SECTION III.a

SSP System Interface

3.1. Introduction

This section covers IBM System/36 specific aspects of the program development environment. It covers performing file I/O under the System Support Program (SSP), making of an SSP system call, and the translation of C to IBM System/36 assembler. It also describes the SSP system interface functions, which are documented in "manual page" form. Although these functions are documented here (because they are SSP-specific), they are packaged in the "hidden" Base Library libb.

The object form of the Base Library is normally found in all standard compiler packages under the name **libbxx.yy**, where **xx** denotes configuration dependencies (such as floating point, memory model, etc.), and **yy** specifies the processor.

The Base Library is a part of all libraries under System/36. There are two libraries built for System/36. #CALIB provides all of the ANSI environment functions. #CEALIB provides all ANSI plus Extended ANSI environment functions. All libraries also provides internal library support functions.

All the routines in this section are C-callable, but calling them directly from within applications is discouraged since they are subject to change. The names of the functions are reserved, because they begin with leading underscores _, and there is usually a routine in one of the higher-level libraries that can be called to do the same job. However, if you are adding more functionality to the system interface (such as a "change mode", "set date", or "set time" function), the Base Library is the place to put it.

Application programmers should be familiar with the section "Using the ANSI Library".

All examples in this section are for illustrative purposes only, as these functions are not generally meant to be called by user applications.

3.2. Using Files Under SSP

The system interface to SSP is designed to be transparent to users that program in a portable style, yet powerful enough to give the IBM-oriented user access to various features under SSP.

All I/O is done through files, where a file currently can be a printer or a sequential file on the fixed disk. Although disk files are sequential, using the GAM access method in the system interface allows applications to access records in a random order via calls to lseek. Note that random access is currently only supported on files opened as binary files, as explained below.

Files under SSP are made up of fixed length records. The system interface converts this structure to and from a byte stream. For a binary file this simply means ignoring the record boundaries. For a text file it also implies separating the records (lines) with a newline character.

With a few exceptions, data of arbitrary length can be read or written. For example, a file containing ten 100-byte records can be copied with 1000 reads and writes specifying one byte, or it can be copied with on single read/write sequence specifying 1000 bytes.

Note: All references to functions in the SSP system interface library, apply to similar functions in the ANSI library.

Text files

All files are by default considered to be text files, as long as the O_BIN flag is not set in the **oflag** parameter to **_open**. Text files are assumed to contain printable text.

A newline character causes **_write** to terminate the current output record. The rest of the record will be filled with spaces and the newline itself will never be written.

The record will also be output if the record length is exceeded.

A call to **_read** returns the number of characters asked for, plus the remaining contents of the current input record, plus one more character (the newline character). A newline is inserted when a complete record has been read.

Text files can be accessed only sequentially. **_lseek** is currently not supported on text files.

Binary files

A file is considered to be a binary file if the O_BIN flag is set in the **oflag** parameter to **_open**.

Binary files contain an unstructured sequence of bytes. There is no record structure visible to the program, and all character codes are allowed. In the physical files data is packed in records, with no interpretation at all. The

last record written will be zero-padded, unless the number of characters written is a multiple of the record length.

Filenames

Filenames, as given to the functions **_open** and **_remove**, have the following form:

[drivername:]filename

The **filename** is decoded by the drivers . For files on a fixed disk, it refers to the NAME parameter in the FILE OCL statement in the case of fixed disk files. If **drivername** is omitted, the default is "DSK". The length of a **drivername** or a **filename** may never exceed eight characters.

I/O Drivers

In the system interface there are several drivers that support different types of files. The following drivers are currently supported:

DSK Access a fixed disk file. A filename must always be specified and the filename must match the NAME parameter in the FILE OCL statement. You may specify a record length, but if an existing file is accessed its record length is used. All files created will be of sequential type but may be accessed randomly if opened or created as binary files.

NULL The data-sink. Returns immediate end-of-file on read. Data written is thrown away, 512 bytes at a time.

PRT Write to the printer. The filename is by default "PRTFILE", but may be specified otherwise. The O_WRONLY flag must be set in the **oflag** parameter to **_open**.

SYSIN Read from system input; i.e. the display station keyboard or a procedure member being executed. The O_RDONLY flag must be set in the **oflag** parameter to **_open**. End of file is reached if a line beginning with /* is read.

SYSLOG Write to the display station. The filename is always ignored. The O_WRONLY flag must be set in the **oflag** parameter to **_open**. If the record length is defined as not equal to zero, all output is also written to the history log file.

SYSLIST Write to the display station or system printer. The filename is always ignored. The O_WRONLY flag must be set in the **oflag** parameter to **_open**. The SYSLIST OCL statement will change the system list output to a printer or the display station, or will not list the output at all.

Using the Disk

The following examples show how to perform some disk operations. All declarations and definitions are excluded, but it is assumed that **<wslxa.h>** and **<fcntl.h>** have been included, all variables are declared with the appropriate

type, and all calls is done from main. After every C example follows one or two examples of necessary OCL-statments.

Create a binary file with the name "MYFILE" and a record length of 100:

```
fd = _open("DSK:MYFILE", O_CREAT|O_TRUNC|O_BIN|O_RDWR, 0, "recl=100");
```

Note: The O_TRUNC flag is the only flag required to force the creation or truncation of a file. Due to SSP protection mechanisms and the current inability of the system interface to test the existence of a file, the O_CREAT flag to be relevant on existing files only. The test of existence is relevant, since O_OFLAG alone may never truncate an existing file. To avoid unwanted truncation, files may only be created if O_TRUNC is set.

To create a new file, add the following OCL-statement to the previous example before running it. We assume that 200 blocks are needed for the file.

```
// FILE NAME-MYFILE,BLOCKS-200,DISP-NEW
```

To truncate an existing file, the OCL-statement may look like this:

```
// FILE NAME-MYFILE,DISP-OLD
```

To open the previously created file:

```
fd = _open("MYFILE", O_BIN|O_RDWR, 0, 0);
```

Note: The example does not specify the record length again since the file already exists. The driver specification is also omitted since the disk driver is choosen by default.

