JAM C Programmer's Guide

Contents

1 Introduction

The Programmer's Guide explains how to write code that works with the JAM run-time environment. First, we offer a general explanation of how to build your routines into the run-time system. Then, the library itself is classified into categories, and an alphabetical listing of all the functions with full explanations appears. Finally, there are explanations of internal processing and tips for writing better applications; this material is organized topically, and is better browsed or used for reference than read straight through.

1.1 The JAM Run-time Environment

A salient fact of programming with JAM is that the application program has no main control loop. Control flow is handled by JAM's main routine, which uses control strings to decide how to respond to what you type; your own code is made up chiefly of screen entry and exit functions, invoked functions, and attached functions. Invoked functions are placed in JAM control strings; they are called immediately upon receipt of a user's menu pick or function key, and may prompt for parameters with an argument window. Attached functions, on the other hand, are associated with data entry fields, and are called with a predefined set of arguments when the cursor enters or leaves a field. The next several sections describe each type of function and provide examples.

It is, of course, possible to gain full control of screen and keyboard processing, by using the library routines. Section 2 categorizes those routines and describes their typical use. It is further possible to alter or abolish the processing of control links by writing your own main routine.

1.2 Functions Invoked by Control Links

A function whose name appears in a JAM control string, preceded by a caret ^, will be called when a user selects the function key or menu pick associated with that control string. This means that JAM must be linked with the function. See Section 1.5 for a detailed explanation of how to link functions with JAM. JAM supplies a number of built-in invoked functions, which are documented in Section 4 of this chapter.

Invoked functions are passed a single argument, a pointer to a character string. In the simplest case the string is just the contents of the control field (the function name plus any following text), with the leading caret stripped off. However, if the control field contains one or more argument window specifications (a percent character % followed by the name of a window), the function receives their contents: JAM opens each window for data entry, then substitutes the concatenated contents of its fields for the percent specification, before calling the function. See the Author's Guide for details on how to create argument windows.

1.2.1 Return Values and Target Lists

The integer return value from an invoked function is ignored, unless

1. it is the value of a logical key with an associated control string on the current screen (EXIT, XMIT, PF1, etc.), or
2. the control string contains a target list.

If there is no target list and the return value is a logical key with an associated control string, JAM will execute the control string. Invoked functions can be chained in this fashion; programmers are responsible for avoiding looping chains.

A target list in a control string provides for more flexible and better-documented chaining of invoked functions. Syntactically, a target list appears between the caret and the function name, and must be enclosed in parentheses. It consists of one or more pairs, of the form

```
     return-value = control-string
```

Here return-value is an integer code returned from the invoked function, expressed as a decimal or hexadecimal integer, a quoted ASCII character, or a logical key mnemonic (refer to smkeys.h for a list). Control-string is any JAM control string, which will be executed whenever the function returns return-value. If the target list contains more than one such pair, as is likely, separate them with semicolons.

In the control string below, a return value of -1 from the function process causes the function cleanup to be invoked, while a return value of 0 causes a window named step2 to be displayed:

```
     ^(-1=^cleanup; 0 = &(8,25)step2)process step1
```

Target lists may nest; that is, caret control strings in a target list may themselves contain target lists.

1.2.2 Example

The example below shows a function invoked when a user hits TRANSMIT or EXIT on its screen; it simply saves the contents of the screen in a flat data file. The two control strings that call it would look like this:

```
     XMIT =>    ^save
     EXIT =>    ^(0=^jm_exit)save close
```

And here is the function itself:

```
#include "smdefs.h"
#Include "smidty.h"
#include "myrec.h"

/* The following is an invoked function bound to the
 * TRANSMIT key. It saves the contents of the screen
 * in the data structure 'myrec' and writes it to a file,
 * which is opened if necessary. The argument determines
 * whether the file should be closed; it is in the control
 * string.
 * The data structure myrec is assumed to be defined in
 * "myrec.h", generated from the screen by the f2struct
 * utility.
 */

int save (closeflag)
char *closeflag;
{
     struct myrec myrec;
     int count;
     static FILE *savefile;

     if (strcmp (closeflag, "close") == 0)
     {
          if (savefile != 0)
               fclose (savefile);
          return 0;
     }

     sm_wrtstruct ((char *)&myrec, &count, 0);

     if (savefile == 0)
     {
          savefile = fopen ("savefile", F_B_WRONLY);
          if (savefile == 0)
               return -1;
     }
     if (fwrite ((char *)&myrec, 1, count, savefile) != count)
          return -1;

     return 0;
}

/* Code to install the invoked function above */
struct fnc_data invlist[] = {
     { "save", save, 0, 0, 0, 0 }
};
int howmanyinv = sizeof(invlist) / sizeof(struct fnc_data);

sm_install (CARET_FUNC, invlist, &howmanyinv);
```

1.3 Functions Attached to Fields

Field entry and field exit functions are both called attached functions, because they are associated with a specific data-entry field. You attach a function to a field by placing its name in the field entry or validation function slot of the screen editor's pop-up windows. (You may place the same function in both slots, and use the fourth parameter to determine the context of the call; see below.)

