantage of my experience to make the task of writing applications easier. The first step down the new path was to allow the application's entry point (from the programmer's perspective) to be a function called "main" which accepted three parameters and returned an integer, just like those applications written in C, C++ and many other languages.

Since DOS applications have some specific requirements at the start, and since much of the necessary information was available only in the context of the starting instruction, the first lines were written in assembler, but passed to PL/M as soon as reasonable. In that first assembly language module, I made the first improvement to the developer's API, I initialized the static heap to zero. This allowed the programmer to make assumptions about the initial value of variables defined on the heap. Detection of initialization was now possible.

Also absent from those early versions of the (Intel) UDI-like environment was the ability to dynamically allocate memory. Since I wanted the ability to allocate memory dynamically in my applications, I chose that task next. That took all of 8 lines of assembly (with 17 lines of comments so that I would not forget WHY I did it).

Next came the real work of the startup code. For that, I moved to PL/M. It is here that I parsed the command line, and initialized an array of pointers to character strings for the command line parameters, commonly referred to as argv in the C language programming references. That is perhaps the single largest improvement in the API from the developer's perspective. No more calls to DQ\$GET\$ARGUMENT for each of the application's arguments. These arguments could be referenced in any order desired, and could be read many times, not just once.

Now that the initial code was in place, application development could begin. However, those first few simple applications took a LONG time to develop. While PL/M was a great language for embedded microprocessor applications, there was no library of useful utilities to manage I/O, strings, memory or anything else of that nature. This led to the next necessary feature of the environment: executing calls to DOS and the BIOS from PL/M-86, and not resorting to assembler every time.

The early C libraries for DOS had a function called `int86()` which allowed the programmer to set values in registers, and execute the appropriate interrupt. This is perhaps the greatest challenge yet: CPU registers must be initialized before the interrupt, and then captured and returned to the caller without context loss.

With that next critical piece in place, the creation of the I/O portion of the library was substantially easier. I could rely on the system call functionality provided by `int86()` to make the job easier. What followed (quite quickly actually) was the low level I/O in files and the user interface.

Of course, there is no reason to have file I/O if you are not going to use it, so I re-created one of my first serious applications for DOS: `fprint`. Indeed, there was a rudimentary version of that even before I discovered DOS and the IBM-PC. But this version was going to be faster, cleaner, and more likely to be used.

The next big challenge was to make the parsing of the command line easier and more consistent. I did not want to require that the parameters to be specified in a particular order. I also did not want to have command line parsing code repeated in every application.

My experience taught me that a common parser was more valuable than some might expect. As a model, I grabbed the simple API model from the Python library, and implemented it. That let me specify parameters in any order I wished. That was a huge improvement. I could now specify what arguments and modifiers were acceptable, and provided both a short version and a longer version to be specified. If the user wanted to know the acceptable command syntax, that could be provided using the same information provided to the command parser. Two wins with one API.

Lastly, there are two more things that I always wanted to do in my own API, `printf()` and its other related functions `fprintf()` and `sprintf()` which allow for the easy formatting of information, and `strftime()` which allows me to format the date/time string as I wish. The date/time report formatter was relatively easy since there is a fixed number of command arguments.

`printf()/sprintf()/fprintf()` would be a bigger challenge because the PL/M language syntax does not allow direct function calls with a variable number of parameters. I could make an indirect call (through a pointer) with no problem, but how could I find the first argument in the function list CONSISTENTLY? If called through a pointer, the function addressed by the pointer can NOT have any arguments, or the calling sequence is different. I found a method that works, but only when not called from a reentrant function.

(update: `printf()/sprintf()/fprintf()` may reliably now be reliably called from reentrant functions)

Table of Contents

1	Introduction	8
1.1	Copyright and Reserved Rights	8
2	The ACME DOS Library	8
2.1	Additional / Pre-defined Types	8
3	Common References	9
3.1	the ACME Software Deli (and Malt Shoppe)	9
3.2	PL/M-86	9
3.3	Integer Arithmetic	9
4	Handling Strings	9
4.1	memset	10
4.2	memcpy	10
4.3	memcmp	10
4.4	strlen	11
4.5	strchr / strchr	11
4.6	toupper / tolower	11
4.7	strupr / strlwr	12
4.8	strcpy	12
4.9	strncpy	12
4.10	strdup	13
4.11	strcat	13
4.12	strncat	13
4.13	strcmp / strcmp	14
4.14	strtok	14
5	File and User I/O	15
5.1	ci	15
5.2	kbhit	15
5.3	getch	16
5.4	fopen	16
5.5	fclose	16
5.6	fread	17
5.7	flushDiskBuffers	17
5.8	fgets	17

5.9	fseek	18
5.10	ftell	18
5.11	fwrite	18
5.12	printString	19
5.13	printStringAsciiZ	19
5.14	printf(), sprintf() and fprintf()	19
5.14.1	Function prototypes	19
5.14.2	Format Specifications	20
5.15	lfnSupported	21
6	Managing Dates and Times	21
6.1	waitForTicks	22
6.2	currentTime	23
6.3	mktime	23
6.4	isLeapYear	23
6.5	dayOfWeek	23
6.6	localtime	23
6.7	dos2time_t	23
6.8	strftime	23
6.9	asctime	23
7	Video I/O	23
7.1	Helpful enumerals and definitions	23
7.2	gotoxy	24
7.3	curPos	24
7.4	cursorOn / cursorOff	24
7.5	screenSize	24
7.6	getAttributeAtLocation	25
7.7	getCharacterAtLocation	25
7.8	vChangeAttributeAt	25
7.9	vInvertAttributeAt	25
7.10	scrollWindow	26
7.11	clrWindow	26
7.12	writeCharsAtCurrentLocation	26
7.13	writeAt	27
7.14	writeManyAt	27
7.15	clrscr	27

7.16	drawBox	27
8	OS Environment	28
8.1	Common Error Codes.....	28
8.2	getenv	28
8.3	getPSP	29
8.4	getHostname.....	29
8.5	osVersion.....	30
8.6	uname	30
8.7	malloc / free.....	30
8.8	changeExtension	30
8.9	loadExec	31
8.10	findFirst, findNext and findClose	31
8.11	int86	33
8.12	reportLocation / reportBasePtr / reportStackPtr	34
8.13	reportRegs.....	34
9	Utility Functionality.....	34
9.1	bit array manager.....	34
9.2	array manager	36
9.3	Configuration file (.ini) manager	39
9.4	Memory buffer manager.....	41
9.5	logging manager.....	42
9.6	Morse code tools	43
9.7	Interpreting an Application's Command Arguments.....	45
9.8	stack manager	48
10	Applications to Aid the Build Process	48
10.1	plmPreProc.py.....	49
10.2	parseOutput.py	49
10.3	rmSubStr	49
10.4	8dot3path.cmd.....	49
10.5	osArch.cmd	49
10.6	mkDepend.py	50
10.7	back2front	50
10.8	pwd	50
10.9	objsToRsp.cmd	50
10.10	cleanup.cmd	50

10.11	refTouch.....	50
10.12	echoEach.awk	50
10.13	objectToSource.awk.....	50
11	Applications Created Using the Library.....	50
11.1	ACMEdir	50
11.2	ACMEmenu	51
11.3	clrscr / color	53
11.4	dls.....	54
11.5	fprint	55
11.6	repKeys.....	57
11.7	repVols	57
11.8	rdskbufs.....	57
11.9	view file	57
11.10	Encoding & Decoding Morse Code.....	57

1 Introduction

The ACME DOS Library is a creation of the ACME Software Deli (and Malt Shoppe). It is written in PL/M-86 so as to be fast and compact, but it works only in a 32-bit Windows console, or DOS environment. The intent is to help the computer user organize commonly used applications in an easy to use environment.

The contents of this document are intended to provide a user's reference for the application, with some historical context and commentary inserted where appropriate.

As with all applications written by the ACME Software Deli, your comments are appreciated. Forward your notes, questions and comments to:

mike.gillmore@gmail.com

Mike Gillmore
Marion, IA

1.1 Copyright and Reserved Rights

This source code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Permission to use for any purpose, modify, copy, and make enhancements and derivative works of the software is granted if attribution is given to R.M. Gillmore as the author.

While R.M. Gillmore, also known as the ACME Software Deli, does not work for money, there is nonetheless a significant amount of work involved. The ACME Software Deli maintains the rights to all code written, though it may be used and distributed as long as the following conditions are maintained.

1. The copyright statement at the top of each code block is maintained in your distribution.
2. You do not identify yourself as the ACME Software Deli
3. Any changes made to the software are sent to the ACME Software Deli

2 The ACME DOS Library

Just as many other tools, the ACME DOS library is composed of a series of functions that fall into one of several categories. As with most libraries, the lines between the functional categories is often blurry.

The code in this library is to be freely shared including any improvements or additions you make. The only restriction is that attribution must be made to the ACME Software Deli for the original library content. The attribution must also be made in any documentation published.

2.1 Additional / Pre-defined Types

The PL/M-86 language types correspond with the register sizes of the 8086/8088/80186/80188 processors. While these types are practical for the assembly language programmer, the more modern languages have defined types which represent the data.

As an aid to the modern programmer, the following types are "defined"

- bool ean – similar to the C++ "bool" type, with typical values of True/False
- char – similar to the C type, but unsigned
- qword – a two element array of unsigned 32-bit words

For complete details about common defined types, see the file `comnDefs. ext`

3 Common References

3.1 the ACME Software Deli (and Malt Shoppe)

My first job in engineering was as what was then called an “Engineering Aide”. It was my job to assist the engineers in my section with tasks which did not require an engineering degree or experience. One of the first of those jobs was to enter all of the handwritten source code into the computer’s files. I was privileged to learn coding styles and to become familiar with computer operations.

As I was completing one of my first large data entry tasks, I wrote the following in a comment block at the end of the program’s code:

```
C    It looks like this is all, Stevie.  
C    Thank you for shopping the ACME Software Deli (and Mal t Shoppe)  
C    Please come agai n.
```

From that day in the fall of 1980 to this very day, I have used the name of “the ACME Software Deli” as my moniker in the private source code I have authored.

3.2 PL/M-86

The first programming that I did as an engineering programmer was in the language created by Gary Kildall specifically for microprocessors, PL/M. PL/M is an acronym of “Programming Language for Microcomputers.” PL/M-86 is the version targeting the 8086, (and later) the 80186, and their 8-bit data bus counterparts, the 8088 and 80188.

3.3 Integer Arithmetic

Since the PL/M-86 compiler was created for a 16-bit processor, there is no native CPU support for integer arithmetic of any type larger than 16-bits. While addition and subtraction can be performed easily using the 16-bit registers, multiplication and division are more complicated. The DWORD multiplication and DWORD division functions are supported in this library. Their names are the same as those in the original PL/M-86 library.

4 Handling Strings

The PL/M languages had no concept of a string, only arrays of bytes. To assist those who may not be familiar with that limitation, two complements were made.

First, the `char` “type” was introduced as a literal of a byte. While that does not change the compiled code, it can improve the readability of code.

Second, the notion of an ASCII string is now supported. With that, the `strlen()`, `strchr()`, `strcmp()` and similar functions were added to the PL/M-86 library for DOS. It should be noted that these functions have no I/O, so they may be adopted for an embedded system with no file system.

One of the greatest benefits of the C programming language is that a relatively simple yet powerful string handling library is defined. The ACME Library also has a string handling section, but certainly not as extensive as a typical string handling library distributed with a C or C++ compiler.