The OCL may be:

```
// FILE NAME-MYFILE
```

To open an existing text file for reading with the name "REPORT":

```
fd = _open("REPORT", O_RDONLY, 0, 0);
```

The OCL may be:

```
// FILE NAME-REPORT
```

Using the Printer

The following sample program shows how to open the printer device, write out the text "Hello world!" on the first row of page one, perform a formfeed, write out "Hello..." on the first row of page two, overwrite it with "XXXXXXXX", and finally close the printer. Note: An OCL-statement is usally not needed for the printer.

```
#include <wslxa.h>
#include <fcntl.h>
```

```
main()
    {
    FD fd;

    fd = _open("prt:", O_WRONLY, 0, 0);
    _write(fd, "Hello world!\n\fHello...\rXXXXXXXX\n", 32);
    _close(fd);
    }
```

Using the ANSI Library

Most of the functions in ANSI the library work as documented in the ANSI standard or, if not supported, fail in a predictable way. The fopen function requires special explanation. To define fopen in a C program, write:

```
#include <stdio.h>
FILE *fopen(fname, type)
    RDONLY TEXT *fname, *type;
```

One problem is the use of the type, argument which points to a string that contains one of the following valid sequences:

r w a rb wb ab r+ w+ a+ r+b w+b a+b

Using the append sequence (i.e., strings containing an a), to open a file that does not exi
s
functions are supposed to work, the system interface would have to test if the specified file exists or not before opening it, but since this functionality is currently no supported under SSP, the function will fail and execution halts.
Do not use "append" to new files.

The string in type may have a second part separated by a comma. This part is passed to _open as its last parameter (xparm). You may specify a record length in the form of a string of the format recl=n where n can be one or more digits. The following example shows how to create a binary file with a record length of 234:

```
fp = fopen("MYFILE", "w+b, recl=234");
```

3.3. Workstation Input and Output

Workstations are not handled by the normal stream I/O model as are the other I/O functions. The workstation screen is read and written one screen at a time, or one or more fields at a time. The layout of the screen is defined by a "format load member", which must be created before your program can use it.

There are funve functions to handle workstation I/O, all of which are in the source file **wsio.c**. All those functions use the structure definition DTFW, which is defined in the header file **<dtfw.h>**.

wsopen allocates and opens the requested workstation. It will also get a format load member. The workstation DTF (DTFW) used as a parameter to **wsopen** must be initialized by the constant DTFW_INIT before the workstation is opened.

wswrite selects a screen format from the format load member, and writes the output buffer to the output fields on the screen.

wsread reads the input fields from the screen into a buffer. This buffer must at an address aligned to an even eight-byte boundry.

wsalign aligns a read buffer address for **wsread**. The buffer must be at least seven bytes bigger than the maximum size used.

wsclose will close the workstation.

The following example will read and write to the workstation, and end if *** is typed into the key field.

```
#include <wslxa.h>
#include "dtfw.h"

/*  input and output buffer
 */
typedef struct {
    TEXT key[5];
    TEXT descr[21];
    TEXT quant[5];
    TEXT price[8];
    } WSBUF;

/*  workstation DTF
 */
DTFW dtfws = DTFW_INIT;
/* 16 byte work area after DTF for each format */
UTINY dtfwrk[16] = {0};

/* i/o buffer
 */
TEXT buf[BUFSIZ + 7] = {0};
```

```
/*  write and read workstation
 */
BOOL main()
    {
    IMPORT VOID *wsalign();
    WSBUF *wsbuf;

    wsbuf = wsalign(buf);    /* get aligned buffer */
    wsopen(&dtfws, "DFS");   /* open ws, format member is DFS */
    wswrite(&dtfws, wsbuf, BUFSIZE, "F1");
                             /* write to ws, format is F1 */
    for (;;)
        {
        wsread(&dtfws, wsbuf, BUFSIZE)       /* read from ws */
        if (cmpbuf(wsbuf->key, "***", 3))    /* test if end */
            break;
        wswrite(&dtfws, wsbuf, BUFSIZE, "F1");  /* rewrite to ws */
        }
    wsclose(&dtfws);     /* close workstation */
    return(0);
    }
```

The workstation format is defined by:

```
    SF1
    DDF0001    23 2 5Y                              Y
    DKEY        5 229Y  Y  Y                       Y
    DFL0003    11 4 5Y
    DFL0004     8 430Y
    DFL0005     5 445Y
    DDESCR     21 5 5Y  Y                              Y
    DQUANT      5 531Y  YN    B                        Y
    DPRICE      8 545Y  YN          Y                  Y
```

CItem Key (*** to end)

CDescription
CQuantity
CPrice

SEE ALSO

For more information on workstatin I/O, see the wsalign, wsclose, wsopen, wsread, and wswrite manual pages in this section of the manual, along with the following IBM publications: IBM System/36 Programming with Assembler (SC21-7908), IBM System/36 Creating Displays: Screen Design Aid and System Support Program (SC21-7902).

3.4. Making a SSP System Call

The Whitesmiths system interface for System/36 provides the assembler routine _svc to access the System Support Program (SSP) via a supervisor call. _svc is documented in manual page form in this section of the manual. The SSP function calls supported are documented in the S/36 Functions Reference Manual and the S/36 System Data Areas.

_svc is a C callable function and was created in order to make a clean and flexible interface to SSP. _svc accepts a variable number of arguments, which are used to set up the supervisor call to SSP. There is no general way of knowing if a call was successful or not. Information is needed about how that particular call indicates success and failure. Every function in the system interface that uses the _svc function also has that information and signals an error by setting the variable **errno** and returning a bad value which often is -1. (See the ssp.h header file manual page in this section of the manual for more information on errno.)