To attach a JPL procedure to a field, you can type your code directly into the screen editor's JPL procedure window. If the JPL procedure is long or useful in many places, you can also put it in a file, and enter jpl filename in the attached function window. You may attach JPL code to both hooks, in which case the attached function is executed before the embedded procedure.

1.3.1 Invocation

A field exit function, also commonly called a validation function, is called:

1. As part of field validation, when you exit the field by filling it or by hitting the TAB or RETURN key; BACKTAB and arrow keys do not normally cause validation. These circumstances are subject to modification by sm_ok_options, q.v.
2. As part of screen validation (sm_s_val), when the TRANSMIT key is hit.
3. When application code calls sm_fval directly.

A field entry function is called from sm_openkeybd whenever the cursor enters its field.

1.3.2 Arguments

All attached functions receive four arguments:

1. the field number
2. a pointer to the buffer containing a copy of the field's contents
3. the data's occurrence number
4. an integer containing the VALIDED and MDT bits associated with the item or field, plus more flag bits indicating why the function was called; refer to the following table.

Bit name	Meaning
VALIDED	If set, indicates that the field has passed all its validations, and has not been modified since.
MDT	If set, indicates that the field has been modified, either by keyboard input or a library function such as sm_putfield,

	since the current screen was first displayed. It is never cleared by JAM, but you may clear it using sm_bitop.
K_ENTEXIT	If set, indicates that the function has been called upon field entry. This bit enables a single function to be used for both field entry and exit.
K_KEYS	This is a mask for the remaining values in this table. You should and the function's fourth parameter with this value, then test for equality to one of the five below, thus: `if ((parm4 & K_KEYS) == K_SVAL)`
K_NORMAL	A normal data key caused the cursor to enter or exit the field in question.
K_BACKTAB	The BACKTAB key caused the cursor to enter or exit the field in question.
K_ARROW	An arrow key caused the cursor to enter or exit the field in question.
K_SVAL	The field is being validated as part of screen validation (TRANSMIT key or sm_s_val).
K_USER	The application has invoked the function directly, as through sm_fval.
K_OTHER	Another key, such as HOME, caused the cursor to enter or exit the field in question.

A field exit function is called whether or not the field was previously validated; it may test the VALIDED and MDT flags to avoid redundant processing. It is called only after the field's contents pass all other validations.

1.3.3 Return Value

If no error occurs, the function should return 0. At this point, the screen manager's validation routine will set the VALIDED bit, if it was not already set. Any nonzero value returned by the function is interpreted as an error. If the returned value is 1, the cursor is not repositioned; if it is any other nonzero value, the cursor is repositioned to the field undergoing validation. In either case, the VALIDED bit is unchanged.

1.3.4 Example

Here is code for an attached function that does nothing much:

```
#include "smdefs.h"

int apfunc1 (field_number, field_data, occurrence, val_mdt)
int field_number, occurrence, val_mdt;
char *field_data;
{
     int error;
     extern int lookup();
     char errbuf[128];

     /* If field is unchanged since last validation, skip */
     if (val_mdt & VALIDED)
          return (0);

/* Check field for validity (externally defined) */
     error = lookup (field_data);

     if (error)
     {
          /* Notify user of error condition. */
          sm_gofield (1);
          sprintf (errbuf, "Can't find %s; please re-enter all data.",
               field_data);
          sm_quiet_err (errbuf);
          return (1);
                    /* leave cursor in field 1 */
     }
     else return (0);
}
```

While an attached function is free to display windows, it must not display a form. To do so would wipe out the form or window to which the function was attached. For the same reason, attached functions must take care to close all the windows they may open before returning to sm_openkeybd.

1.4 Screen Entry and Exit Functions

Screens, like fields, may have entry and exit functions defined in the screen editor. The screen entry function is called after the screen has been displayed, but before control returns from sm_r_window, sm_d_form, or whatever you use to display the screen. There are several steps in screen initialization, performed in the following order:

1. The new screen is displayed, as a form or window.
2. The screen-entry function is called.
3. The jam_first control string is executed. (This is an obsolescent feature.)
4. Screen fields are updated with values from the local data block (LDB merge).
5. The jam_auto control string is executed.

During screen initialization, entering data into fields with sm_putfield or other library routines will not cause the fields' MDT bits to be set; this applies to the LDB merge as well. If you want to set MDT bits, call sm_bitop.