All of the string related functions described below are defined in the file `stri ng. ext`

4.1 memset

The memset function is used to initialize a block of memory with a value of your choice. Values are limited to 8 bits, but since this function uses built-in processor instructions, this is a small function. The function accepts 3 parameters and is defined as follows:

```
memset:
procedure ( basePtr, byteToWrite, arrayLength ) external ;
    declare    basePtr                pointer,
               byteToWrite            byte,
               arrayLength            word;
end memset;
```

- basePtr - the address of the first byte in your memory block
- byteToWrite - the value to write to each byte in the memory block
- arrayLength - the number of bytes to be written to the block

4.2 memcpy

The memcpy function is used to copy the contents of a memory block to another memory block. The function accepts 3 parameters and is defined as follows:

```
memcpy:
procedure ( destinationPtr, sourcePtr, arrayLength ) external ;
    declare    destinationPtr        pointer,
               sourcePtr              pointer,
               arrayLength            word;
end memcpy;
```

- destinationPtr – the address of the first byte in the destination memory block
- sourcePtr – the address of the first byte in the source memory block
- arrayLength – the number of bytes to copy

4.3 memcmp

The memcmp function compares two memory blocks. When the blocks are identical, a True is returned. If the contents of the blocks differ, False is returned. The function accepts 3 parameters and is defined as follows:

```
memcmp:
procedure ( leftBlockPtr, rightBlockPtr, arrayLength ) boolean external ;
    declare    leftBlockPtr           pointer,
               rightBlockPtr          pointer,
               arrayLength            word;
end memcmp;
```

- leftBlockPtr – the address of the first byte in a memory block
- rightBlockPtr – the address of the first byte in another memory block
- arrayLength – the number of bytes to compare

4.4 strlen

The strlen function counts the number of consecutive non-zero bytes starting with the first byte. The only required parameter is the address of the first byte of the string.

```
strlen:
procedure ( stringPtr ) word external ;
  declare  stringPtr          pointer;
end strlen;
```

- stringPtr – the address of the first byte in the string

4.5 strchr / strrchr

Manipulating a string is a very common practice, especially when interacting with a user. Searching for a particular character in a string is at the top of the list, excepting strlen() perhaps. There are two functions to search for a character in a string. strchr() searches from the beginning of the string, while strrchr() searches backward from the end of the string.

Two parameters are required.

```
strchr:
procedure ( stringPtr, charIn ) pointer external ;
  declare  stringPtr          pointer,
           charIn             char;
end strchr;
```

```
strrchr:
procedure ( stringPtr, charIn ) pointer external ;
  declare  stringPtr          pointer,
           charIn             char;
end strrchr;
```

- stringPtr – the address of the first byte in the string
- charIn – the value for which to search. (any value is allowed, excepting zero which may give unpredictable results)

4.6 toupper / tolower

There are two functions discussed here. Both are made to “force” an alphabetic character to either uppercase or lowercase. Both accept a single byte (char) and return a single byte (char)

If the char presented is an alphabetic character, it is modified, if necessary, to the desired case. If the character presented is not an alphabetic character, the character passed in is returned without modification.

```
toupper:
procedure ( charIn ) char external ;
  declare  charIn            char;
end toupper;
```

```

tolower:
procedure ( charIn ) char external ;
    declare charIn char;
end tolower;

```

- charIn – The character whose case is to be modified

4.7 strupr / strlwr

Just as a single character can be changed to the desired alphabetic case, so can a string be changed to the desired case. Each of these functions accept a single parameter, the address of the first character in the string to be modified. The string presented is modified in place.

Only alphabetic characters are modified.

```

strupr:
procedure ( stringPtr ) external ;
    declare stringPtr pointer;
end strupr;

strlwr:
procedure ( stringPtr ) external ;
    declare stringPtr pointer;
end strlwr;

```

- stringPtr – the address of the first character in the ASCIIZ string to be modified

4.8 strcpy

Similar to memcpy, strcpy copies the contents of one ASCIIZ string to another block, including the NUL terminating character. As with every memory copy, care should be taken to ensure that the destination block is large enough to hold the source string.

```

strcpy:
procedure ( destinationPtr, sourcePtr ) external ;
    declare destinationPtr pointer,
           sourcePtr pointer;
end strcpy;

```

- destinationPtr – the address of the first character position in the destination string
- sourcePtr – the address of the first character in the source string

4.9 strncpy

This function copies from the source string to the destination string. The last character copied is always a NUL character. The difference between this function and strcpy is that a maximum number of characters to be copied is specified also.

```

strncpy:
procedure ( destinationPtr, sourcePtr, maxStringLength ) external ;
    declare    destinationPtr    pointer,
               sourcePtr         pointer,
               maxStringLength   word;
end strncpy;

```

- destinationPtr – the address of the first character in the destination string
- sourcePtr – the address of the first character in an ASCII string
- maxStringLength – the maximum number of characters which may be copied, including the NUL termination character

4.10 strdup

As the name implies, this function creates a duplicate string, and returns it to the caller. The only possible error is insufficient memory.

```

strdup:
procedure ( inputStringPtr ) pointer external ;
    declare    inputStringPtr    pointer;
end strdup;

```

- inputStringPtr is the address of the first character in an ASCII string
- (returned) pointer was allocated from the DOS memory pool using malloc(). It is the responsibility of the caller to return this buffer to the DOS memory pool using free().

4.11 strcat

Concatenating ASCII strings is a common task in many applications. The strcat function appends the contents of the source string to the destination string, including the NUL character.

```

strcat:
procedure ( destinationPtr, sourcePtr ) external ;
    declare    destinationPtr    pointer,
               sourcePtr         pointer;
end strcat;

```

- destinationPtr – the address of the first character in the string to receive addition characters at the end
- sourcePtr – the address of the first character in the string that is to be appended to the destination

4.12 strncat

The strncat function is similar to the strcat function except that the number of characters to be concatenated to the destination string is limited by the maximum number specified. In all other respects, it is the same as the strcat function.

```

strncat:
procedure ( destinationPtr, sourcePtr, maxNumberBytes ) external ;
    declare    destinationPtr    pointer,
               sourcePtr        pointer,
               maxNumberBytes    word;
end strncat;

```

- destinationPtr – the address of the first character in the string to receive addition characters at the end
- sourcePtr - the address of the first character in the string that is to be appended to the destination
- maxNumberBytes - the maximum number of characters which may be copied, including the NUL termination character

4.13 strcmp / stricmp

This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached. The return code is one of the following conditions:

Return Value	Indicates
< 0	the first character that does not match has a lower value in leftPtr than in rightPtr
0	the contents of both strings are equal
> 0	the first character that does not match has a greater value in leftPtr than in rightPtr

```

strcmp:
procedure ( leftPtr, rightPtr ) integer external ;
    declare    leftPtr          pointer,
               rightPtr         pointer;
end strcmp;

```

```

stricmp:
procedure ( leftPtr, rightPtr ) integer external ;
    declare    leftPtr          pointer,
               rightPtr         pointer;
end stricmp;

```

- leftPtr – the address of the first character in the string to be compared
- rightPtr – the address of the first character in the string to be compared

The stricmp() function performs the same comparison as the strcmp() function except that stricmp considers that a lowercase alphabetic character is equivalent to the corresponding uppercase character.

4.14 strtok

One of the most useful of all string handlers allows a single string to be broken into tokens. A token is defined as a sequence of printable characters. Since spaces, tabs, carriage returns, line feed and form feed characters are "printable", there must be a way to allow them to be contained in the token.

However, the user may not wish a token to contain some of those characters. The tokens will be separated by the user's set of delimiters.

A sequence of calls to this function splits the input string into tokens, using the specified delimiters.

```
strtok:
procedure ( stringInPtr, delimiterArray ) pointer external ;
    declare    stringInPtr        pointer,
               delimiterArray     pointer;
end strtok;
```

On a first call, the function expects an ASCIIZ string as the first argument. The first character of the string is used as the starting location to scan for tokens. In subsequent calls, the function expects a NULL pointer and uses the position right after the end of the last token as the new starting location for scanning.

To determine the beginning and the end of a token, the function first scans from the starting location for the first character not contained in `delimiterArray` (which becomes the beginning of the token), and then scans starting from this beginning of the token for the first character contained in delimiters, which becomes the end of the token. The scan also stops if the terminating NUL character is found.

This end of the token is automatically replaced by a null-character, and the beginning of the token is returned by the function.

Once the terminating null character of `stringInPtr` is found in a call to `strtok`, all subsequent calls to this function (with a NULL pointer as the first argument) return a NULL pointer.

The point where the last token was found is kept internally by the function to be used on the next. Unlike most implementations, the memory area containing the original string is NOT modified by calling `strtok`. This is managed by duplicating the input string, then working on the duplicate.

5 File and User I/O

The PL/M-86 language has no library for file input and output. Language purists will tell you that the C language has no defined I/O either, though nearly every C compiler distribution has a standards compliant library.

The DOS library functions for I/O follow those traditions. The API is similar to the C language library with some differences. For instance, a file handle is a word and not a "FILE *". The commonly used pre-defined file handles (`stdin`, `stdout`, `stderr`) are defined in the library.

The functions described below are defined in the file `fileio.ext`

5.1 ci

The `ci()` function is a throwback from Intel's Programming Interface used with the ISIS-II, ISIS-III or iNDX operating systems. This function clears the keyboard buffer, then waits for a character from the keyboard. The key read is NOT reflected to the console when read.

```
ci:
procedure integer external ;
end ci;
```

See the `getch()` function for additional information about the returned value.

5.2 kbhit

The `kbhit()` function examines the keyboard buffer to see if a key has been pressed.


```

kbhit:
procedure boolean external;
end kbhit;

```

The function does not return the key pressed, only an indication whether a key was pressed.

5.3 getch

The `getch()` function reads the next integer from the keyboard buffer. If the keyboard buffer is empty, the function will wait for another keypress.

```

getch:
procedure integer external;
end getch;

```

The key returned is an integer. Values less than zero are "function" keys (F1, F2, ...) or cursor keys. The example application "repKeys" uses this function, then reports the value read.

5.4 fopen

The `fopen()` procedure opens the specified file using the mode specified. The file will be a "regular" file, not a directory or volume label.

The specified mode, is specified by a combination of 'r', 'w' and '+' as defined below.

Mode	Description
r	Read from the file. File is not created if it does not exist
w	Write to the file. If the file does not exist, it will be created
+	After the file is opened, the file position is set at the end

In the mode string, the characters may be combined, for instance 'rw' or 'rw+'. The modes may also be specified in any order.

```

fopen:
procedure ( fileNameString, openTypeString ) word external;
    declare    fileNameString    pointer,
               openTypeString    pointer;
end fopen;

```

- `fileNameString` – the name of the file. The file specified need not be fully qualified. On systems which support the long filenames, this function also supports long file names
- `openTypeString` – a NUL-terminated string. The mode specification characters are expected to be lowercase.

The word returned is a valid file handle, or is 0ffffh if an error was detected. If a valid file handle is not returned, the global variable `errno` contains the error returned from the OS call.

5.5 fclose

The `fclose()` procedure flushes the output buffer for the specified handle, then returns the handle to the DOS available pool. The only parameter specified is the handle returned from the `fopen` procedure.

```

fclose:
procedure ( fileHandle ) external ;
    declare    fileHandle          word;
end fclose;

```

- fileHandle – the word returned from the fopen procedure

5.6 fread

The fread() procedure gathers a block of data from a file represented by a handle. The block into which the data is placed is specified by the caller. The size of that block is specified as a number of bytes. There is no interpretation of the data read.

```

fread:
procedure ( fileHandle, byteArray, byteArrayLength ) word external ;
    declare    fileHandle          word,
               byteArray           pointer,
               byteArrayLength     word;
end fread;

```

- fileHandle – the word returned from the fopen procedure
- byteArray – the address of the memory block into which the data read from the file is placed
- byteArrayLength – the maximum number of bytes read from the file is limited by this value. If there are fewer than this number, it can be inferred that the end of file has been reached.

5.7 flushDiskBuffers

The flushDiskBuffers() procedure commands DOS to refresh all disk buffers, including the directory records. All defined disk buffers are affected, whether in your application or any other background processing.

```

flushDiskBuffers:
procedure external ;
end flushDiskBuffers;

```

5.8 fgets

The fgets() procedure reads characters from the specified file and stores them as a string into charArray until:

- (charArrayLength-1) characters have been read
- a newline has been read
- the end-of-file is reached

whichever happens first.

While a newline character makes fgets() stop reading, it is considered a valid character by the procedure and included in the string copied to charArray.

```

fgets:
procedure ( fileHandle, charArray, charArrayLength ) pointer external ;
    declare    fileHandle      word,
               charArray       pointer,
               charArrayLength word;
end fgets;

```

- fileHandle – this may be either the pre-defined handle (stdin), or a handle returned from fopen()
- charArray – this is the block of memory into which the characters read from the specified file handle are placed.
- charArrayLength – this is the number of bytes allocated for charArray.