The following sample program uses the SSP system service function SNAP to produce a memory dump on the printer of the sample programs first 1024 bytes in main storage.

```
/*  PROGRAM DOSNAP to dump main storage memory on the printer
 */
#include <wslxa.h>
#include <ssp.h>

struct
    {
    UTINY options;
    TEXT *low;
    TEXT *high;
    TEXT dumpid[4];
    }
snpp =
    {
    0,
    (TEXT *) 0,
    (TEXT *) 1023,
    'D', 'E', 'M', '0'
    };

main()
    {
    _svc(1, 2, 4, 1, 17, &snpp);
    }
```

All supervisor calls consist of three parts; an R-byte to specify the type of supervisor call, an Q-byte to further specify the requested function, and an inline parameter list. The _svc system interface function expects the R-byte in argument four, the Q-byte in argument three, and the inline parameter list in argument five to seven depending on the value of argument one. Argument

two specifies whether or not the index registers should be loaded with the values of the one to two last arguments.

The sample program above first specifies to **_svc** that it wishes to pass one inline parameter by the value 1 in argument one and then that index register two should be loaded by the value 2 in argument two. To enter SSP, the Q-byte should have the value 4 and the R-byte the value 1 in arguments two and three respectively. To further access the SNAP Function, SSP needs an inline byte with the value 12, which the program places in argument five. Finally, the last argument is an address of a control block which will be loaded down in index register two and read by the SNAP function in order to get the dump parameters from the control block and perform the required operation.

After _svc is executed, control returns to the calling function and execution continues. Normally some sort of completion code is returned in the control block, but this is not the case after a call to SNAP. It is up to the user of _svc to check if the operation was successful or not.

Whitesmiths provides support for the six SSP function calls listed in the header file <ssp.h> via _svc. Examples of the use of these calls can be found throughout the source to the system interface library. While _svc should work for any of the other SSP function calls, not all of them are explicitly supported by means of a system interface subroutine.

3.5. Compiler Interface to Assembly Language

Programs written in C for operation on IBM System/36 under SSP are translated into assembly language according to the following specifications:

External Identifiers

External identifiers may be written in both uppercase and lowercase, but will be translated to uppercase only. The first six characters must be distinct, since all identifiers will be cut to six characters. Any existing underscore appearing in the first six characters of the identifier is left alone and no underscore will be added.

Function Text

Function text is generated into the same segment as data. This is because the S/36 assembler does not make any distinction between program text and data. External function names are published via **ENTRY** declarations.

Literal Data

Literal data is generated into the same segment as function text and data. (see the note on function text below). External function names are published via **ENTRY** declarations.

Initialized Data

Initialized data is generated into the same segment as function text. External data names are published via **ENTRY** declarations. Declarations with an initial value of zero may be generated as the a DS (**define storage**) declaration if the creation of bss symbols in the compiler is not disabled.

Uninitialized Declarations

Uninitialized declarations result in a DS (**define storage**) declaration as long as the creation of bss symbols in the compiler is not disabled. If the creation of bss symbols is disabled, or if the size of the data element being declared is unknown, an **EXTRN** declaration will result. One instance will appear per program file.

Function Calls

Function calls are performed by:

1) moving arguments to the memory area below the current stack frame, right to left. Character data is sign-extended to integer; float is converted to double.

2) reserving the stack frame for the calling function including save area (four bytes), auto and temporary storage, and arguments and space for a struct return area if this is required. This frame is reserved by loading XR1 with the size of the frame negated.

3) calling via "B FUNC", The called function will then decrement the frame pointer in memory (@FP) with the value in XR2, and then save the return address and old frame pointer.

The only registers preserved on return from a function call are @AC0 and @FP which is also copied into XR1. The returned value is in @AC0 (char sign-extended to integer, float widened to double), or in the designated stack area in the case of a structure or union.

Example of an assembly language program calling a C function:

```
        MVC   245(2,#XR1),ARG3+1    move argument 3 to stack
        MVC   243(2,#XR1),ARG2+1    move argument 2 to stack
        MVC   241(2,#XR1),ARG1+1    move argument 1 to stack
        LA    -10,#XR1              args (6 bytes) + save area (4 bytes)
        B     FUNC                  call function
```

Stack Frames

Stack frames are maintained by each C function, using the in memory register @FP as a frame pointer. On entry to a function, @FP is added to XR1 to give a new value for @FP. Before XR1 is copied to @FP the ARR (address recall register) is saved at offset 248(XR1) and the old @FP is saved at offset 246(XR1). This is done by a call to the funtion @CENT. Arguments to the function are at 250(@FP), 252(@FP), etc. Auto storage may be reserved on the stack at 248(@FP) on down. To return, the jump "B @RET" restores @FP, XR1 and the return address. If stack checking is requested (via the -ck flag to p2), the call "B @CENTS" will behave as does the call to @CENT, and in addition ensure that adding the quantity in XR1 to @FP will not cause the stack to wrap around or go below the value in the C variable _stop.

Example of an assembler function called by a C program:

```
FUNC    START 0
        LA    LABEL1,#XR2           ret address for @CENT
        BD    @CENT                 save registers
LABEL1  EQU   *
        MVC   ARG1+1(2),251(,#XR1)  get arg 1
        MVC   ARG2+1(2),253(,#XR1)  get arg 2
        MVC   ARG3+1(2),255(,#XR1)  get arg 3
*
*       assembler function
*
        MVC   @AC0+7,RETVAL+1       return value
        B     @CRET                 restore registers and return
*
ARG1    EQU   *
        IL2'1'
ARG2    EQU   *
        IL2'2'
ARG3    EQU   *
        IL2'3'
RETVAL  EQU   *
```

IL2'0'

Data Representation

All machine instruction addressing in the S/36 architecture is to the last (highest) address of a data object. This is not true in the case of a floating number, however. Floating variables are always addressed by the address to their first (lowest) byte. To be consistent, pointers to data objects generated by the C compiler are always to the first byte of the object and the compiler adds an the size of the data object minus one as an offset before using the pointer in any operation. Integer is the same as short; two bytes stored most significant byte first. Long integers are stored as four bytes, most significant byte first.