Screen exit functions are called after named items have been stored in the LDB, but before the screen is removed from the display and its data structures destroyed. They are called from sm_close_window, and also from sm_r_form and the other form-display functions, which cause all open windows to be destroyed automatically. This implies that the screen may not always be visible when your exit function is called.

1.4.1 Arguments

Screen entry and exit functions receive two parameters:

1. The name of the screen, if available. It will not be available if the screen is memory-resident and displayed using sm_d_window or a variant.
2. A flag containing the following bits, defined in smdefs.h :

K_ENTEXIT	Set if the function was called during screen entry, clear during exit.
K_NORMAL	Set if the function was called from sm_close_window, clear otherwise.

A single function used for both entry and exit can use this bit to distinguish its context. One advantage of coding screen entry and exit processing in the same function is that persistent variables, such as pointers to dynamically allocated buffers, can be made static rather than global.

Any value returned from a screen entry or exit function is ignored.

1.4.2 Example

Here is an example of a screen entry function. This one loads up an item selection list from a disk file whose name is stored in the screen.

```
#include "smdefs.h"

/* Here is a screen entry function for a generic item
 * selection screen. It turns the screen name into the
 * name of a text file containing a list of items, one
 * per line, and loads them into the array named "items".
 * This technique could be easily adapted to query a database
 * instead.
 */

void gen_entry (name, context)
char *name;
int context;
{
     FILE *inf;
     char line[256];
     int k;

     sprintf (line, "%s.dat", name ? name : "default");
     if ((inf = fopen (line, "r")) == 0)
          return;
     for (k = 1; fgets (line, sizeof(line), inf); ++k)
          sm_i_putfield ("items", k, line);
     fclose (inf);
}

/* Here is code to install the above function in the screen
 * entry function list. */
static struct fnc_data sentry[] = {
     { "gen_entry", gen_entry, 0, 0, 0, 0 }
};
int count;

count = sizeof(sentry) / sizeof(struct fnc_data);
sm_install (FENTRY_FUNC, sentry, &count);
```

1.5 The Function Lists

JAM stores function names as text strings in screen files. It needs to associate these names with the functions' addresses in order to call them. You must furnish this association by building lists of data structures called function lists. Attached, invoked, and screen entry functions must all appear; there is one list for each type of function. For example, if an application's screens contained two functions named apfunc1 and apfunc2 that were attached to a field, and another named invoca in a caret control string, its startup code would need to include the following:

```
#include "smdefs.h"
#define C_FUNCTION 0

extern int apfunc1(), apfunc2();

struct fnc_data funlist[] =
{
     { "apfunc1", apfunc1, C_FUNCTION, 0, 0, 0 },
     { "apfunc2", apfunc2, C_FUNCTION, 0, 0, 0 },
};
/* The following definition saves a lot of grief */
int howmuchfun = sizeof(funlist) / sizeof(struct fnc_data);

extern int invoca();

struct fnc_data invlist[] =
{
     { "invoca", invoca, C_FUNCTION, 0, 0, 0 }
};
int howmanyinv = sizeof(invlist) / sizeof(struct fnc_data);

...
sm_install (ATTCH_FUNC, funlist, &howmuchfun);
sm_install (CARET_FUNC, invlist, &howmanyinv);
...
```

In the structure definitions, the quoted strings are names as entered on the screen, and the non-quoted entries are addresses of functions. Note that the two need not always be the same; in particular, the same function can be entered in the list under different names, or aliases. Possible uses for this technique include mapping functions yet to be written to a stub routine, and using the same function to perform slightly different tasks (with the name as an implied parameter).

Refer to the library page on sm_install, and to preceding sections, for fuller details about each type of function.

1.6 The JAM Main Programs

As a starting point for your own applications, JYACC provides source code for a main routine, in a file named jmain.c. This routine performs various necessary initializations before calling JAM's main control loop, which resides in a library. You can modify the main routine to change the default settings and, more importantly, to install your application code with calls to sm_install, as in the previous section. Extensive documentation in the source code will show you where and how to make your modifications.

You will find similar source files, fmain.c and jxmain.c, that are main routines for the authoring programs, xform and jxform. Under Release 4.0, you may link your application code into the authoring environment as well as the run-time system. This enables you to prototype and revise your screens with even greater convenience than before.

Refer to the Installation Notes for your system for the locations of these files. For additional hints on putting your application together, take a look at the sample programs (next section).

1.7 Sample Programs

JYACC supplies a number of sample programs and screens to demonstrate the use of JAM routines and the procedures you must use to compile and link application programs. They include both programs that supply their own main routine, and programs that use the one provided with JAM. They can generally be found in a subdirectory, named samples, of the directory where JAM has been installed. Refer to the Installation Notes for your specific system for a list of the files making up the sample programs.