5.9 fseek

The fseek() procedure moves the file I/O pointer to the location indicated. The file must be open, and is represented by the fileHandle specified. The location is defined by a combination of the filePosition and seekType.

seekType	description
FROM_END_OF_FILE	The file I/O position is calculated from the end of the file (filePosition is subtracted from the file length)
FROM_BEGINNING_OF_FILE	The file I/O position is moved to the absolute filePosition specified
FROM_CURRENT_POSITION	The file I/O position is calculated by adding the specified filePosition to the current file I/O position.

```

fseek:
procedure ( fileHandle, seekType, filePosition ) external ;
    declare    fileHandle      word,
               seekType        byte,
               filePosition     dword;
end fseek;

```

- fileHandle – this is handle returned from fopen().
- seekType – specified according to the table above
- filePosition – this is the number of bytes to move based on the seek type specified

5.10 ftell

The ftell procedure asks the OS for the file I/O location of the specified file. The file must be open, and is represented by the fileHandle specified. The location is returned as a 32-bit unsigned indicator.

```

ftell:
procedure ( fileHandle ) dword external ;
    declare    fileHandle      word;
end ftell;

```

- fileHandle – returned from the fopen() function

5.11 fwrite

The fwrite() procedure writes an array of bytes (or chars), specified by the byteArray, to a file, specified by the fileHandle. The file is assumed to be open, and the current file I/O position is to be used. The number of bytes to be written is specified by the byteArrayLength, and is limited to an unsigned 16-bit value of 65535.

```

fwrite:
procedure ( fileHandle, byteArray, byteArrayLength ) word external ;
    declare    fileHandle      word,
               byteArray       pointer,
               byteArrayLength word;
end fwrite;

```

5.12 printString

For many of us, the ability to write to the stdout should not be difficult. We don't want formatting, the string already exists in the format we need. We should also note, that sometimes we have the number of characters to print, but they are not represented in a NUL-terminated sequence of characters.

For instance, suppose you have a sequence of characters with a byte whose value is zero. Anything that finds the length of the string by looking for a NUL would not work.

`printString()` accepts two arguments: a pointer to the character array to write, and the number of characters to write. The output always goes to the pre-defined handle stdout.

```

printString:
procedure ( stringPtr, stringLength ) external ;
    declare    stringPtr    pointer,
               stringLength word;
end printString;

```

5.13 printStringAsciiZ

When we have a NUL-terminated sequence of characters to write to the pre-defined handle stdout, we can simply use this function. The only parameter accepted is a NUL-terminated pointer to the character sequence.

```

printStringAsciiZ:
procedure ( stringPtr ) external ;
    declare stringPtr pointer;
end printStringAsciiZ;

```

5.14 printf(), sprintf() and fprintf()

Some of the most useful, but also most complex, of functions allow for the passing of a variable number of arguments, each argument's type defined in another parameter. The `printf()` family of functions do exactly that.

Some of the challenges of writing in PL/M-86 is the language syntax does not support function prototypes with a variable number of arguments, and there are no built-in functions to find the parameters when called through a pointer. In the C-based languages, there is support using `var_args()`.

The `printf()`, `fprintf()` and `sprintf()` procedures are variadic. Because functions with a variable number of parameters are not supported in PL/M, these functions are called through pointers of the same name as the traditional functions.

There has been a conscious effort to make the interface for these functions as close to the C counterparts as possible.

5.14.1 Function prototypes

With these functions, the common parameter, whether first or second, is the `formatString`. This string is always NUL-terminated. If `formatString` includes *format specifiers* (subsequences beginning with % or \), the additional

arguments following `formatString` are formatted and inserted in the resulting string replacing their respective specifiers. In PL/M-86, the value of the `formatString` parameter is a pointer to the first character in the string.

The order in which the additional function parameters are passed **MUST** correspond to the format specifiers in size and type.

The `printf()` function is generally considered the most easy to understand, and is called thusly.

```
call printf( formatString, ... );
```

The `fprintf()` function contains the handle of the file to which the translated string is written. The `fileHandle` is returned from `fopen()`, or is one of the pre-defined output handles (`stdout`, `stderr`, `stdaux` and `stdprn`). It is called thusly:

```
call fprintf( fileHandle, formatString, ... );
```

The `sprintf()` function accepts the address in the caller's data area where the translated string is copied. It is the caller's responsibility to ensure the data area is large enough to accept the translated string. It is called thusly:

```
call sprintf( returnString, formatString, ... );
```

5.14.2 Format Specifications

In the tables below, there are three types of string formatters described. Every effort has been made to allow the specifiers to match those in the common C or C++ libraries, but they are not always the same.

Table 1 - Predefined ASCII control characters

Specifier	Action taken
\a	replace with the BELL (ASCII) character
\b	replace with the BACKSPACE (ASCII) character
\f	replace with the FORMFEED (ASCII) character
\n	replace with the new-line sequence (CR/LF) (assume the DOS new-line sequence)
\r	replace with the CARRIAGE RETURN (ASCII) character
\t	replace with the TAB (ASCII) character

Table 2 - Insert raw byte values

Specifier	Action taken
\0xxx	Interpret the value as octal (the value is interpreted as a maximum of 3 octal digits following the first zero)
\hxx	Interpret the two digits following the \h as hexadecimal digits. (the case of the hex digit is not significant => A is a, c is C)

Table 3 - Format string modifiers

Specifier	Action taken
%x	using as many hex digits as necessary, insert the 16-bit value
%lx	using as many hex digits as necessary, insert the 32-bit value
%bx	using as many hex digits as necessary, insert the 8-bit value
%d	using as many decimal digits as necessary, insert the SIGNED 16-bit value
%ld	using as many decimal digits as necessary, insert the SIGNED 32-bit value
%bd	using as many decimal digits as necessary, insert the SIGNED 8-bit value
%ud	using as many decimal digits as necessary, insert the UNSIGNED 16-bit value
%uld	using as many decimal digits as necessary, insert the UNSIGNED 32-bit value
%ubd	using as many decimal digits as necessary, insert the UNSIGNED 8-bit value
%c	insert the character
%s	insert the NUL terminated string
%p	insert the pointer value
%t	insert the "T"/"F" boolean indicator
%T	insert the "True"/"False" boolean indicator

5.15 IfnSupported

Depending on the DOS version, "long" file names are supported. This means that the calls to the DOS API may be different. This function was originally written specifically for the file I/O function `fopen()` and the directory I/O functions `findfirst()`, `findnext()` and `findclose()`.

When called, TRUE is returned if the long file names are supported. For example:

"C: \Users\Owner\Documents\utils" rather than the traditional "C: \USERS\OWNER\DOCUME~1\UTILS". With the long names, mixed case is allowed/supported, and each name may exceed the original 8.3 file name format.

```

IfnSupported:
    procedure boolean external ;
end IfnSupported;

```

6 Managing Dates and Times

Dates and Times are generally stored in one of two ways. The first is in a 32-bit word typed as `time_t`. It contains all of the information to extract a complete year, month, date, along with the complete time of day (hours, minutes, seconds).

The `time_t` type is actually a packed date/time stamp.

`timeStamp` is defined as a word, and is bit-mapped as

```

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  hours      |  minutes      |  seconds      |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0

```

`dateStamp` is defined as a word, and is bit-mapped as

```

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  years since 1980  |  month   |  day of month  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0

```

The second method of storing the date and time is the `tmStruct` structure, whose definition is similar to that found in the UNIX variants.

Member	Type	Meaning	Range
<code>tm_sec</code>	<code>int</code>	seconds after the minute	0-61*
<code>tm_min</code>	<code>int</code>	minutes after the hour	0-59
<code>tm_hour</code>	<code>int</code>	hours since midnight	0-23
<code>tm_mday</code>	<code>int</code>	day of the month	1-31
<code>tm_mon</code>	<code>int</code>	months since January	0-11
<code>tm_year</code>	<code>int</code>	years since 1900	
<code>tm_wday</code>	<code>int</code>	days since Sunday	0-6
<code>tm_yday</code>	<code>int</code>	days since January 1	0-365
<code>tm_isdst</code>	<code>int</code>	Daylight Saving Time flag	

The Daylight Saving Time flag (`tm_isdst`) is greater than zero if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and less than zero if the information is not available. (NOT supported in our implementation)

- `tm_sec` is generally 0-59. The extra range is to accommodate for leap seconds in certain systems. (not likely supported in DOS)

```

declare  tmStruct  literally 'structure
(
    tm_year      word,
    tm_yday      word,
    tm_isdst     integer,
    tm_sec       byte,
    tm_min       byte,
    tm_hour      byte,
    tm_mday      byte,
    tm_mon       byte,
    tm_wday      byte
);

```

Both definitions can be found in the `date. ext` file.

6.1 `waitForTicks`

To manage the speed at which an application runs, it is often helpful to wait for a clock tick. The `waitForTicks()` function allows the caller to wait for the specified number of clock ticks. In DOS, the clock ticks at the rate of 18.2 Hz.

One should note that the first clock tick may appear at a rate greater than 18.2 Hz, but those which follow will be at the 18.2 Hz rate.

```

waitForTicks:
procedure ( numberTicks ) external ;
    declare  numberTicks      word;

```

```
end waitForTicks;
```

6.2 currentTime

The `currentTime()` function makes a call to DOS for the time and date, translates what DOS reports, and fills the `tmStruct` provided by the caller.

```
currentTime:
procedure ( tmStructPtr ) external;
    declare    tmStructPtr    pointer; /* to a tmStruct */
end currentTime;
```

6.3 mktime

The `mktime()` function converts from a `tmStruct` type to a `time_t` type.

```
mktime:
procedure ( tmStructPtr ) time_t external;
    declare    tmStructPtr    pointer; /* to a tmStruct */
end mktime;
```

6.4 isLeapYear

The `isLeapYear()` function examines the specified year to determine if it is a leap year. True indicates a leap year.

```
isLeapYear:
procedure ( yearIn ) boolean external;
    declare    yearIn        word;
end isLeapYear;
```

6.5 dayOfWeek

This function returns the day of the week in numeric form.

6.6 localtime

6.7 dos2time_t

6.8 strftime

6.9 asctime

7 Video I/O

This section describes the functions available to manage the cursor shape and location, change the screen colors, scroll a rectangular window on the display, and draw a rectangular box.

These functions address the text mode, and not the video modes.

The function prototypes are defined in the file `videoI/O.ext`. It is defined so that it may be included as desired.

7.1 Helpful enumerals and definitions

At the top of the `videoI/O.ext` file are several helpful definitions. At the very top of the file are definitions for the line drawing characters, including the corners.

Following the definitions for the line drawing characters are definitions for scrolling direction, a definition for a screen coordinate (in (x, y)) and the screen colors.

The screen colors are defined thusly:

The character attribute is defined as a byte, grouped in two blocks of four bits. The upper four bits define the background color and whether the character is to blink. The lower four bits define the foreground color (the color of the character) including its intensity. The following table provides some examples.

"blink" attribute	Background color	Character color intensity	Character color	Description
0	010	1	111	Green background with white characters
1	001	0	011	Blinking cyan character on blue background

7.2 gotoxy

The gotoxy() procedure accepts 2 bytes, defining the xCoordinate (which column) and the yCoordinate (which row) of the display. The console cursor is positioned at the desired coordinate location so the next console I/O operation takes place at that location.

```
gotoxy:
procedure ( xCoordinate, yCoordinate ) external ;
    declare    xCoordinate    byte,
               yCoordinate    byte;
end gotoxy;
```

7.3 curPos

The curPos() function returns an array of two bytes. The first byte (the least significant byte of the word) is the xCoordinate. The second byte (the most significant byte of the word) is the yCoordinate.

```
curPos:
procedure word external ;
end curPos;
```

7.4 cursorOn / cursorOff

The cursorOn() and cursorOff() functions show/hide the cursor, regardless of the cursor's shape.

```
cursorOn:
procedure external ;
end cursorOn;

cursorOff:
procedure external ;
end cursorOff;
```

7.5 screenSize

The screenSize() function reports the screen size in two words. The first word is the number of screen columns (a word). The second word is the number of screen rows.