All signed integers are twos complement. Floating numbers are represented as required by the IBM scientific macroinstructions, which are the same as the IBM S/370 standard (four bytes for float, eight for double) and are stored in ascending order of significance. This format is also used by the S/36 Fortran compiler.

Storage Bounds

No storage bounds need to be enforced.

Module Name

Module names are not used.

3.6. SSP System Interface Manual Pages

The following "manual pages" document the library functions of the SSP system interface. For an explanation of the organization and effective use of manual pages, see section 1.2.6, entitled "Manual Page Conventions".

NAME

ssp.h - SSP header file

SYNOPSIS

#include <ssp.h>

FUNCTION

All standard system library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a system header file, **<ssp.h>**, at the top of any program making SSP system calls or using the internal control structures of the system interface. This header is used in addition to <wsltyp.h>, <wslxa.h>, or <std.h>.

When a SSP system interface function fails, the global variable **errno** is set and the function returns an error indicator, which is -1 for most of the functions. The following is a list of interface specific values for **errno** and what they mean. Note: Severe I/O errors will halt the execution of your application, an error message will be displayed, and you will be prompted for a response. For more information, see the S/36 System Messages Manual.

Name	Value	Description
EBADFD	1	bad file descriptor
EBADNM	2	bad filename
EOPEN	3	open error
ENODRVR	4	no such driver
EBADMODE	5	bad mode
EMFILE	6	too many open files
ESEEK	7	bad call to lseek
EOEXCL	8	open exclusive not supported
ENOTSUP	9	function not supported
ENOMEM	10	no memory

<ssp.h> defines various useful system parameters. The following is a summary of system call parameters used by the system interface and what they do. For more information, see the S/36 Functions Reference Manual, the **S/36 System Data Areas** manual, and examine **<ssp.h>** directly.

Name	Value	Description
SSP_ALOC	1,2,0x4,0x1,0xc	allocate file or device
SSP_OPEN	1,2,0x4,0x1,0x2	open file or device
SSP_CLOS	1,2,0x4,0x1,0x3	close file or device
SSP_DREQ	1,2,0xc,0x1,0x0	disk data management request
SSP_PREQ	1,2,0xc,0x1,0x8	printer request
SSP_EOJ	1,0,0x4,0x1,0x4	end of job

System calls to SSP are performed using the function **_svc**. Communication with _svc is made with arguments like those above, and often with one additional argument containing an address of a control block.

The following miscellaneous definitions are also set up by <ssp.h>:

Name	Value	Description
NFAIL	-1	standard failure return code
NFILES	16	maximum number of open files
_DEFNAM	"DSK"	default driver name
_NAMSIZ	8	max name length of files and drivers

NAME

stack - define the stack size for a C program

SYNOPSIS

Assembler file **stack.asm**

```
*************************************************************************
EFRAME   EQU   *
         DS    48CL1          change this to change exception stack size
*                             now 48 bytes
*                             every exception takes at least 12 bytes
EFREND   EQU   *
*************************************************************************
         ORG   *,8,0
_MEMRY   EQU   *
         DS    4096CL1        change this to change heap+stack size
*                             now 4096 bytes
@STTOP   EQU   *
*************************************************************************
```

FUNCTION

This assembler file is provided for defining the stack + heap size and exception stack size. The first executable code in this routine is a jump to the startup routine. The stack is used for all automatic data used by the C program. The stack starts at address @STTOP and grows downwards in memory. The heap is used by the routine **sbreak** for dynamic memory allocation, and **sbreak** is in turn used by other memory management routines. The default stack + heap size is 4096 bytes, but can be changed if you need a different stack size. The exception stack is used by the routine **_when** to store exception handler frames. If you get the message "exception stack too small" while running a C program you must increase the size ot the exception stack.

NAME

wsalign - align buffer address used by wsread

SYNOPSIS

```
VOID *wsalign(buf)
        VOID *buf;
```

FUNCTION

The read buffer address for wsread must be aligned to an even eight byte boundry. wsalign will align a address given to it as an input parameter. Note that the input buffer must be at least seven bytes bigger than its maximum usable size.

RETURNS

wsalign returns the aligned address.

SEE ALSO

wsread

NAME
wsclose - close a workstation

SYNOPSIS
```
COUNT wsclose(dtfw)
        DTFW *dtfw;
```

FUNCTION
wsclose closes the workstation defined by dtfw.

RETURNS
wsclose returns the SSP CLOSE completion code.

EXAMPLE

SEE ALSO
wsopen

NAME

wsopen - open a workstation

SYNOPSIS

```
COUNT wsopen(dtfw, member)
        DTFW *dtfw;
        TEXT *member;
```

FUNCTION

wsopen allocates and opens the workstation defined by dtfw. To do this it requires the format load member with name member.

RETURNS

wsopen returns the SSP CLOSE completion code.

EXAMPLE

SEE ALSO

wsclose

NAME

wsread - read from a workstation

SYNOPSIS

```
COUNT wsread(dtfw, rcad, inlen)
        DTFW *dtfw;
        TEXT *rcad;
        COUNT inlen;
```

FUNCTION

wsread reads from the workstation defined by **dtfw** into a buffer with record address **rcad**. The **rcad** address must be aligned to an even eight byte boundary. A maximum of **inlen** characters are read.

RETURNS

wsread returns the workstation completion code.

SEE ALSO

wsalign, wswrite

NAME

wswrite - write to a workstation

SYNOPSIS

```
COUNT wswrite(dtfw, rcad, outlen, format)
        DTFW *dtfw;
        TEXT *rcad;
        COUNT outlen;
        TEXT *format;
```

FUNCTION

wswrite writes to the workstation defined by dtfw from a buffer with record address rcad. A maximum of **outlen** characters are written. The format with name **format** is used when writing.