2 Overview of Library Functions

After screens have been created with the JAM authoring utility, an application program can access them using routines from the library. A typical sequence follows.

> Note that the JAM environment performs something very similar to this sequence, and it is not generally necessary for application programs to do it; application code is simply attached to fields and function keys. Only in circumstances where very tight control over data entry is required should it be necessary for an application program to do all this work.

1. sm_initcrt is called to initialize the terminal.
2. sm_r_form is called to bring up a form on the screen.
3. The application program may call sm_putfield, or a variant, to initialize fields. This can be done as often as desired, and at any time.
4. sm_openkeybd is called, and you may key data into unprotected fields.
5. One or more data access routines are called to return the fields' current contents to the application program.
6. While the data contain errors or inconsistencies:

    a. sm_gofield, or a variant, is called to reposition the cursor to the field containing the error.
    b. sm_err_reset is called to display an error message.
    c. sm_openkeybd is called to accept fresh data.

7. If special conditions require additional input:

    a. sm_r_window, or a variant, is called to bring up a window with additional fields. The entire sequence may be repeated here recursively.
    b. sm_close_window is called to close the current window and restore the immediately previous display.

8. sm_resetcrt is called to reset the terminal prior to exiting.

2.1 Variants

Many library routines deal with fields. Most of these have several variants that accept different sorts of field designations, enabling the application programmer to choose the most convenient. The variants are, by convention, distinguished by prefixes to the function name:

Prefix	Field designation
sm_	Field number only
sm_o_	Field number and occurrence number
sm_n_	Field name only
sm_i_	Field name and occurrence number
sm_e_	Field name and onscreen element number

In the library section of the manual, only the first variant is documented. The behavior of the others is identical, save for the handling of errant field specifications. Also, the n_ and i_ variants will operate on fields that do not appear in the current screen but are in the local data block.

A similar convention exists for library routines that display screens, depending on whether the screens file is on disk, in memory, or included in a screen library. The prefixes are as in the table below. All these functions are documented separately.

Prefix	Screen file storage
sm_r_	Single disk-resident file

sm_d_	Memory-resident file
sm_l_	Member of screen library

Most JAM library routines fall into one of the following categories:

- Initialization
- Form display
- Data entry (from the application program)
- Keyboard entry
- Cursor control
- Data access (by the application program)
- Mass storage and retrieval
- Message display
- Altering the operation of other library routines
- Scrolling and shifting
- Data validation

The following sections summmarize the routines within each category. Modules calling JAM library routines should include smdefs.h . Modules testing for non-ASCII values returned by sm_getkey, sm_openkeybd or sm_menu_proc should also copy or include smkeys.h .

2.2 Initialization

The following routines are called for initialization. Note that, by default, most of these are called to set up the JAM run-time environment, and the application program need not call them itself.

sm_initcrt	Initializes the terminal for JAM, and saves the path name for disk resident forms and windows.
sm_keyinit	Initialize memory-resident key translation file.
sm_msgread	Loads message files.
sm_vinit	Initialize memory-resident video file.
sm_smsetup	Initialize memory-resident configuration variable file. sm_unsetup restores the default configuration.
sm_formlist	Add to the list of memory-resident forms.
sm_rmformlist	Destroys the memory-resident form list.
sm_install	Installs attached functions, screen entry/exit functions, and other user routines to be called from library functions.
sm_resetcrt	Restores normal terminal characteristics before exiting.
sm_leave	Prepares the display for a temporary escape from JAM to the operating system.
sm_return	Prepares the display to resume processing after an sm_leave.
sm_cancel	Resets the terminal and exits. Normally bound to a keyboard interrupt handler.
sm_fextension	Changes the default file extension for screen files.
sm_inictrl	Changes the default control string bindings for function keys.

2.3 Form Display

The following routines are called to display screens.

sm_r_form	Displays a form. Any previously displayed form is cleared. Similarly, sm_d_form, sm_l_form.
sm_r_window	Displays a window at a specified line and column on the screen. The previous contents of the window area are saved. Similarly, sm_d_window, sm_l_window.
sm_r_at_cur	Displays a window at the current cursor position. The previous contents of the window area are saved. Similarly, sm_d_at_cur, sm_l_at_cur.
sm_close_window	Closes the current window and restores the immediately previous display.
sm_wselect	Brings a "buried" window to the active position. sm_wdeselect puts it back.
sm_l_open	Opens a form library.
sm_l_close	Closes a form library.
sm_hlp_by_name	Displays the help screen attached to the current field or screen, gets user input, and restores the previous display.
sm_rescreen	Refreshes the screen display.