The two parameters are pointers to the words as described in the previous paragraph.

```
screenSize:
procedure ( columnsPtr, rowsPtr ) external ;
    declare    columnsPtr      pointer /* to a word */,
               rowsPtr        pointer /* to a word */;
end screenSize;
```

7.6 `getAttributeAtLocation`

The `getAttributeAtLocation()` function reports the attribute at the specified location. Display page zero is assumed. The attribute returned may be decoded using the definitions in the `videoIO_ext` file.

```
getAttributeAtLocation:
procedure ( xCoordinate, yCoordinate ) byte external ;
    declare    xCoordinate      byte,
               yCoordinate      byte;
end getAttributeAtLocation;
```

7.7 `getCharacterAtLocation`

The `getCharacterAtLocation()` function reports the character at the specified location. The value of the character is decoded as extended ASCII (line drawing characters, and so forth). Display page zero is assumed.

```
getCharacterAtLocation:
procedure ( xCoordinate, yCoordinate ) char external ;
    declare    xCoordinate      byte,
               yCoordinate      byte;
end getCharacterAtLocation;
```

7.8 `vChangeAttributeAt`

The `vChangeAttributeAt()` function moves to the designated screen position as specified by `xCoordinate` and `yCoordinate`. At the specified location, the character attribute is changed as specified by `newAttribute`.

```
vChangeAttributeAt:
procedure ( newAttribute, xCoordinate, yCoordinate ) external ;
    declare    newAttribute      byte,
               xCoordinate      byte,
               yCoordinate      byte;
end vChangeAttributeAt;
```

7.9 `vInvertAttributeAt`

The `vInvertAttributeAt()` function moves to the designated screen position (as specified in the coordinate whose address was passed), reads the character attribute at that location, inverts the colors (preserving the blinking and intensity indicators), then sets the attribute to the calculated value.

```

vInvertAttributeAt:
procedure ( positionPtr ) external ;
    declare    posi ti onPtr          pointer; /*    to a coordi nateType */
end vInvertAttributeAt;

```

7.10 scrollWindow

The `scrollWindow()` function moves the specified window contents in the direction indicated, filling the resulting blank line(s) using the attribute at the window origin (before scrolling).

Using this function, a window of text may be scrolled either up or down. The window to be scrolled is contained by the coordinates (upperLeftRow, upperLeftCol umn) and (l owerRi ghtRow, l owerRi ghtCol umn). The window may be scrolled either up (where the blank line appears at the bottom) or down (where the blank line appears at the top).

Indeed, the `clrscr()` function is a special case of the `scrollWindow()` function.

```

scrollWindow:
procedure ( upperLeftRow, upperLeftCol umn,
            l owerRi ghtRow, l owerRi ghtCol umn, di recti on ) external ;
    declare    upperLeftRow          byte,
               upperLeftCol umn      byte,
               l owerRi ghtRow        byte,
               l owerRi ghtCol umn    byte,
               di recti on            scrollDi recti on_t;
end scrollWindow;

```

7.11 clrWindow

The `clrWindow()` function clears the specified window using the specified video attribute. The window to be cleared is contained by the coordinates (upperLeftX, upperLeftY) and (l owerRi ghtX, l owerRi ghtY).

The attribute is defined in accordance with the color definitions at the top of this section.

```

clrWindow:
procedure ( upperLeftX, upperLeftY, l owerRi ghtX, l owerRi ghtY, attri bute ) external ;
    declare    upperLeftX            byte,
               upperLeftY            byte,
               l owerRi ghtX          byte,
               l owerRi ghtY          byte,
               attri bute             byte;
end clrWindow;

```

7.12 writeCharsAtCurrentLocation

The `writeCharsAtCurrentLocati on()` function writes the character indicated for the designated number of times beginning at the current location.

```
wri teCharsAtCurrentLocati on:
procedure ( characterToWri te, numberToWri te ) external ;
    decl are   characterToWri te   char,
               numberToWri te     byte;
end wri teCharsAtCurrentLocati on;
```

7.13 writeAt

The wri teAt() function moves to the designated screen position (x, y), then writes the character indicated. The character is written exactly once.

```
wri teAt:
procedure ( charToWri te, xCoordi nate, yCoordi nate ) external ;
    decl are   charToWri te        char,
               xCoordi nate        byte,
               yCoordi nate        byte;
end wri teAt;
```

7.14 writeManyAt

The wri teManyAt() function moves to the designated screen position (x, y), then writes the character for the designated number of times.

```
wri teManyAt:
procedure ( charToWri te, xCoordi nate, yCoordi nate, numberToWri te ) external ;
    decl are   charToWri te        char,
               xCoordi nate        byte,
               yCoordi nate        byte,
               numberToWri te      byte;
end wri teManyAt;
```

7.15 clrscr

The cl rscr() function clears the entire screen ((0, 0) to (numberCol umns-1, numberRows-1)) using the video attribute at the origin (0,0) screen location.

```
cl rscr:
procedure external ;
end cl rscr;
```

7.16 drawBox

The drawBox() function creates a box of extended ASCII characters. The line type is either SI NGLE_LI NE or DOUBLE_LI NE, and may differ for the horizontal and vertical lines.

The upper left and lower right corners are specified using a pointer to the coordinate type.

```

drawBox:
procedure ( vertical Li neType, hori zontal Li neType, upperLeft, lowerRi ght ) external ;
    declare vertical Li neType      l i neType,
           hori zontal Li neType    l i neType,
           upperLeft                poi nter, /*    to coordi nateType */
           lowerRi ght              poi nter;
end drawBox;

```

8 OS Environment

Each operating system (OS) has an environment which provides a consistent behavior among the applications run. Part of the environment provides information such as the path to search for an application to be executed. Another part of the environment is a series of services a developer may use while developing an application.

In the case of the Acme Software Microsoft Disk Operating System (MS-DOS, or simply DOS) Library, the OS Application Programming Interface (API) is a collection of services that don't really fit anywhere else.

8.1 Common Error Codes

Like every other OS environment, there are commonly defined error codes. These codes are defined in two categories:

- error codes returned from the services provided by the OS
- error codes based on conditions detected by an application

In the `errCode. ext` file, there are two lists of named constants; one list for each category noted above. The `decodeError()` function allows an application to get an English description of an error code returned.

```

decodeError:
procedure ( errorType, errorNumber, stri ngPtr ) external ;
    declare errorType      errorType_t,
           errorNumber     i nteger,
           stri ngPtr       poi nter;
end decodeError;

```

- `errorType` - either `OS_ERROR` or `APP_ERROR`
- `errorNumber` - the value returned from a library function
- `stri ngPtr` - the address of the first char in a caller-defined string. The string returned will be NUL-terminated. It is the caller's responsibility to ensure the string is large enough to hold the description information.

8.2 getenv

Like the C and C++ libraries, this `getenv()` function returns a pointer to the environment variables. The number of variables may vary, depending entirely upon what has been created by DOS, and what has been added by the user.

This function allows for two options. If the `vari abl eNameStri ng` is `NULL` (this is actually a pointer), the address of the environment variable block is returned. However, you may get the address of a particular variable by putting its name in the string.

The address of either the variable name block, or of the specified variable is returned to the caller. `NULL` is returned if the caller has provided a name, but a variable of that name is not present in the environment.

```

getenv:
procedure ( variableNameString ) pointer external ;
    declare   variableNameString      pointer;
end getenv;

```

8.3 getPSP

The Program Segment Prefix (PSP) is created by the operating system when an application is loaded into memory and executed. The PSP contains lots of useful information, defined by the `pspStructure`. A pointer to this structure is returned by a call to the `getPSP()` function.

```

declare pspStructure      literally
    'structure (
        int20h                (2) byte,
        allocationEndBlock    word,
        reservedByte1         byte,
        farCallToDosDispatcher (5) byte,
        prevTermHandler        pointer,
        prevControlCHandler    pointer,
        prevCriticalSection    pointer,
        parentProcessPsp       selector,
        handleTable            (20) byte,
        environmentBlock       selector,
        reservedDosWorkArea    (34) byte,
        int21hRetf              (3) byte,
        reservedBytes2         (2) byte,
        FcbNum1Extension       (7) byte,
        FcbNum1                 (16) byte,
        FcbNum2                 (16) byte,
        reservedBytesUnknown   (4) byte,
        commandTailLength      byte,
        commandTail            (127) byte
    )';

```

While much of this information may be useful, some is lost as the application runs. Most notably, the `commandTailLength` and the `commandTail` are lost as the application runs.

For additional information about the program segment prefix, I would recommend Ralf Brown's document, or the Peter Norton and Ray Duncan books referenced in the foreword.

8.4 getHostname

In the more modern versions of DOS (those with networking capability), the name of the machine is necessary. By calling the `getHostname()` function, that name may be discovered.

```

getHostname:
procedure ( nameStringPtr ) external ;
    declare nameStringPtr      pointer;
end getHostname;

```

The one procedure argument is the address of a memory block large enough to contain the name of the machine on which the application is running.

8.5 osVersion

The `osVersion()` function returns a word (16-bits) which represents the major version and minor version of the operating system on which the application is running.

The major version is in the lower byte, while the minor version is returned in the upper byte. For example, if the OS version is 3.11, the version value returned would be 0b03h.

```
osVersion:
  procedure word external ;
end osVersion;
```

8.6 uname

The `uname()` function gathers the name of the operating system and its version, presenting them in ASCII in the memory block whose address is presented in the `stringPtr` parameter.

```
uname:
  procedure ( stringPtr, maxStringLength ) external ;
    declare stringPtr      pointer,
           maxStringLength  word;
end uname;
```

8.7 malloc / free

The function `malloc()` is used to request an allocation of a certain amount of memory during the execution of a program. If the request is granted, the operating system will reserve the requested amount of memory, returning the address of the memory block. If a block of the size requested is not available, NULL is returned.

That block then “belongs” to the application, which may use it as long as necessary.

When the allocated memory block is no longer needed, you must return it to the operating system by calling the function `free()`. Failure to free the allocated block will result in a memory leak.

```
malloc:
  procedure ( blockSize ) pointer external ;
    declare blockSize      word;
end malloc;

free:
  procedure ( blockPtr ) boolean external ;
    declare blockPtr      pointer;
end free;
```

8.8 changeExtension

One of the more common functions relating to files is to identify a group which differ only by the extension. The extension is identified as those characters which follow the last period in the name of a file.

```

changeExtension:
procedure ( fileNameString, newExtension ) external ;
    declare   fileNameString      pointer,
              newExtension        pointer;
end changeExtension;

```

There are two parameters. The first, `fileNameString`, is a pointer to the beginning of a file's name, which may be fully-qualified. The second parameter, `newExtension`, is a pointer to the beginning of the desired file extension.

It is the responsibility of the calling function to ensure the `fileNameString` is large enough to the new filename extension. The expectation is that these strings are NUL-terminated. If these strings do not have the NUL terminating character, this function will likely fail catastrophically.

8.9 loadExec

In the modern operating system libraries, there is an API which allows an application to load and execute another application. One of the greatest challenges in my early DOS programming with PL/M-86 was to create a function which allows such things. This is a simple, but quite effective, function which accepts a command line, like one would type at the DOS prompt.

Just as with DOS, there is a limitation to the length of the command to be executed. In DOS, the limit is 127 bytes. This implementation reduces that limit to 124. It is the calling function responsibility to ensure this restriction is not violated.

The `ERRORLEVEL` returned by the application is given to the caller as a return value.

```

loadExec:
procedure ( commandLinePtr ) word external ;
    declare   commandLinePtr      pointer;
end loadExec;

```

As with many other functions, the `commandLinePtr` string is NUL-terminated.

8.10 findFirst, findNext and findClose

With the implementation of the POSIX functions of `opendir()`, `readdir()`, `rewinddir()` and `closedir()`, these functions are not likely to be used often, but they are published nonetheless since they have other useful purposes beyond simple file searches in the DOS file system.

If the DOS platform on which your application is running supports "long" filenames (beyond the 8.3 specification), these functions will search using the long filename functions.

The function prototypes are:

```

findFirst:
procedure ( fileNamePtr, findDataRecordPtr ) word external ;
    declare   fileNamePtr          pointer,
              findDataRecordPtr    pointer;
end findFirst;

findNext:
procedure ( findDataRecordPtr ) word external ;
    declare   findDataRecordPtr    pointer;
end findNext;

```



```

fi ndCl ose:
procedure ( fi ndDataRecordPtr ) external ;
    declare   fi ndDataRecordPtr  pointer;
end fi ndCl ose;

```

The `fi ndFi rst()` function accepts two parameters. The first is the filename search specification defined in a NUL-terminated ASCII string. It may contain all characters as defined by your version of DOS. This name also supports the * and ? wildcard characters. The asterisk allows the name to specify one or more consecutive filename characters for which to search. The question mark will match exactly one character in the filename. See your DOS manual for more information to describe the wildcard character set.