RETURNS

wsread returns the workstation completion code.

SEE ALSO

wread

NAME

xlate - translate a buffer of characters

SYNOPSIS

```
TEXT *xlate(table, buf, n)
    TEXT table[], *buf;
    BYTES n;
```

FUNCTION

xlate translates n characters starting at buf, by replacing each character in buf with the byte in table that the character indexes. table should be a 256 byte array containing the replacement characters.

There are two tables defined for easy translation between ASCII and EBCDIC:

_atoe - is a 256 byte array consisting of the corresponding EBCDIC character for each ASCII value.

_etoa - is a 256 byte array consisting of the corresponding ASCII character for each EBCDIC value.

RETURNS

The value returned is the start address of buf.

EXAMPLE

To translate an input buffer buf of length len from EBCDIC to ASCII:

```
xlate(_etoa, buf, len);
```

SEE ALSO

_atoe, _etoa

NAME

_atoe - translation table from ASCII to EBCDIC

SYNOPSIS

```
/*  TRANSLATION TABLE FROM ASCII TO EBCDIC
 *  copyright (c) 1983, 1984, 1985, 1986 by Whitesmiths, Ltd.
 */
#include <std.h>

GLOBAL TEXT _atoe[]
    {
    0x00, 0x01, 0x02, 0x03, 0x37, 0x2d, 0x2e, 0x2f,
    0x16, 0x05, 0x25, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x3c, 0x3d, 0x32, 0x26,
    0x18, 0x19, 0x3f, 0x27, 0x1c, 0x1d, 0x1e, 0x1f,
    0x40, 0x5a, 0x7f, 0x7b, 0x5b, 0x6c, 0x50, 0x7d,
    0x4d, 0x5d, 0x5c, 0x4e, 0x6b, 0x60, 0x4b, 0x61,
    0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
    0xf8, 0xf9, 0x7a, 0x5e, 0x4c, 0x7e, 0x6e, 0x6f,
    0x7c, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
    0xc8, 0xc9, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6,
    0xd7, 0xd8, 0xd9, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6,
    0xe7, 0xe8, 0xe9, 0xad, 0xe0, 0xbd, 0x5f, 0x6d,
    0x79, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
    0x88, 0x89, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
    0x97, 0x98, 0x99, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6,
    0xa7, 0xa8, 0xa9, 0xc0, 0x4f, 0xd0, 0xa1, 0x07,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x15, 0x06, 0x17,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x09, 0x0a, 0x1b,
    0x30, 0x31, 0x1a, 0x33, 0x34, 0x35, 0x36, 0x08,
    0x38, 0x39, 0x3a, 0x3b, 0x04, 0x14, 0x3e, 0xe1,
    0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
    0x49, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
    0x58, 0x59, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75,
    0x76, 0x77, 0x78, 0x80, 0x8a, 0x8b, 0x8c, 0x8d,
    0x8e, 0x8f, 0x90, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e,
    0x9f, 0xa0, 0xaa, 0xab, 0xac, 0x4a, 0xae, 0xaf,
    0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
    0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0x6a, 0xbe, 0xbf,
    0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf, 0xda, 0xdb,
    0xdc, 0xdd, 0xde, 0xdf, 0xea, 0xeb, 0xec, 0xed,
    0xee, 0xef, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff
    } ;
```

FUNCTION

This is the table provided for translation from ASCII to EBCDIC. It can be changed to comply with local conventions. If possible, this table and _etoa should be complementary.

SEE ALSO

_etoa

NAME

_close - close a file

SYNOPSIS

```
COUNT _close(fd)
        FD fd;
```

FUNCTION

_close closes the file associated with the file descriptor fd, making the fd available for future _open calls. If the file was written to or created, _close ensures that the last record is written out and the directory properly closed.

RETURNS

_close returns zero, if successful. Otherwise, it returs a negative number and sets errno to a relevant error code.

EXAMPLE

To copy an arbitary number of files:

```
while (0 <= (fd = getfiles(&ac, &av, STDIN, -1)))
    {
    while (o < (n = _read(fd, buf, BUFSIZ)))
        _write(STDOUT, buf, n);
    _close(fd);
    }
```

SEE ALSO

_open, _remove, _uniqnm, ssp.h

NAME

_etoa - translation table from EBCDIC to ASCII

SYNOPSIS

```
/*  TRANSLATION TABLE FROM EBCDIC TO ASCII
 *  copyright (c) 1983, 1984, 1985, 1986 by Whitesmiths, Ltd.
 */
#include <std.h>

GLOBAL TEXT _etoa[]
    {
    0x00, 0x01, 0x02, 0x03, 0x9c, 0x09, 0x86, 0x7f,
    0x97, 0x8d, 0x8e, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x9d, 0x85, 0x08, 0x87,
    0x18, 0x19, 0x92, 0x8f, 0x1c, 0x1d, 0x1e, 0x1f,
    0x80, 0x81, 0x82, 0x83, 0x84, 0x0a, 0x17, 0x1b,
    0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x05, 0x06, 0x07,
    0x90, 0x91, 0x16, 0x93, 0x94, 0x95, 0x96, 0x04,
    0x98, 0x99, 0x9a, 0x9b, 0x14, 0x15, 0x9e, 0x1a,
    0x20, 0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6,
    0xa7, 0xa8, 0xd5, 0x2e, 0x3c, 0x28, 0x2b, 0x7c,
    0x26, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
    0xb0, 0xb1, 0x21, 0x24, 0x2a, 0x29, 0x3b, 0x5e,
    0x2d, 0x2f, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
    0xb8, 0xb9, 0xe5, 0x2c, 0x25, 0x5f, 0x3e, 0x3f,
    0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf, 0xc0, 0xc1,
    0xc2, 0x60, 0x3a, 0x23, 0x40, 0x27, 0x3d, 0x22,
    0xc3, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9,
    0xca, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
    0x71, 0x72, 0xcb, 0xcc, 0xcd, 0xce, 0xcf, 0xd0,
    0xd1, 0x7e, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
    0x79, 0x7a, 0xd2, 0xd3, 0xd4, 0x5b, 0xd6, 0xd7,
    0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
    0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0x5d, 0xe6, 0xe7,
    0x7b, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
    0x48, 0x49, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed,
    0x7d, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50,
    0x51, 0x52, 0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3,
    0x5c, 0x9f, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
    0x59, 0x5a, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff
    } ;
```

FUNCTION

This is the table provided for translation from EBCDIC to ASCII. It can be changed to comply with local conventions. If possible this table and _atoe should be complementary.