2.4 Data Entry

The following routines enable an application program to enter data into fields in a screen or the local data block, or to change its display attributes.

sm_putfield	Copies the data from a string into a specified field or occurrence. If the string is too long for the field, it is truncated.
sm_amt_format	Copies a string into the specified amount field or occurrence, after applying the currency format.
sm_itofield	Converts an integer to a string, and copies the string into the specified field or occurrence.
sm_dtofield	Converts a double floating point value to a string, applies a field currency edit if it exists, and copies the string into the specified field or occurrence.
sm_ltofield	Converts a long integer to a string and copies the string into the specified field or occurrence.
sm_calc	Evaluates a mathematical expression, possibly involving field values, and places the result in a field.
sm_ldb_init	Initialize the local data block from the data dictionary.
sm_lclear	Clear all occurrences with a given scope in the LDB.
sm_lreset	Reinitialize all occurrences in the LDB with a given scope.
sm_chg_attr	Changes the display attributes of a field.
sm_achg	Changes the display attributes of a scrolling data item.
sm_cl_unprot	Clears onscreen and offscreen data from unprotected fields on the current screen. Date and time fields that take system values are reset.
sm_cl_everyfield	Clears all field data, onscreen and off, regardless of field protection.
sm_1clear_array	Clears all data from a scrolling array.
sm_clear_array	Clears all data from a scrolling array and all parallel scrolling arrays.
sm_ioccur	Inserts a blank occurrence into an array or scroll.
sm_doccur	Deletes an occurrence from an array or scroll.

2.5 Keyboard Entry

The following routines accept and process data entered from the keyboard.

sm_getkey	Returns the interpreted value of the key hit. In the case of a displayable key, this is its standard ASCII value; in the case of a function key with special meaning to JAM, this is a value defined in the file smkeys.h . This function is used by all other keyboard input functions.
sm_ungetkey	Pushes a key back on the input stream, to be retrieved by sm_getkey.
sm_openkeybd	Accepts and interprets user keyboard entry. Displayable data is entered into fields on the screen, subject to any restrictions or edits that were defined for those fields (see the JAM Author's Guide). Function keys control the position of the cursor and aid in editing data.
sm_menu_proc	Returns the initial character of the user's selection from a menu whose entries start with distinct ASCII characters.
sm_choice	Returns the text of the chosen entry on a menu screen.
sm_query_msg	Displays a question on the status line, and returns a yes or no answer.
sm_keyhit	Tests whether a key has been hit during a specified interval.
sm_isabort	Returns control to the application through nested keyboard entry routines.

2.6 Cursor Control

The following routines affect the positioning and visibility of the cursor.

sm_getcurno	Returns the number of the field within which the cursor is currently positioned.
sm_home	Places the cursor in the first unprotected field.
sm_last	Positions the cursor to the start of the last unprotected field of the current form.
sm_tab	Moves the cursor to the next unprotected field, or to the unprotected field specified by a next field edit.
sm_nl	Moves the cursor to the next unprotected field following the current line on the screen.
sm_backtab	Moves the cursor to the start of the current field, or if it is already there, to the start of the previous unprotected field.
sm_gofield	Positions the cursor to the start of the specified field or occurrence.
sm_off_gofield	Positions the cursor to a given offset from the start of the specified field.
sm_disp_off	Returns the displacement of the cursor from the starting column of the current field.
sm_c_off	Turns the cursor off.
sm_c_on	Turns the cursor on.
sm_c_vis	Turns the cursor position display at the end of the status line on or off.

2.7 Data Access

The following routines give an application program access to data entered on a screen or in the LDB.

sm_edit_ptr	Returns the text of a special field edit.
sm_getfield	Returns the contents of a field or occurrence in a buffer supplied by the user.
sm_fptr	Returns the contents of a field in an internal buffer.
sm_pkptr	Returns the contents of a field in an internal buffer, with as many blanks as possible removed.
sm_strip_amt_ptr	Returns the contents of a field, stripped of any extraneous characters supplied by currency formatting.

sm_num_occurs	Returns the largest number of items entered so far into a scrollable field or array.
sm_intval	Returns the integer value of the data found in the specified field or occurrence.
sm_dblval	Returns the double floating point value of the data found in the specified field or occurrence.
sm_lngval	Returns the long integer value of the data found in the specified field or occurrence.
sm_is_yes	Returns 1 if the first character of a yes/no field looks like yes, and 0 otherwise.
sm_dlength	Returns the length of the data currently in the specified field or occurrence.
sm_allget	Merge LDB data items onto the screen.
sm_lstore	Store screen data into the LDB.