The second is a pointer to an allocated block of memory of type `fi ndDataRecordType`. The `fi ndDataRecordType` is defined thusly:

```

structure (
    fileAttr      dword,
    creati onTi me qword,
    accessTi me   qword,
    modi fi edTi me qword,
    fi leSi ze     qword,
    reserved      ( 8 )    byte,
    ful lName      ( MAX_FI LENAME )    char,
    dosName        ( 14 )    char,
    fi ndHandl e   word,
    dtaData        dta_t
)

```

The nested `dta_t` structure is defined as, though it exists to pass from `findFirst()` to `findNext()`, and generally should not be used in an application.

```

structure (
    fi ndNextData      (21) byte,
    fi leFoundAttri bute    byte,
    ti meStamp          word,
    dateStamp          word,
    fi leSi ze          dword,
    fi leName           (13) char
)

```

It should also be noted that though the `dtaData` element is defined, it does not contain valid data when the long filenames are supported.

The `fi ndNext()` function is designed to be called repetitively following the initial call to `fi ndFi rst()`. The same `findDataRecord` populated by `fi ndFi rst()` is passed to `fi ndNext()` on every call until the desired file records have been collected.

The `fi ndCl ose()` function is called to clean up the DOS memory blocks used for `fi ndFi rst()` and `fi ndNext()`. The same `fi ndDataRecord` used with the `fi ndNext()` is passed to `fi ndCl ose()`.

8.11 int86

The early C libraries for DOS had a function called `int86()` which allowed the programmer to set values in registers, and generate the appropriate interrupt. In this library, the `int86()` function has been recreated to allow lower level functions to be implemented in PL/M-86 rather than in assembly.

To use `int86()`, there are three code macros which were created. The first macro, `registerType`, is a structure with each of the 16 bit CPU registers, defined in a predictable order. The second macro, `byteRegisterType`, is a structure defining the 8 bit definitions of the general purpose registers. The third macro, `cpuRegisters`, is what most who call `int86()` will use. This macro uses both `registerType` and `byteRegisterType` to allow access to the 8 bit registers by name while modifying the 16 bit general purpose registers like one might do in assembly language.

```
declare registerType    l i t e r a l l y
    'structure (
        AX            word,
        BX            word,
        CX            word,
        DX            word,
        SI            word,
        DI            word,
        DS            selector,
        ES            selector,
        FLAGS_reg     word
    )';

declare byteRegisterType l i t e r a l l y
    'structure (
        ( AL, AH )    byte,
        ( BL, BH )    byte,
        ( CL, CH )    byte,
        ( DL, DH )    byte
    )';

declare cpuRegisters    l i t e r a l l y
    '    wordRegs      registerType,
        byteRegs      byteRegisterType at ( @wordRegs )';

int86:
procedure ( intNumber, cpuRegsPtr ) external;
    declare    intNumber        byte,
               cpuRegsPtr       pointer,
               registers         based cpuRegsPtr registerType;
end int86;
```

The `intNumber` is a value from 0 to 0ffh (255) as defined in the DOS programming references. The `cpuRegsPtr` is the address of the `registerType` structure.

The value of each register immediately following the interrupt instruction are stored in the `registerType` structure for an application's use.

8.12 reportLocation / reportBasePtr / reportStackPtr

The reportLocation() function walks the stack, returning the address to which the function will return. While the location is not precise, it is generally close enough for logging purposes. The same is true for the reportBasePtr() and reportStackPtr() functions.

```
reportLocation:
procedure pointer external ;
end reportLocation;

reportBasePtr:
procedure pointer external ;
end reportBasePtr;

reportStackPtr:
procedure pointer external ;
end reportStackPtr;
```

8.13 reportRegs

The reportRegs() function is generally used for logging information while the application is running. There are two parameters passed, containing the CPU register values and a location the code.

```
reportRegs:
procedure ( regPtr, locationPtr ) external ;
    declare    regPtr          pointer,
               locationPtr     pointer;
end reportRegs;
```

The regPtr parameter is the address of the structure (a registerType) containing the values of the general purpose registers. The locationPtr is a value returned from the reportLocation() function.

9 Utility Functionality

Now that we have some of the lower level functionality described, we move on to some of the utilitarian functionality. In the object-oriented world, these sections would be in the lower level class structure.

9.1 bit array manager

The bit array manager has its history in the world of true embedded software, a class of software which has no operating system, and which manages the hardware directly. In the early days of embedded software, a boolean was represented as a bit. The boolean variables were stored in an array of bytes.

As with many objects in more modern software, the bit array is a blob, which can decode itself, but whose characteristics are revealed only by the interface. That model was followed as closely as possible. Some may refer to this as "object directed" rather than "object oriented". Nonetheless, the interface will be discussed in groups: Constructors, Accessors, and Manipulators.

```
/*
*****
*   Constructors
*****
*/
```

```

createBitArray:
procedure ( numberBits ) bitArray_t external ;
    declare    numberBits        integer;
end createBitArray;

deleteBitArray:
procedure ( thisPtr ) external ;
    declare    thisPtr          bitArray_t;
end deleteBitArray;

```

One will notice that construction of a bit array is based on the number of bits.

Each instance of the bit array is implemented using a block of memory allocated using `malloc()`. So, for every instance created by `createBitArray()`, a call to `deleteBitArray()` must be made to free the block allocated. Failure to call `deleteBitArray()` following a successful return from `createBitArray()` will cause a memory leak.

```

/*
*****
*   Accessors
*****
*/

bitArrayLength:
procedure ( thisPtr ) integer external ;
    declare    thisPtr          bitArray_t;
end bitArrayLength;

bitIsSet:
procedure ( thisPtr, bitIndex ) boolean external ;
    declare    thisPtr          bitArray_t,
               bitIndex         integer;
end bitIsSet;

bitIsClear:
procedure ( thisPtr, bitIndex ) boolean external ;
    declare    thisPtr          bitArray_t,
               bitIndex         integer;
end bitIsClear;

```

The accessors will report on the status of each instance of a bit (set/clear) and the number of bits in the array. A call to any of these functions does not modify the array in any way.

One may notice that since a bit is a boolean, it can only be true (set) or false (clear). This means that a call to `bitIsSet()` will always return the opposite of `bitIsClear()` for any bit in the array. Providing both allows code to be more easily read, and provides no additional functionality.

The `bitArrayLength()` function returns the number of bits in the array. The current implementation is limited to 32767 bits.

```

/*
*****
*   Manipulators
*****
*/

newBitArrayLength:
procedure ( thisPtr, numberBits ) external ;
    declare    thisPtr      bitArray_t,
               numberBits   integer;
end newBitArrayLength;

```

The `newBitArrayLength()` function allows the array to either shrink or grow. A call to this function on a system with an insufficient amount of memory to grow will fail, but not report. To confirm that the array has indeed grown, a call to the `bitArrayLength()` accessor should be made to confirm the growth.

It should also be noted that the growth of the array does not lose information, and the new bits added to the array are cleared.

```

setBit:
procedure ( thisPtr, bitIndex ) external ;
    declare    thisPtr      bitArray_t,
               bitIndex     integer;
end setBit;

clearBit:
procedure ( thisPtr, bitIndex ) external ;
    declare    thisPtr      bitArray_t,
               bitIndex     integer;
end clearBit;

```

The `setBit()` and `clearBit()` functions address a single bit in the array, either setting or clearing as expected.

9.2 array manager

One of the more useful objects in any application is that of an array. For instance, the first two parameters passed to function `main()` are values allowing one to access an array of strings representing the tokens located on the command line. The array manager is a more general purpose manager of objects of any fundamental type.

Unlike C++, python, and other more object-oriented languages, which copy data for any object type, this array manager does not copy the data associated with structures. In the case of those more complex objects, the address of the object is stored for later retrieval.

As with the bit array manager, the functions will be discussed by the common purposes of construction, accessing or manipulating.

```

/*
*****
*   Constructors
*****
*/

```

```

createArray:
procedure ( numberElements, arrayElementType ) pointer external ;
    declare    numberElements      integer,
               arrayElementType    numberReportType;
end createArray;

deleteArray:
procedure ( basePtr ) external ;
    declare basePtr                pointer;
end deleteArray;

```

The createArray() function accepts two parameters. The first, numberElements, is a starting number of elements in the array. The second parameter, arrayElementType, tells the manager the size of each element to be managed.

There are ten types defined for the arrayElementType, the same as those in the numStrs. ext header file. This definition allows the array manager to handle different size data using the same code. They are defined thusly:

byteType, byteDecimalType, wordType, wordDecimalType, dwordType, dwordDecimalType, pointerType, stringType, booleanType and integerType.

One should note that the type specified only provides a data size. There is no interpretation of the data provided.

The deleteArray() function accepts only one parameter, the pointer returned by the createArray() function. The failure to execute deleteArray() for each execution of createArray() causes a memory leak.

```

/*
*****
*   Accessors
*****
*/

reportArray:
procedure ( basePtr ) external ;
    declare basePtr                pointer;
end reportArray;

getElement:
procedure ( basePtr, index, elementPtr, statusPtr ) external ;
    declare basePtr                pointer,
               index                integer,
               elementPtr           pointer,
               statusPtr            pointer;
end getElement;

nextElement:
procedure ( basePtr, elementPtr, statusPtr ) external ;
    declare basePtr                pointer,
               elementPtr           pointer,
               statusPtr            pointer;
end nextElement;

```

```

arrayLength:
procedure ( basePtr ) integer external ;
    declare basePtr                pointer;
end arrayLength;

```

There are four accessors, though only three are in general use. The first parameter of each function is the address returned from the `createArray()` function. One might think of the provided value as the "this pointer".

The `reportArray()` function is intended for debugging, and has no executable code. That code may be restored at any time by modifying the condition of the `$if / $endif` block within the function.

The `getElement()` function allows the array to be accessed random order. The index provided is verified within the limits of the number of defined in the array.

The `nextElement()` function allows the elements to be examined in the order in which they were added to the array. One might think of this as the "succession accessor". The `rewindArray()` manipulator function is related.

The last two function parameters of the `getElement()` and `nextElement()` functions are the same, and serve the same function. The `elementPtr` parameter is the address of a memory block into which the value in the array will be copied. The `statusPtr` is the address of a word reflecting the status of the function called. A value other than zero indicates a failure in the function execution.

The `arrayLength()` function reports the number of elements currently defined.

```

/*
*****
*   Manipulators
*****
*/

addElement:
procedure ( basePtr, elementValuePtr, statusPtr ) external ;
    declare   basePtr                pointer,
              elementValuePtr        pointer,
              statusPtr               pointer;
end addElement;

setElement:
procedure ( basePtr, index, elementValuePtr, statusPtr ) external ;
    declare   basePtr                pointer,
              index                   integer,
              elementValuePtr         pointer,
              statusPtr               pointer;
end setElement;

rewindArray:
procedure ( basePtr ) external ;
    declare   basePtr                pointer;
end rewindArray;

```

The three manipulator functions accept as their first parameter the address returned from `createArray()`.

The last two parameters of the `addElement()` and `setElement()` functions are the address of the value to be added to the array, and address of a word which reflects any error condition detected. Further discussion of the status word is provided with the accessor functions.

The `addElement()` function accepts the value indicated, appending it to the array.

The `setElement()` function places the input value at the specified location in the array. An error is returned if the specified index is outside of the currently defined array space.

The `rewindArray()` function is often used in conjunction with the `nextElement()` accessor function. One would call this function to adjust the index used by the `nextElement()` accessor function to the beginning of the array. The array order is not changed.

9.3 Configuration file (.ini) manager

Several years ago, I was working on a navigation application for Windows embedded and Android. The characteristics of the application could be changed “over-the-air”. We used a configuration file to make the changes.

When it came time to recreate this library, I decided to add this feature. It was written to allow multiple instances. While not completely object-oriented, every consideration was given to hide the implementation outside the manager.

A typical configuration files is formatted like this:

```
[section1]
name1 = value1
name2 = value2

[ section2 ]
  Aname=Avalue
  AnotherName=YetAnotherValue
```

There may be more than one key/value pair in each section, and more than one section in a file. The key must be unique within a section, but need not be unique within the file. Indentation is not required, though it is typical. Whitespace, either a tab or an ASCII space, is often used to aide read-ability.

Section names, Key names, and Key values may contain whitespace, but will always begin and end with a printable character. The names and values are isolated with the open/close bracket. The key/value statement contains an “equals” sign.