SEE ALSO

_atoe

NAME

_exit - terminate program execution

SYNOPSIS

```
VOID _exit(status)
        BOOL status;
```

FUNCTION

_exit terminates program execution. **_exit** is called with a zero to indicate success, or a non-zero to indicate unsuccessful termination. The status value is currently disregarded by the system interface.

RETURNS

_exit will never return to the caller.

EXAMPLE

To open a file and check for errors:

```
if ((fd = _open("file", O_RDONLY, CPERM, 0)) < 0)
        {
        _write(STDERR, "can't open file\n", 16);
        _exit(FAIL);
        }
```

SEE ALSO

_onexit

NOTES

_exit is called when immediate termination is desired, without calling all functions previously registered with _onexit.

NAME

_getenv - get environment variable

SYNOPSIS

```
TEXT *_getenv(name)
        TEXT *name;
```

FUNCTION

_getenv searches an externally-supplied environment list for a string of the form "NAME=value", where NAME matches the NUL terminated string at name.

Under SSP, _getenv is a dummy.

RETURNS

_getenv always returns NULL.

NAME

_kill - send signal to a process

SYNOPSIS

```
INT _kill(pid, sig)
        INT pid, sig;
```

FUNCTION

_kill is supposed to implement hardware signals.

Under SSP, _kill is a dummy.

RETURNS

_kill always returns zero.

SEE ALSO

_signal

NAME

_lseek - set file read/write pointer

SYNOPSIS

```
LONG _lseek(fd, offset, sense)
        FD fd;
        LONG offset;
        COUNT sense;
```

FUNCTION

_lseek uses the long offset provided to modify the read/write pointer for the file fd, under control of sense. If sense is SEEK_SET (0), the pointer is set to offset, which should be positive. If sense is SEEK_CUR (1), the offset is algebraically added to the current pointer. Otherwise sense is SEEK_END (2) of necessity and the offset is algebraically added to the length of the file in bytes to obtain the new pointer.

_lseek is currently only supported on binary disk files.

RETURNS

_lseek returns the new long offset if successful. Otherwise it returns a negative number and sets errno to a relevant error code.

EXAMPLE

To read a 512-byte block:

```
BOOL _getblock(buf, blkno)
        TEXT *buf;
        BYTES blkno;
        {
        _lseek(STDIN, (LONG)blkno << 9, 0);
        return (_read(STDIN, buf, 512) != 512);
        }
```

SEE ALSO

ssp.h

NAME

_main - setup for main call

SYNOPSIS

```
BOOL _main(s)
        TEXT *s
```

FUNCTION

_main is the function called whenever a C program is started. It opens files for STDIN, STDOUT and STDERR, and also parses the command line at **s** into argument strings. Then it calls the user's main function.

The command line is interpreted as a series of strings separated by whitespace. All strings are taken by _main as argument strings to be passed to the user's main function. The command name, av[0], is taken from _pname.

EXAMPLE

To avoid the loading of _main and all the file I/O code it calls on, one can provide a substitute _main instead:

```
BOOL _main()
        {
        <program body>
        }
```

Note:

Arguments to a program must be placed in the local data area before the program is called. The following procedure member will take six arguments, and then load and run the load member ECHO:

```
// LOCAL DATA-'?1? ?2? ?3? ?4? ?5? ?6?',BLANK-*ALL
// LOAD ECHO
// RUN
```

If the procedure member was called ECHO the command line:

```
ECHO HELLO,WORLD
```

would be turned into the following **av** list:

```
av[0] echo
av[1] hello
av[2] world
```

Note that all arguments will be translated to lowercase.

RETURNS

_main returns the boolean value obtained from the main call, which is then passed to _exit.

SEE ALSO

• _exit, _pname, _terminate

NOTES

Under SSP, STDIN is opened to read the NULL: device. STDOUT and STDERR are opened to write to the SYSLOG: device. Currently _main also creates a parameter list containing _pname as av[0] only.

NAME

_onexit - call function on program exit

SYNOPSIS

```
FNPTR _onexit(pfn)
        FNPTR (*pfn)();
```

FUNCTION

_onexit registers the function pointed at by **pfn**, to be called on program exit. The function at **pfn** returns the pointer returned by the **_onexit** call, so that any previously registered functions can also be called.

RETURNS

_onexit returns a pointer to another function; it is guaranteed not to be NULL.

EXAMPLE

To register the function pointed to by **thisguy**:

```
IMPORT FNPTR _onexit();
IMPORT FNPTR nextguy, thisguy, _onexit();
          ...
if (!nextguy && !(nextguy = _onexit(&thisguy)))
        remark ("can't register with onexit", NULL);
```

SEE ALSO

_exit, _terminate

NAME

_onintr - capture interrupts

SYNOPSIS

```
VOID _onintr(pfn)
        VOID (*pfn)();
```

FUNCTION

_onintr ensures that the function at **pfn** is called on a keyboard interrupt, usually caused by a DEL key or ctl-BREAK typed during terminal input or output. Any earlier call to **_onintr** is overridden.

The function at pfn is called with one integer argument, whose value is always zero, and must not return. If it does, an immediate error exit is taken.