2.8 Mass Storage and Retrieval

The following functions move data to or from groups of fields in the screen or LDB.

sm_save_data	Writes the contents of the current form's fields to a buffer for subsequent retrieval.
sm_restore_data	Restores all fields to the current form from a buffer created by sm_save_data.
sm_sv_data	Writes the contents of some of the current form's fields to a buffer for subsequent retrieval.
sm_rs_data	Restores part of the current form from a buffer created by sm_sv_data.
sm_wrtstruct	Copies the current form's fields to a program data structure generated from the screen.
sm_rdstruct	Reads into the current form's fields from such a data structure.
sm_wrt_part	Writes the contents of some of the current form's fields to such a data structure.
sm_rd_part	Reads some of the current form's fields from such a data structure.
sm_rrecord	Copies from a data structure defined by a data dictionary record to the screen or LDB.
sm_wrecord	Copies from the screen to such a data structure.

2.9 Message Display

The following routines display a message on the status line of the screen (typically the bottom line).

sm_d_msg_line	Displays a message with a user-supplied display attribute.
sm_msg	Displays a message at a given column on the status line.
sm_err_reset	Displays a message using a standard error message attribute. Processing is halted until the user hits the space bar, at which time the status line is reset. Similarly, sm_emsg.
sm_quiet_err	Displays the word ERROR: followed by the user-supplied message. Processing is halted until the user hits the space bar, at which time the status line is reset. Similarly, sm_qui_msg.
sm_mwindow	Displays your message in a pop-up window, at a location you specify.
sm_query_msg	Displays a user-supplied question. Processing is halted until the user answers yes or no, at which time the status line is reset, and the answer is returned to the calling program.
sm_msg_get	Gets a message from the message file. So does sm_msgfind.

sm_setbkstat	Displays a background message, which will be hidden if the cursor enters a field that has status text, or if another message display routine is called.
sm_setstatus	Turns on or off the automatic display of alternating Ready and Wait status line messages, corresponding to an open or closed keyboard.

2.10 Altering the Operation of Other Functions

These functions affect the behavior of other parts of the run=-time system. Many have corresponding setup variables, so that you can experiment with different conditions with no need to change or recompile your application; see the Configuration Guide's section on setup.

sm_er_options	Changes the way sm_err_reset and related functions handle error message acknowledgement.
sm_ch_form_atts	Changes the display attributes of standard windows used by the run-time system.
sm_ch_emsgatt	Changes the display attributes used by sm_err_reset and sm_quiet_err.
sm_ch_qmsgatt	Changes the display attribute used by sm_query_msg.
sm_ch_stextatt	Changes the display attribute used for the status line text associated with fields.
sm_ch_umsgatt	Changes the display attributes used for error windows.
sm_mp_options	Sets options for sm_menu_proc.
sm_mp_string	Changes sm_menu_proc's interpretation of data keys.
sm_ok_options	Changes the way sm_openkeybd interprets arrow keys, field validation, and other items.
sm_zm_options	Changes details of the way zooming is done.
sm_dicname	Changes the application's data dictionary name.
sm_dd_able	Turns LDB write-through on or off.
sm_keyfilter	Turns the keystroke playback/recording mechanism on or off.
sm_dw_options	Turns JAM delayed write on or off.

2.11 Scrolling and Shifting Functions

The following routines refer specifically to scrollable or shiftable fields. Some are listed above, and repeated here for convenience. The sm_e_ prefix is not used for scrolling routines, since the element numbers of an array designate the individual on-screen fields which constitute that array, and the scrolling routines normally process an array as a whole. Individual items within a scrolling field or scrolling array are referred to by item number, which is independent of which items are currently displayed on the screen.

sm_t_scroll	Determines whether the specified field or array is scrollable.
sm_item_id	Returns the item number of the data in the current scrollable field.
sm_sc_max	Changes the maximum number of items in a scrollable field or array.
sm_clear_array	Clears all data from a scrollable array and all parallel scrollable arrays.
sm_rscroll	Scrolls a single scrollable field or array, or a group of parallel scrollable fields or arrays, by a given amount.
sm_ascroll	Scrolls a single scrollable field or array, or a group of parallel scrollable fields or arrays, to a given location.
sm_num_items	Returns the largest number of items entered so far into a scrollable field or array.
sm_t_shift	Determines whether the specified field or array is shiftable.

sm_oshift	Shifts the contents of the current shiftable field or array.
sm_sh_off	Returns the displacement of the cursor from the start of the shiftable data in the current field.
sm_ind_set	Turns shifting and scrolling field indicators on or off.