The `openIni()` function accepts the name of a configuration file. It may be fully-qualified, but need only be sufficiently qualified so that it can be opened from the current working directory.

If the file exists, it is opened and read into memory. The file is then closed to help preserve the number of available file handles.

```
openIni :
procedure ( fileNamePtr ) iniHandle_t external ;
  declare   fileNamePtr      pointer;
end openIni ;
```

`openIni()` returns a pointer to the information as contained in the configuration file. That pointer (handle) is used with all other functions defined in this section. When the pointer returned is NULL, the named file does not exist, there was insufficient memory to store the data in the file or there was some other error condition detected.

The `closeIni()` uses the `iniHandle` provided to determine if any changes have been made to the data. If so, any modified data is written to the configuration file.

```
closeIni :
procedure ( iniHandle ) integer external ;
    declare    iniHandle      iniHandle_t;
end closeIni ;
```

The `closeIni()` function then frees all of the allocated memory and returns an integer. The return code will be zero unless the file could not be opened or closed successfully. Non-zero values returned correspond to the DOS error codes.

Using the `iniHandle` returned from `openIni()`, the `abortIni()` function allows the manager to release memory, but does not store any changes that may have occurred since opening.

```
abortIni :
procedure ( iniHandle ) external ;
    declare    iniHandle      iniHandle_t;
end abortIni ;
```

The `getFromIniData()` function searches the data referenced in the `iniHandle` for a key in the named section. When the key is found, the corresponding value is returned. If there is no key in the specified section, NULL is returned for the value.

```
getFromIniData:
procedure ( iniHandle, sectionNamePtr, keyNamePtr ) pointer external ;
    declare    iniHandle      iniHandle_t,
               sectionNamePtr pointer,
               keyNamePtr     pointer;
end getFromIniData;
```

The `storeIniData()` function is among the more complex in the library. This function allows the user to:

- Change a value of a key in a section
- Add a key/value to a section
- Add a section, then add the key/value to the newly created section

```
storeIniData:
procedure ( iniHandle, sectionNamePtr, keyNamePtr, valuePtr ) external ;
    declare    iniHandle      iniHandle_t,
               sectionNamePtr pointer,
               keyNamePtr     pointer,
               valuePtr       pointer;
end storeIniData;
```

The `iniHandle` defines which configuration data records are to be modified. The `sectionNamePtr` is the name of the section, while the `keyNamePtr` and `valuePtr` is the key/value pair definition.

The `removeFromIniData()` function will remove:

- A key/value pair

- An entire section

If the specified key is the only key in the specified section, the section is removed also. If `keyNamePtr` is `NULL`, the entire section is removed from the configuration data.

`removeFromIniData:`

```
procedure ( iniHandle, sectionNamePtr, keyNamePtr ) external ;
    declare    iniHandle    iniHandle_t,
               sectionNamePtr    pointer,
               keyNamePtr    pointer;
end removeFromIniData;
```

9.4 Memory buffer manager

Before the `malloc()` and `free()` functions were implemented in the library, I ported a set of functions written in C which managed a series of buffer pools defined statically in memory. In most embedded systems, dynamic memory allocation is a violation of the requirements document. Nonetheless, it is often necessary to pass information around the system.

The memory buffer manager serves as an elementary supervisor of a series of memory buffer pools.

Before a buffer may be requested, the pool should be initialized. While it is not good practice to request a buffer before the buffer pool definitions are initialized, that condition is handled in the code.

```
vInitializeBufferPool :
procedure external ;
end vInitializeBufferPool ;
```

Report the status of the buffer pools. It is the responsibility of the caller to correctly allocate the integer arrays based on the number of buffer pools reported.

```
iNumberBufferPools:
procedure integer external ;
end iNumberBufferPools;
```

One can get information about the buffer pools once initialized. The structure provides:

- size of the buffer in the pool
- number of buffers in the pool
- number of available buffers in the pool

```
declare bufferStatusStruct    literally '
    structure (
        bufferPoolSize        integer,
        totalBuffersPerPool    integer,
        availableBuffersPerPool    integer
    )';
```

Calling `vBufferPoolStatus()` will provide the information about each buffer pool as defined in the `bufferStatusStruct`.

```

vBufferPool Status:
procedure ( psBufferStatusPtr ) external ;
    declare psBufferStatusPtr pointer; /* to array of bufferStatusStruct */
end vBufferPool Status;

```

The `pvGetBuffer()` function is called to request a buffer from the pool. The input parameter is the maximum size needed for the buffer. It is the responsibility of the buffer pool requester to ensure that the buffer is not used to hold more than the maximum requested.

A pointer to the buffer is returned. If there is no buffer available of the desired size, NULL is returned.

```

pvGetBuffer:
procedure ( iBufferSi zeRequested ) pointer external ;
    declare iBufferSi zeRequested integer;
end pvGetBuffer;

```

To return a previously requested buffer to the pool, one calls the `eReturnBuffer()` function. If the buffer whose address is passed is not a member of the managed pools, an error is returned.

```

eReturnBuffer:
procedure ( pvBufferToReturn ) eBufferErrorCodes external ;
    declare pvBufferToReturn pointer;
end eReturnBuffer;

```

9.5 logging manager

The logging manager allows an application to write notes to a log file while running. The logging is based on the logging level (which may be modified as the application runs) defined in the statement in the code. There are five logging levels defined:

- FATAL
- ERROR
- WARNING
- ROUTINE
- DEBUG

The logging levels can be considered based on verbosity. For instance, if the log level was changed to FATAL, only those logging statements designated as FATAL would be written to the log file. Increasing the logging level (from FATAL to WARNING, for instance) would include all logging statements of FATAL, ERROR and WARNING, but not of ROUTINE or DEBUG. It should be noted that the default logging level is WARNING.

By default, there is no log file created, and all log statements are sent to the standard console output (stdout). A file may be opened, or closed, at any time during the life of your application.

The `i ni ti al i zeLoggi ng()` function closes an open log file, and sets the logging level to WARNING.

```

initializeLogging:
procedure external;
end initializeLogging;

```

The `openLogFile()` function closes a previously opened log file, then attempts to open the file specified by the `fileNamePtr`. This specification must be a NUL-terminated ASCII string. If that file already exists, it will be deleted before reopening.

```

openLogFile:
procedure( fileNamePtr ) external;
    declare fileNamePtr pointer;
end openLogFile;

```

The `closeLogFile()` function closes the current logfile, and changes the logging output to the stdout. The contents of the logfile are not disturbed when the file is closed.

```

closeLogFile:
procedure external;
end closeLogFile;

```

The `changeLogLevel()` function affects only the level, and does not affect any other feature.

```

changeLogLevel:
procedure ( newLogLevel ) external;
    declare newLogLevel LogLevel_t;
end changeLogLevel;

```

`addToLog()` is a variadic procedure. Its call is like that of `printf` (indeed, the `printf` engine is used to do the work of creating the log entry) In C, the interface would appear thusly:

```

void
addToLog( LogLevel_t LogLevel, char * fileName, int lineNumber,
    char * formatString, ... );

```

Since PL/M-86 does not support the notion of variadic functions, you are calling through a pointer, and not the actual function reference.

Like the `fprintf()` function, the `formatString`, and all parameters which follow, follow the same convention as `printf()`.

For example, consider this call from one of the library functions:

```

call addToLog( DEBUG, __FILE__, __LINE__,
    @( '[%d]:%bx', 0 ), index, byteArray( index ) );

```

It should be noted that the code macros `__FILE__` and `__LINE__` are not defined in the PL/M-86 language syntax, but are translated by the `plmPreProc.py` filter which is called before presenting the file to the compiler.

9.6 Morse code tools

One afternoon, a friend and I were talking about something that his daughter had learned one summer; Morse code. Remembering something that we needed for a project some 35 years earlier, I again went back into the wayback machine (with thanks to "Mister Peabody and Sherman") and captured the series of dots and dashes just as Samuel B. Morse taught me.

With Mr. Morse's code captured, and the rules of its use, I built some data tables and got to work. What follows are the functions to access and use that data according to the rules.

Using the `morseRenderChar()` function, the given ASCII character (letter or number/digit), provides the proper combination of dit and dah (in audio).

```
morseRenderChar:
procedure ( characterToRender ) external ;
    declare characterToRender char;

end morseRenderChar;
```

The `morseRenderCharTextual()` function, using the proper combination of ASCII dots and dashes to represent the dit and dah, renders the specified character.

```
morseRenderCharTextual :
procedure ( characterToRender ) external ;
    declare characterToRender char;
end morseRenderCharTextual ;
```

Given an ASCII character string, the `morseRenderString()` function generates the proper sequence of dit and dah until the string has been completely rendered.

```
morseRenderString:
procedure ( renderStringPtr ) external ;
    declare renderStringPtr pointer,
            renderString based renderStringPtr (*)char;
end morseRenderString;
```

Given an ASCII character string, the `morseRenderStringTextual()` function will generate the proper sequence of ASCII dot, dash and spaces until the string has been completely rendered.

```
morseRenderStringTextual :
procedure ( renderStringPtr ) external ;
    declare renderStringPtr pointer,
            renderString based renderStringPtr (*)char;
end morseRenderStringTextual ;
```

Given an ASCII string of dot and dash (along with its length), `decodeMorseChar()` will render the proper ASCII character, either alphabetic or numeric. If the sequence is not understood, an asterisk is returned.

```
decodeMorseChar:
procedure ( numberChars, stringInPtr ) char external ;
    declare numberChars word,
            stringInPtr pointer,
            dotsAndDashes based stringInPtr (*) char;
end decodeMorseChar;
```

Given an ASCII string of dot, dash and spaces, `decodeMorseString()` will translate the dot and dash sequence to ASCII character string.

```

decodeMorseString:
procedure ( stringInPtr ) public;
    declare    stringInPtr      pointer,
               stringIn based   stringInPtr (*) char;
end decodeMorseString;

```

9.7 Interpreting an Application's Command Arguments

While the DOS command line is limited to 127 characters, one can still convey a great deal of information to an application. The big challenge is to parse the command arguments consistently without replicating the logic in each application. While writing some Python applications, I started using one of its libraries. I found it easy to use, whether within the application, or while executing the application.

While using this library is easy, building the data to present can be a bit intimidating. We will use this example to present the API and provide discussion about the various points along the way. All the function prototypes and types discussed in the section are defined in `cmdArgs.txt`.

The `fprint` application (which is discussed in other sections of this document), has several different options which will affect the way the information is interpreted and presented. The data in the binary file may be presented as a byte, a word, or a dword. One may even choose to view the data in a scrollable window.

The real quandary is interpreting the user's preferences which may be specified in a random order, and to collect the name of the file whose contents are being presented. The application may be executed with a combination of the following options.

DOS(5.0)FPRINT, Version 10.02

Copyright (c) 2020, the ACME Software Deli

Execute as:

```
FPRINT [-h] [-v] [-b] [-w] [-d] [-no] [-s] [-if <INPUTFILE>]
```

Command Options:

```

-h      --help
        (optional) displays this information
-v      --verbose
        (optional) write all kinds of helpful information to the log file
-b      --bytes
        (default) contents of the file are interpreted and reported as unsigned 8-bit
        values
-w      --words
        (optional) contents of the file are interpreted and reported as unsigned 16-bit
        values
-d      --doublewords
        (optional) contents of the file are interpreted and reported as unsigned 32-bit
        values
-no     --noASCII
        do not report ASCII characters which may be in each block. the default is to
        report the ASCII characters, regardless of the number base or data size
-s      --scrollingWindow
        creating a scrolling data window
-if     --inputFile
        (SWITCH specification optional) specifies the name of the input file which will

```

be read, parsed, and presented

To present these options, we will present an array of structures, one structure for each option. The structure is defined thusly:

```
declare commandOption_t literally'
    structure (
        opti onType          opti onType_t,
        resul tsStorePtr     poi nter,
        shortSwi tchChars     poi nter,
        longSwi tchChars      poi nter,
        descri ptionString    poi nter
    )';
```

The first element in the structure is an `opti onType_t`, defined in an enumeration. The values in the enumerals are:

- **EXISTS**
The existence of the command switch is reported as a boolean at the address specified by `resul tsStorePtr`
- **NEXT_ARG**
Get the next argument from the command line, and store the string at the address specified by `resul tsStorePtr`. There is no verification for correctness of the string content.
- **NO_SWI TCH**
If there is a command argument with no associated switch, store the string at the address specified by `resul tsStorePtr`. There can be any number of these. The strings are stored in the order received, so the order specified remains significant to the caller

The second element is the address is a pointer, `resul tsStorePtr`, which is used as directed by the `opti onType`.