If (**pfn** == NULL) then these interrupts are disabled (turned off).

Under SSP, **_onintr** is currently a dummy.

RETURNS

Nothing.

EXAMPLE

A common use of **_onintr** is to ensure a graceful exit on early termination:

```
FNPTR rmtemp()
        {
        IMPORT FNPTR fnext;

        _remove(_uniqnm());
        return(fnext);
        }
                ...
        fnext = _onexit(&rmtemp);
        _onintr(&exit);
```

Still another use is to provide a way of terminating long printouts, as in an interactive editor:

```
        while (!enter(docmd, NULL))
                _putstr(STDOUT, "?\n", NULL);
                        ...
VOID docmd()
        {
        _onintr(&leave);
                        ...
```

SEE ALSO

_onexit

NAME

_open - open a file

SYNOPSIS

```
FD _open(fname, oflag, mode, xparm)
        TEXT *fname;
        BITS oflag, mode;
        TEXT *xparm;
```

FUNCTION

_open opens a file **fname**, sets the file attributes from **oflag**, and assigns a file descriptor to it. Acceptable values for oflag are:

O_RDONLY - the file is opened for reading.

O_WRONLY - the file is opened for writing.

O_RDWR - the file is opened for updating.

O_APPEND - seek to the end of the file prior to each write. Each write will be at the current end of the file.

O_CREAT - create the file if it does not already exist. If the file must be created, the size is set to zero, and the permissions are set to the low 12 bits of **mode**. For maximum portability to other systems, mode should be set to the manifest constant CPERM. **mode** is meaningless under SSP, and O_TRUNC must also be set if a new file is to be created, due to the mobility to test for the existence of a file.

O_TRUNC - truncate the file if it exists. The length is set to zero, but the other attributes of the file are unchanged. Under SSP, the file will be created if it does not exist.

O_BIN - open the file for binary I/O. If O_BIN is set, data is transmitted unaltered. If O_BIN is not set, the file is treated as a text file and **write** terminates each output record on a newline, deletes the newline, and pads the rest of the record with spaces. O_BIN also causes **read** to return the number of characters asked for and the remainder of the input record, with trailing blanks deleted and a newline added. Note that this flag is a non-portable extension.

O_EXCL - when O_EXCL and O_CREAT are set, **_open** will fail if the file exists. Under SSP, **_open** will fail anyhow, since it is currently not possible to check for the existence of a file.

O_XTYPE - if O_XTYPE is set, then **_open** accepts an extra parameter, **xparm**, which defines operating system specific information. **xparm** is currently only used to specify the record length. Acceptable values for **xparm** are 0 for the default record length or a pointer to a string which contains **"recl="** and a record length value following the equal sign.

Only one of O_RDONLY, O_WRONLY, and O_RDWR may be used; if none of them is used, O_RDONLY is assumed. The file pointer will be set to the beginning

of the file (but see O_APPEND above).

Note: Only NFILES (16) files may currently be open simultaneously.

Filenames take the general form: [driver:]name or name, where **driver** is the driver designator and **name** is the eight-character filename. If **driver:** is omitted, it is taken as the disk driver ("DSK"). A missing **name** is taken as all blanks.

RETURNS

_open returns a file descriptor (the lowest numbered one available) for the opened file. Otherwise it returns a negative number and sets errno to a relevant error code.

EXAMPLE

To open a file and check for errors:

```
if ((fd = _open ("xeq", O_WRONLY, CPERM)) < 0)
        _write(STDERR, "can't open xeq\n", 15);
```

SEE ALSO

_close, _remove, _rename, _uniqnm

NAME

_pname - program name

SYNOPSIS

```
TEXT *_pname;
```

FUNCTION

_pname is the (NUL terminated) name by which a given program was invoked, at least as anticipated at compile time. If the user program provides no definition for _pname, a library routine supplies the name "error", since it is used primarily for labelling diagnostic printouts.

Under SSP and other operating systems where the program name as specified on the command line is unavailable, the contents of _pname (pointed to by av[0]) defaults to a constant (the string "error" in this implementation).

EXAMPLE

To change _pname from "error" to "application name":

```
TEXT *_pname = {"application name"};
```

SEE ALSO

_main

NAME

_rawmode - query or set tty file to a new state

SYNOPSIS

```
COUNT _rawmode(fd, new)
        FD fd;
        COUNT new;
```

FUNCTION

Under SSP, _rawmode is a dummy.

RETURNS

_rawmode always returns R_QUERY.

SEE ALSO

_lseek

NAME

_read - read characters from a file

SYNOPSIS

```
ARGINT _read(fd, buf, size)
        FD fd;
        TEXT *buf;
        ARGINT size;
```

FUNCTION

_read reads up to size characters from the file specified by fd into the buffer starting at buf. If the file was created or opened as a text file (i.e., if O_BIN was not set in the flag), _read will skip trailing blanks in a record, and add a newline at the end, effectivly translating SSP records into C lines. A call to _read on a text file returns the number of characters asked for, plus the remainder of the current input record, plus one more (the newline character).

RETURNS

If an error occurs, _read returns a negative number and sets errno to a relevant error code. If end of file is encountered, _read returns zero. Otherwise the value returned is between 1 and size if O_BIN is defined, or between 1 and the record length plus 1 if O_BIN is undefined.

EXAMPLE

To copy a file:

```
while (0 < (n = _read(STDIN, buf, BUFSIZE)))
        _write(STDOUT, buf, n);
```

SEE ALSO

_open, _write

NAME

_remove - remove a file

SYNOPSIS

```
COUNT _remove(fname)
        TEXT *fname;
```

FUNCTION

_remove deletes the file fname from the filesystem.

RETURNS

_remove returns zero if successful. Otherwise it returns a negative number and sets errno to a relevant error code.

Under SSP, _remove is not supported by any drivers.