2.12 Validation Routines

sm_fval	Forces field validation and end of field processing.
sm_s_val	Performs field validation and end of field processing on all fields of the current form.
sm_bitop	Tests, sets, and clears a number of bits associated with field validation. Overlaps with some more specialized routines listed below.
sm_novalbit	Resets the validated bit of a field, so that the field will (again) be subject to validation when it is next filled or tabbed from.
sm_cl_all_mdts	Resets the modified bits of all occurrences of every field.
sm_tst_all_mdts	Returns the field and occurrence numbers of the first occurrence that has its modified bit set.

2.13 Miscellaneous

sm_occur_no	Returns the occurrence number of the data in the field where the cursor is.
sm_max_occur	Returns the number of the maximum occurrence that can be entered in the specified field or array.
sm_n_fldno	Returns the number of a field, given its name.
sm_base_fldno	Returns the field number of the base element of an array, obtained from the field number of any element of the array.
sm_size_of_array	Returns the number of elements in an array.
sm_length	Returns the length of a field (not its contents).
sm_1protect	Protects a field from some combination of tabbing into, data entry, validation, and clearing. sm_protect does all four.
sm_1unprotect	The inverse of sm_1protect.
sm_aprotect	Sets protection on an entire array. sm_aunprotect reverses the effect.
sm_sdate	Returns current system date information in a formatted string.
sm_stime	Returns current system time information in a formatted string.
sm_do_region	Paints arbitrary data on one line of the display.
sm_hlp_by_name	Invokes a named help window on the current field.
sm_flush	Forces buffered updates out to the display; JAM buffers them for greater efficiency. sm_m_flush just flushes the message line.
sm_resize	Changes the size of the display area available to JAM.
sm_bel	Beeps or flashes the terminal.
sm_keylabel	Given a logical key name, returns its label on a real keyboard.
sm_label_key	Initialize a softkey label.
sm_plcall	Invoke a routine written in the JYACC Procedural Language.
sm_putjctrl	Replace a JAM control string for a function key.
sm_rescreen	Redraw the display if it gets garbled.

2.14 Which Function Do I Use?

The JAM library provides more than one way of doing certain things. Following are some guidelines for choosing the right ones.

Screen storage and display. There are three ways to store screens for display: in individual disk files, in memory, and in a form library. This involves the sm_r_, sm_d_, and sm_l_ families of functions respectively. The first method is the most flexible, since the screen editor operates only on disk files; it is clearly best in the early development stages of an application. Form libraries are useful to reduce the clutter of a large number of screen files, and also to reduce the overhead of file system access, once an application is in production. Memory-resident forms are still faster to bring up, although they consume extra memory.

The most flexible way is to combine all three, using the memory-resident form list and SMFLIBS setup variable. The sm_r_ family of functions can be used to display disk-, library-, or memory-resident screens; if a screen needs alteration, you can simply remove it from the list or library and the altered disk copy will be used. The JAM run-time system uses the sm_r_ family, so all three options are open to you.

Field names and numbers. It is normally best to refer to fields by name rather than by number. The reason is that any addition, deletion, or shuffling of fields on the screen alters their numbering, with unpredictable consequences for programs that use hard-coded field numbers.

There are nevertheless certain situations where it makes sense to use field numbers. A common one is where a group of fields bear some relation to one another, so that their ordering will not change; in this case, the number of the first field can be obtained from its name and the rest processed in a loop on the field number. Such a loop is much more convenient than processing a group of fields by name. Attached functions are another example; they are passed the field number as a parameter, and it makes sense to use it for direct access to that field, or to offset it for access to related fields.

A further consideration for JAM applications is that unnamed fields cannot appear in the LDB or data dictionary.

Data retrieval. There are three sets of routines for retrieving data from a field: sm_fptr, sm_pkptr, and sm_getfield. The first two share a limitation: they store their results in a small ring of buffers which are overwritten after a few subsequent calls to those routines using the buffers. If you pass a returned pointer to a subroutine, bugs may develop; such data must be processed locally and soon. For more durable values use sm_getfield.

Menus. You can use sm_menu_proc or sm_choice; the latter returns you the whole text of the selected menu item, the former only the first character. If you can design your menus so that each entry is unique in the first character (perhaps by preceding the entries with labels), sm_menu_proc is easier to use. sm_mp_string can be used to make sm_menu_proc cooperate with menus whose entries are not unique in the first character, but then additional processing is required after the function returns.

If your menu is too big to fit on the screen and you want to use scrolling, or if you need to process the whole menu entry, sm_choice is preferable; it will search the whole scrolling buffer for a match. sm_menu_proc will only search the entries that appear on the screen, although it will allow you to scroll through entries with the PAGE UP and PAGE DOWN keys.

Finally, if you wish to recognize function keys as well as menu picks, use sm_menu_proc; it will return the value of the function key struck, which sm_choice cannot, since its return value is the text of the field. The JAM run-time system uses sm_menu_proc for menus with contol fields, and sm_choice for item selection screens.