The next two strings specify the short name, and long name of command switches. The strings are formatted as a single token of one or more printable ASCII characters, terminated by a NUL. The switch delimiter (the hyphen in this example) is not specified as part of the string. If either switch is detected among the command arguments, the `opti onType` will be interpreted, and the proper value stored.

The final element of the structure is a NUL-terminated string which serves as a description of the command argument. The string is not used by the command interpreter, but is used when an incorrect command syntax has been discovered.

The example below is used in the `fprint` application.

```
declare argDefi ni ti ons (*) commandOption_t data
(
    EXISTS,          @hel p,    @( 'h' , 0 ),    @( 'hel p' , 0 ),
                    @( ' (optional) displays this information' , 0 ),

    EXISTS,          @verbo se, @( 'v' , 0 ),    @( 'verbo se' , 0 ),
                    @( ' (optional) write all kinds of helpful information to ',
                    ' the log file' , 0 ),
```

```

EXISTS,      @asByte, @( 'b', 0 ), @( 'bytes', 0 ),
    @( ' (default) contents of the file are interpreted and reported ',
        'as unsigned 8-bit values', 0 ),

EXISTS,      @asWord, @( 'w', 0 ), @( 'words', 0 ),
    @( ' (optional) contents of the file are interpreted and reported ',
        'as unsigned 16-bit values', 0 ),

EXISTS,      @asDword, @( 'd', 0 ), @( 'doublewords', 0 ),
    @( ' (optional) contents of the file are interpreted and reported ',
        'as unsigned 32-bit values', 0 ),

EXISTS,      @noASCII, @( 'no', 0 ), @( 'noASCII', 0 ),
    @( 'do not report ASCII characters which may be in each block. ',
        'the default is to report the ASCII characters, regardless ',
        'of the number base or data size', 0 ),

EXISTS,      @scrollingWindow, @( 's', 0 ), @( 'scrollingWindow', 0 ),
    @( 'creating a scrolling data window', 0 ),

NEXT_ARG, @inFileName, @( 'if', 0 ), @( 'inputFile', 0 ),
    @( ' (SWITCH specification optional) specifies the name of the input file ',
        'which will be read, parsed, and presented', 0 ),

NO_SWITCH,   @noSwitch1, NULL, NULL,
    @( 'a temporary holding place for arguments that are not specified with ',
        'a switch, and are not a switch', 0 )
);

```

To take advantage of an array of structures such as this, another structure captures the address of the array, along with the number of structure elements in the array, and the character used as the switch delimiter. In the fprint example, note that the switch delimiter is a hyphen.

```

declare commandOptionDefinition_t literally '
    structure (
        numberOptions    integer, /* length of the commandOptions array */
        commandOptions    pointer, /* your commandOption_t array */
        delimiterChar     char     /* 1 used for a short switch, 2 for long */
    );

```

In the fprint code, the following data declarations are made.

```

declare  commandOptionDef  commandOptionDefinition_t;

commandOptionDef.numberOptions = signed( length( argDefinitions ) );
commandOptionDef.commandOptions = @argDefinitions;
commandOptionDef.delimiterChar = '-';

```

It is this structure that is passed (by address, of course) to the command argument interpreter, `getCmdOptions()`.


```

getCmdOptions:
procedure ( numberArguments, argumentLi stPtr, optionsLi stPtr ) integer external ;
    declare numberArguments      integer,
            argumentLi stPtr      pointer,
            argumentArray based argumentLi stPtr    (*) pointer,
            optionsLi stPtr      pointer,
            optionsLi st      based optionsLi stPtr  commandOptionDefi ni ti on_t;
end getCmdOptions;

```

The first parameter, `numberArguments`, is the number of argument strings in the array that was created by parsing the command line. The number includes the name of the application, just as passed to function `main`.

The second parameter, `argumentLi stPtr`, is an array of pointers to the argument strings.

The third parameter, `optionsLi stPtr`, is the address of the command definition array that defines the options.

The `getCmdOptions()` function returns an integer whose value will explain the type of failure. This value may be interpreted by the error code API.

Finally, if you would like to report the command options as we did with the `fprint` application. Again, we use the `commandOptionDef`, just as we did with `getCmdOptions()`. However, rather than passing all of the command arguments, we pass only the first argument, the name of the application. The function `reportCmdOptions()` will report the “simple” name of the application, then each of the valid command arguments, including the description provided. However, rather than simply writing the description string, letting the console driver wrapping the string as it will, the string tokens are presented, with word wrapping managed by the reporting function.

```

reportCmdOptions:
procedure ( firstCommandArgument, optionsLi stPtr ) external ;
    declare firstCommandArgument pointer,
            optionsLi stPtr      pointer,
            optionsLi st      based optionsLi stPtr  commandOptionDefi ni ti on_t;
end reportCmdOptions;

```

9.8 stack manager

The stack manager is a last in, first out (LIFO) array, just like the stack of modern computing systems. It was written mostly as an exercise, but may also be used in an application. It follows the same rules as the array manager, though there is only one stack in an application. The stack is limited to 32767 words, just as with 16 bit DOS applications.

The `ini ti al i zeStack()` function must be called first to initialize the management variables.

The rest of the stack manager is based on the `push()` and `pop()` functions as one might expect. These functions handle 16-bit words. For the sake of convenience, there are three other pairs of functions:

- `pushByte()` and `popByte()`
- `pushPoi nter()` and `popPoi nter()`
- `pushDword()` and `popDword()`

The functions can be found in the `stack. ext` file.

10 Applications to Aid the Build Process

The building of the ACME Software Deli's DOS Library is easily done with the use of the `make` application, but to make the location of the Makefile as machine-independent as possible, several supporting applications are needed.

10.1 plmPreProc.py

PL/M-86 is an old language, designed to fast and lean. The original versions of the compiler would run on an ISIS-II machine with 24 kilo-bytes of RAM. The idea is that it would be used to create embedded microprocessor software, similar to the original PC's BIOS. Indeed, there were some electronic flight display systems written entirely in PL/M-86.

For these reasons, and perhaps others, there are no macros like `__FILE__` or `__LINE__` as exist in the C and C++ languages. However, the `l o g g e r` series of functions need the name of the file and the line number in the file for its logging statements, so the `pl mPreProc. py` script was written. The script does these things:

1. creates a output file with the same name as the input file, but with a `.PPP` extension
2. creates a string constant immediately following the module start into which the name of the input source file is placed
3. replaces `__FILE__` with a reference to the string constant
4. replaces `__LINE__` with the line number in the source

10.2 parseOutput.py

On 32-bit Windows machines, 16-bit software can be run in the CMD window. Because this is the case, when a 16-bit application (the Microsoft Assembler and PLM-86) returns an exit code other than zero, that condition is captured by `Make`, and the process stops.

However, on the 64-bit Windows machines, `DosBox` and `vDos` do not return an error code which `Make` can capture and act upon. To work around this problem, `parseOutput. py` was written. It parses the list file looking for the warning and error statements at the bottom of the file, returning:

- zero if no warnings or errors were reported
- one if warnings (but not errors) were reported
- two if errors were reported

10.3 rmSubStr

The `rmSubStr` application removes the specified sub-string from the designated string. For instance if we had the string `"C:\Documents\utils\source"` and we wanted to leave only `"utils\source"`, we would execute

```
rmSubStr "C:\Documents\utils\source" "C:\Documents\"
```

then capture the string written to the stdout by `rmSubStr`. This is a 32-bit application written in the C language.

10.4 8dot3path.cmd

One of the problem with running in the DOS environment is that the path names are limited to the 8.3 standard. Of course in the Windows environment, paths have no such limitation. Since the `MASM` and `PL/M-86` tools are DOS-based tools, the path must conform to the DOS path specification. This script was written to convert the Windows path to the DOS specification.

10.5 osArch.cmd

The `osArch.cmd` script uses the tools of Microsoft Windows to determine whether a 32-bit or 64-bit architecture is installed and running. There are conditional expressions in the `Makefile` which take advantage of this information.

10.6 mkDepend.py

Supporting the dependency lists can be tedious, so the mkDepend.py script was created. Originally written in AWK, then re-written in Python, this application reads a source file looking for files that are included. When it finds such a file, it adds the file name to the list, then opens the file it discovered, then repeats the process.

10.7 back2front

This script searches a file path, changing the backslashes (curse you, Microsoft) to the more traditional slash directory delimiter. Most of the make applications do not interpret the backslash character as a directory delimiter.

10.8 pwd

The pwd.exe application reports the full name of the current directory. It performs the same as the UN*X application.

10.9 objsToRsp.cmd

This script accepts a list of object filenames, creating a file used as a command to the librarian to add/replace objects to an object library.

10.10 cleanup.cmd

Originally written before businesses began using Windows 3.x, this script removes "temporary" files created during the development process. Generally, these are files that end in ".lst" and ".bak". While this script is not necessary for building the DOS library, it is quite useful for forcing a build of all object files.

10.11 refTouch

This application is a functional copy of the touch application I first used in UN*X. A feature that I added to mine was giving it the ability to change the date/time stamp of the target file (or files) to match the date/time stamp of a referenced file. If the target file does not exist, it will be created, and will have a length of zero.

10.12 echoEach.awk

The script accepts a line, echoing each element one line at a time.

10.13 objectToSource.awk

This script helps with the often tedious task of creating and maintaining the dependency lists. Because of the architecture developed in the DOS library, the dependency list for an executable looks like this:

```
acmeMenu.exe : acmeMenu.obj pl m86. l i b
```

The functionality of the DOS Library described herein is contained completely in the file named pl m86. l i b

11 Applications Created Using the Library

11.1 ACMEdir

Though not ready for actual use (there are some features not complete and some bugs to cleanup), the ACMEdir is the first application I have written for DOS using the POSIX-like directory functions.

The ACMEdir application is executed thusly:

Execute as:

```
ACMEDIR [-?] [-r] [-f <FILESPEC>] [-ar] [-vol] [-di r] [-hi d]
```

Command Options:

```
-?      --hel p  
(optional) displays this information
```

- p --pause
 (optional) print one page of the file list, then wait for the user to press a key to continue. <ESC>, 'X' and 'Q', and their lowercase equivalents, will discontinue the reporting of files and exit the program. Without this option, files are listed without pausing until all are listed
- f --filespec
 (switch optional) file specification for which to report
- dir --directories
 (optional) also report those entries which are directory names
- hid --hidden
 (optional) include those files which are hidden

Executing ACMEDIR -f *.exe will report like this:

DOS(5.16)ACMEDIR, Version x182
 (c)Copyright 2020-2021, the ACME Software Deli

The volume in Drive C is C_DRIVE

```

46,514  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\ACMEDIR.EXE
44,498  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\ACMEMENU.EXE
 6,434  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\CLRSCR.EXE
30,450  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\COLOR.EXE
38,242  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\DL.S.EXE
37,090  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\FPRINT.EXE
13,250  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\MACHTYPE.EXE
14,562  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\MOVINGBK.EXE
27,602  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\MRS2CHAR.EXE
 5,794  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\RD SKBUFS.EXE
13,266  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\REPKEYS.EXE
24,690  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\REPVOLS.EXE
22,882  2/10/21 12:28p C:\DOCUME~1\WORK\PLM86\TOMRSCOD.EXE
40,002  2/10/21 12:59p C:\DOCUME~1\WORK\PLM86\VIEWFILE.EXE

```

365,276 bytes in 14 files

Rather than presenting a number six or seven digit without commas, I decided that a comma would make that number MUCH easier to read and understand. Secondly, note that the file names are fully-qualified.

Since there are likely some changes coming, stay tuned.

11.2 ACMEmenu

A throwback from the days of old, when MS-DOS was the only operating system for the PC, and there were so few folks who could both type and remember commands to enter at the prompt.

the ACME Menu, Version 3.1 Copyright (c) 2020-2021, the ACME Software Deli		Thu Feb 11 12:35:14 2021 Page 1 of 4
A) Selection A, Page 1 B) Blue Foreground, Cyan Background C) Cyan Foreground, Blue Background D) DLS Listing E) ACMEdir Listing F) G) H) I) J) K) L) M)	N) O) P) Q) R) S) T) U) V) W) X) Y) Z)	
1 .. 4 to change page A .. Z to execute item ESC to exit		

The menu screen is composed of 26 possible selections (A - Z) on each of the 4 menu screens. Change the menu screen by typing the key 1, 2, 3 or 4.