EXAMPLE

To remove the file temp.c:

```
if (_remove("temp.c") < 0)
        _write(STDERR, "can't remove temp file\n", 23);
```

SEE ALSO

_rename

NAME

_rename - change old file to new file

SYNOPSIS

```
COUNT _rename(old, new)
      TEXT *old, *new;
```

FUNCTION

_rename changes the name of the file **old** to the name **new.**

Under SSP, **_rename** is not supported.

RETURNS

_rename always returns NFAIL and sets **errno** to ENOTSUP.

SEE ALSO

_remove

NAME

_sbreak - set system break

SYNOPSIS

```
VOID *_sbreak(size)
        BYTES size;
```

FUNCTION

_sbreak moves the system break, to make available at least size contiguous bytes of new memory, on a storage boundary adequate for representing any type of data. There is no guarantee that successive calls to _sbreak will deliver contiguous areas of memory.

RETURNS

_sbreak returns a pointer to the start of the new memory if successful. Otherwise the value returned is NULL, and _sbreak sets errno to a relevant error code.

EXAMPLE

To buy space for an array of symbols:

```
if (!(p = _sbreak(nsyms * sizeof (symbol))))
        remark("not enough memory!", NULL);
```

NAME
    _signal - capture signals

SYNOPSIS
```
    VOID (*_signal(sig, func))()
            INT sig;
            VOID (*func)();
```

FUNCTION
    _signal is supposed to implement hardware signals.

    Under SSP, _signal is a dummy.

RETURNS
    _signal always returns zero.

SEE ALSO
    _kill

NAME

_svc - make a supervisor call to SSP

SYNOPSIS

```
VOID _svc(ilc, xrs, rbyte, qbyte, ...)
        UTINY ilc;
        UTINY xrs;
        UTINY rbyte;
        UTINY qbyte;
```

FUNCTION

_svc is a C-callable function that permits arbitrary calls to be made on the System Support Program (SSP). The argument ilc specifies the number of inline parameter list byte arguments that will passed after the first four. The argument xrs is used to specify which index registers to load with the last additional arguments. Select index register one with the value of 1, index register two with the value 2, both with the value 3, and none with 0. rbyte is the SVC R-byte and qbyte is the SVC Q-byte (see the S/36 Functions Reference Manual for further information). At the end of those first four fixed arguments follows zero to three inline bytes and zero to two index register values as specified by ilc and xrs.

Register load is supported because several functions in the SSP require an address to a control block in index register one or two.

RETURNS

Nothing.

EXAMPLE

To make a memory dump with the SNAP procedure in SSP:

```
#include <wslxa.h>
#include "ssp.h"

struct
    {
    UTINY options;
    TEXT *low;
    TEXT *high;
    TEXT dumpid[4];
    }
snpp;

snpp.options = 0;
snpp.low = (TEXT *) 0;
snpp.high = (TEXT *) 1024;
strncpy(snpp.dumpid, "TEST", 4);
_svc(1, 2, 0x4, 0x1, 0x11, &snpp);
```

SEE ALSO

ssp.h

NAME

_system - execute a command

SYNOPSIS

```
COUNT _system(cmd)
        TEXT *cmd;
```

FUNCTION

_system is supposed to invoke a command interpreter to parse and execute the string pointed at by **cmd**. A value of NULL for **cmd** is a query to the command interpreter and should return non-zero if other non-NULL values of **cmd** are meaningful.

Under SSP, **_system** is a dummy.

RETURNS

_system always returns -1.

NAME

_terminate - terminate program execution

SYNOPSIS

```
VOID _terminate(status)
        BOOL status;
```

FUNCTION

_terminate calls all functions registered with **_onexit**, and then terminates program execution by calling **_exit**. **_terminate** is called with a zero to indicate success, or a non-zero to indicate unsuccessful termination.

RETURNS

_terminate will never return to the caller.

SEE ALSO

_exit, _onexit

NAME
_time - get system time

SYNOPSIS
ULONG _time()

FUNCTION
_time is supposed to get the system time, which is the number of seconds since the 1 January 1970 epoch.

Under SSP, _time is currently a dummy.

RETURNS
_time returns 0 as an unsigned long integer.

EXAMPLE
To print the date:

```
ULONG lt;
        ...
lt = _time();
printf("%s", ctime(&lt));
```

NAME

_uniqnm - create a unique file name

SYNOPSIS

```
TEXT *_uniqnm()
```

FUNCTION

_uniqnm returns a pointer to the start of a NUL-terminated name that is likely not to conflict with normal user filenames. The name may be modified by a letter suffix, so that a family of process-unique files may be dealt with as a group. The name may be used as the first argument to a subsequent _open call, so long as any such files created are removed before program termination. It is considered bad style to leave scratch files lying about.

RETURNS

_uniqnm returns the same pointer on every call, which is currently the string "CTEMPC". The pointer will never be NULL.

EXAMPLE

To create a temporary file:

```
if ((fd = _open(_uniqnm(), O_WRONLY|O_CREAT|O_TRUNC, CPERM)) < 0)
        _write(STDERR, "can't create sort temp\n", 23);
```

SEE ALSO

_close, _open, _remove

NAME

_write - write characters to a file

SYNOPSIS

```
ARGINT _write(fd, buf, size)
        FD fd;
        TEXT *buf;
        ARGINT size;
```

FUNCTION

_write writes **size** characters starting at **buf** to the file specified by **fd.** If the file was created or opened with O_BIN undefined, a newline terminates the current output record, the newline is deleted, and the rest of the record is padded with spaces.

If the O_APPEND flag was set when the file was opened, the file pointer will be set to end of file prior to each write.

RETURNS

If an error occurs, **_write** returns a negative number and sets **errno** to a relevant error code. Otherwise the value returned should be **size.**

EXAMPLE

To copy a file:

```
while (0 < (n = _read(STDIN, buf, size)))
        _write(STDOUT, buf, n);
```

SEE ALSO

_open, _read