To select an item on the screen, you may either type the associated letter, or you may use the cursor keys (up, down, left, right) to highlight the selection, the press the Enter key to make the selection.

To edit a selection, highlight that item, and press the F2 key. That will allow you to change the line that is displayed. This is a simple editor to change only the contents of the line. The following keys are accepted in this editor:

key	action
<home>	move to the beginning of the line
<end>	move to the end of the contents (not necessarily the end of the string)
<left>	move toward the beginning of the line
<right>	move toward the end of the line
<backspace>	remove the character to the left (and compress the string)
<delete>	remove the character at the cursor location (and compress the string)
<enter>	returns to the main screen saving the current contents
<esc>	returns to the main screen with no changes applied

To place a command (or a set of commands in batch file form) behind a selected line, press F4. This will open a simple text editor developed for the open source community and target to DOS. This editor was originally created as an article in "PC Magazine" sometime in the 1980's, and is called the Tiny Editor (TED). It is a simple, yet powerful, editor for creating scripts, or even applications.

To save the script, EXIT from the editor, and confirm. ABORT will allow you to leave the editor without saving the changes made during that editing session.

When a command is selected, it runs until completion. The ACME Menu then writes "Press Any Key to Continue" on the display, and waits for you to grant permission to continue.

There is not really any expectation that this will be used ever again, but it was a great exercise to create an application using a language that did not die, at least in my mind.

Pressing the F1 key will display this information in ASCII text format.

11.3 clrscr / color

In the early days (before the more modern cmd window), if one was to “clear the screen”, the character attributes were returned to grey characters on a black background. There are two applications to help with this problem.

First, I wrote an application that would allow the user to clear the screen, and then set the character attribute in each location on the screen with the desired color combination. Called `col or`, it was written in assembly language, and was not the most user friendly. The biggest problem was that the user had to know the numeric value which described his favorite color combination. Likely not a favorite.

Next came `cl r`, which would read the character attribute at the screen origin (0,0), then use that to clear the screen. Again, written in assembly language. Very small, VERY fast. The modern version is written in PL/M-86, and is called `cl rscr`.

However, the biggest problem to solve was allowing the user to specify his colors using abbreviations. A second version of `color` allowed the user to specify the foreground and background (and border on CGA and EGA screens) colors using a two letter designator. Much better. Also written in assembly language, with both a resident version (`TSRcol or`) and a non-resident version (`col or`), so maintenance was not as easy as what I have now, but with a MUCH smaller disk footprint.

As an aside, the TSR version could be executed again, with a different color combination. The color attribute in memory would be changed to the new combination.

You might try the latest version, written in PL/M-86. It is executed thusly:

DOS(5.16)COLOR, Versi on 3.1

Copyri ght(c) 2020-2021, the ACME Software Deli

Execute as:

```
COLOR [-?] [-fg <FOREGROUND>] [-bg <BACKGROUND>] [-list]
```

Command Opti ons:

```
-?      --hel p
        (optional) di splay s thi s i nformation
-fg      --foreground
        (swi tch optional) speci fi es the foreground color
-bg      --background
        (swi tch optional) speci fi es the foreground color
-list    --listColors
        (optional) shows color codes and any restrictions on their use
```

Bk	Black	Dg	DarkGrey
Bl	Blue	Bb	BrightBlue
Gr	Green	Bg	BrightGreen
Cy	Cyan	Bc	BrightCyan
Rd	Red	Br	BrightRed
Mg	Magenta	Bm	BrightMagenta
Bn	Brown	Ye	Yellow
Gy	Grey	Wh	White

The first 8 colors (Black .. Grey) are valid for either foreground or background. The final 8 colors (DarkGrey .. White) are valid only in the foreground. Specifying the same color for the foreground and background is not valid.

11.4 dls

Much like the ACMEdir application, the dls (DOS list files) application presents the information like the standard (DOS) dir command, but updated to allow the user to pause the output when the screen has been filled. The presentation of the date is a more modern format and the file size is represented in comma-separated numbers.

It is executed thusly:

```
DOS(5.16)DLS, Version x205
(c)Copyright 2020, the ACME Software Deli
```

Execute as:

```
DLS [-?] [-p] [-r] [-hid] [-f <FILEMASK>]
```

Command Options:

```
-?      --help
        (optional) displays this information
-p      --page
        (optional) print one page of the file list, then wait for the user to press a
        key to continue. "X" and "Q", and their lowercase equivalents, will discontinue
        the reporting of files and exit the program. Without this option, files are
        listed without pausing until all are listed
-hid    --hidden
        (optional) include those files which are hidden
-f      --fileMask
        (switch optional) specifies the filename mask (following the OS rules)
```

So, if one wanted a list of the files with the "a86" extension, it could be specified in these ways:

```
"dls -f *.a86" or "dls *.a86"
```

Of course with either specification, the optional -p switch could be chosen to pause the presentation one screen at a time.

Sample output:

```
DOS(5.16)DLS, Version x205
(c)Copyright 2020, the ACME Software Deli
```

```
The volume in Drive C is C_DRIVE
Directory of C:\DOCUME~1\WORK\PLM86
```

```
2,192  1/29/21 12:54p ABSDREAD.A86
5,206  1/29/21 12:54p ASMSTART.A86
2,797  1/29/21 12:54p ASMTTEST.A86
1,382  1/29/21 12:54p COMPASM.A86
2,399  1/29/21 12:54p GETREGS.A86
5,398  1/29/21 12:54p HEXASCII.A86
6,421  4/26/20  6:11p ICPUID.A86
```

```

1,184 1/29/21 12:54p INT8ASM.A86
2,425 1/29/21 12:54p LDXCASM.A86
1,414 1/29/21 12:54p LODWORD.A86
1,434 1/29/21 12:54p MODDI V.A86
5,906 1/29/21 12:54p SHRTSTRT.A86
4,789 1/29/21 12:54p STARTSUP.A86
2,797 1/29/21 12:54p SYSCAASM.A86
53,100 1/29/21 12:54p TED.A86
98,844 bytes in 15 files

```

In this presentation, there are so few assembly language source files, the “-p” switch has no effect. However, if we were to get a listing of PL/M-86 source files (*.p86), the pause switch could be quite helpful.

DOS(5.16)DLS, Version x205

(c)Copyright 2020, the ACME Software Deli

The volume in Drive C is C_DRIVE

Directory of C:\DOCUME~1\WORK\PLM86

```

12,631 1/29/21 12:54p ACMEDI R.P86
27,329 2/01/21 11:14a ACMEMENU.P86
13,031 1/29/21 12:54p ARRAYMGR.P86
7,859 1/29/21 12:54p BI TARRAY.P86
11,959 1/29/21 12:54p BUFFMGR.P86
2,683 1/29/21 12:54p CHANGEXT.P86
2,041 1/29/21 12:54p CHDI R.P86
2,189 1/14/21 11:26a CLKTI CK.P86
1,593 1/29/21 11:08a CLRSCR.P86
22,948 1/29/21 12:54p CMDARGS.P86
8,897 2/01/21 1:54p COLOR.P86
2,983 1/29/21 12:54p CTYPE.P86
2,744 1/29/21 12:54p CURDI R.P86
2,169 1/29/21 12:54p CURDI SK.P86
10,410 1/29/21 12:54p DATE.P86
16,532 1/29/21 12:54p DATESTR.P86
7,620 1/29/21 12:54p DEBUGHLP.P86
4,204 1/29/21 12:54p DIRFUNCT.P86
16,555 1/29/21 12:54p DLS.P86
7,488 1/29/21 12:54p DOSFI ND.P86
2,540 1/29/21 12:54p DPARMBLK.P86
1,813 1/29/21 12:54p DRVLOGNM.P86
1,763 1/29/21 12:54p DTAMODLE.P86

```

-- More --

11.5 fprint

The fprint application is one of the first applications that I ported from the Intel Development Systems (IDS) to DOS. Nearly all of the porting effort was to create a DOS library which resembles the Intel iNDX OS. As I learned more languages, I ported this to each language as a learning exercise. Languages such as Pascal, C, x86 assembler, Ada (yes, that Ada) and C++. I even wrote a version in AWK and Python.

This version is arguably one of the more complex versions written. The command options are:

Execute as:

FPRI NT [-?] [-v] [-b] [-w] [-d] [-no] [-s] [-i f <INPUTFILE>]

Command Options:

- ? --help
(optional) displays this information
- v --verbose
(optional) write all kinds of helpful information to the log file
- b --bytes
(default) contents of the file are interpreted and reported as unsigned 8-bit values
- w --words
(optional) contents of the file are interpreted and reported as unsigned 16-bit values
- d --doublewords
(optional) contents of the file are interpreted and reported as unsigned 32-bit values
- no --noASCII
do not report ASCII characters which may be in each block. the default is to report the ASCII characters, regardless of the number base or data size
- s --scrollingWindow
creating a scrolling data window
- i f --inputFile
(SWITCH specification optional) specifies the name of the input file which will be read, parsed, and presented

The file data is presented in Hexadecimal and ASCII (when printable). The application will present data thusly:

DOS(5.16)FPRI NT, Version 10.02

Copyright (c) 2020, the ACME Software Deli

Representation of file: fprint.exe

00000000	4d 5a c4 01 2f 00 09 03 e0 00 2f 02 ff ff d3 06	MZ../...../.....
00000010	72 05 00 00 00 00 00 00 1e 00 00 00 01 00 03 00	r.....
00000020	00 00 06 00 00 00 16 00 00 00 1b 00 00 00 26 00&.
00000030	00 00 2c 00 00 00 34 00 00 00 3a 00 00 00 42 004...:...B.
00000040	00 00 49 00 00 00 52 00 00 00 36 05 09 00 3a 05	..l...R...6....
00000050	09 00 3e 05 09 00 10 00 09 00 14 00 09 00 1a 00	..>.....
00000060	09 00 1e 00 09 00 22 00 09 00 26 00 09 00 2c 00"....&....
00000070	09 00 30 00 09 00 34 00 09 00 38 00 09 00 3e 00	..0...4...8...>.
00000080	09 00 42 00 09 00 46 00 09 00 4a 00 09 00 50 00	..B...F...J...P.
00000090	09 00 54 00 09 00 58 00 09 00 5c 00 09 00 62 00	..T...X...\...b.
000000a0	09 00 66 00 09 00 6a 00 09 00 6e 00 09 00 74 00	..f...j...n...t.
000000b0	09 00 78 00 09 00 7c 00 09 00 80 00 09 00 86 00	..x...
000000c0	09 00 8a 00 09 00 8e 00 09 00 92 00 09 00 98 00
000000d0	09 00 9c 00 09 00 a0 00 09 00 a4 00 09 00 aa 00
000000e0	09 00 b6 00 09 00 44 05 09 00 90 05 09 00 96 05D.....

000000f0	09 00 b8 05 09 00 f9 05 09 00 ff 05 09 00 0e 06
00000100	09 00 14 06 09 00 7d 06 09 00 83 06 09 00 94 06}......
00000110	09 00 02 07 09 00 08 07 09 00 1f 07 09 00 25 07%.
00000120	09 00 31 07 09 00 37 07 09 00 84 07 09 00 a8 07	..1...7.....
00000130	09 00 d5 07 09 00 db 07 09 00 30 08 09 00 34 080...4.
00000140	09 00 70 08 09 00 76 08 09 00 82 08 09 00 88 08	..p...v.....
00000150	09 00 96 08 09 00 a8 08 09 00 ae 08 09 00 de 08
00000160	09 00 0a 09 09 00 18 09 09 00 1e 09 09 00 28 09(.
00000170	09 00 2b 09 09 00 3c 09 09 00 42 09 09 00 49 09	..+...<...B...l.

Up/Down (by line) -- PgUp/PgDn (by page) -- Home/End (top/bottom)

The Up/Down arrow keys will move backward and forward (respectively) in the file one line at a time. PgUp/PgDn will move backward and forward (respectively) in the file one screen at a time, depending on the number of lines on the DOS display window. Home/End moves to the beginning and end of the file. The <Esc> key will exit the application.

[11.6 repKeys](#)

[11.7 repVols](#)

[11.8 rdskbufs](#)

[11.9 view file](#)

[11.10 Encoding & Decoding Morse Code](#)