3 Library Functions by Name

An alphabetical list of JAM library functions follows. For each, you will find several items:

- A synopsis similar to a C function declaration, giving the types of the arguments and return value.
- A detailed description of the function's arguments, prerequisites, results, and side effects.
- The function's return values, if it has any, and their meanings.
- A list of variants and functions that perform related actions, if there are any.
- One or two brief coding examples illustrating the function's use. These examples have no global framework; where reference is made to external functions or variables, their purpose is supposed to be apparent from their names, and no more need be read into them.

Modules calling JAM library routines should normally include smdefs.h . Those testing key values returned by sm_openkeybd, sm_menu_proc, or sm_getkey should include smkeys.h . Include files necessary for specific functions are shown in the synopsis.

NAME

sm_1clear_array - clear all data from an array

SYNOPSIS

```
int sm_1clear_array (field_number)
int field_number;
```

DESCRIPTION

Clears the onscreen and offscreen data of a scrollable array or field. The buffers that held the offscreen data are not freed, but are blanked. Clears the onscreen data of a non-scrollable array or field. This function clears an array even if it is protected from clearing (CPROTECT). Unlike sm_clear_array, it does not clear parallel scrolling arrays.

RETURNS

-1 if the field was not found; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_1clear_array (field_name);
sm_clear_array (field_number);
sm_protect (field_number);
sm_aprotect (field_number);
```

EXAMPLE

```
/* Clear the totals column of a screen when the field-erase
 * key is pressed. */

#include "smdefs.h"
#include "smkeys.h"

int key;

sm_route_table[FERA] |= RETURN;
...
if (key == FERA)
     sm_n_1clear_array ("totals");
```

NAME

sm_1protect - selectively protect a field

SYNOPSIS

```
#include "smdefs.h"

int sm_1protect (field_number, mask)
int field_number;
int mask;
```

DESCRIPTION

sm_1protect sets the protection bits to protect the specified field from any combination of data entry, tabbing into, clearing, or validation. Mnemonics for mask are defined in smdefs.h , and are listed below. Multiple sets can be done by oring mnemonics together.

Mnemonic	Meaning
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into (or from entering via any other key)
CPROTECT	protect from clearing
VPROTECT	protect from validation routines

As an example of combined protections, it is often useful to protect a field from data entry while still allowing the cursor to enter it (tabbing into). This is suitable for fields in which one selects an item from a circular scroll list, or from an item selection screen.

sm_1unprotect clears protection bits. sm_protect and its variants set all protect bits. sm_aprotect sets protection bits for all the fields in an array.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_1protect (field_name, mask);
sm_e_1protect (field_name, element, mask);
sm_1unprotect (field_number);
sm_aprotect (field_number);
sm_protect (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Protect field number 5 from data entry and clearing,
 * while still allowing the cursor to enter it. */

sm_1protect (5, EPROTECT | CPROTECT);
```

NAME

sm_1unprotect - unprotect a field

SYNOPSIS

```
#include "smdefs.h"

int sm_1unprotect (field_number, mask)
int field_number;
int mask;
```

DESCRIPTION

sm_1unprotect clears protection bits to unprotect the specified field from some combination of data entry, tabbing into, clearing, or validation.

Mnemonics for the mask are defined in smdefs.h , and are listed below. Multiple sets can be done by oring the mnemonics.

Mnemonic	Meaning
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into (or from entering via any other key)
CPROTECT	protect from clearing
VPROTECT	protect from validation routines

sm_1protect sets protection bits. sm_protect and related functions set all protect bits. sm_aprotect sets protection bits for all the fields in an array.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_1unprotect (field_name, mask);
sm_e_1unprotect (field_name, element, mask);
sm_1protect (field_number);
sm_aunprotect (field_number);
sm_unprotect (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Make field number 5 available for data entry and clearing. */

sm_1unprotect (5, EPROTECT | CPROTECT);
```

NAME

sm_achg - change the display attribute attached to a scrolling item

SYNOPSIS

```
#include "smdefs.h"

int sm_o_achg (field_number, item_id, disp_attr)
int field_number;
int item_id;
int disp_attr;
```

DESCRIPTION

Changes the display attribute attached to a scrollable item. If the item is onscreen, also changes the attribute with which the item is currently displayed. Here is a table of attribute mnemonics.

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

If disp_attr is zero, the scrollable display attribute is removed (set to zero). If the item is onscreen, it is displayed with the attribute attached to its field.

The attribute change this function makes is associated with a data item rather than a field on a form, and overrides the attribute associated with the field where the item is displayed (if any). Use sm_chg_attr to change the display attribute of a field.

Note: this function has only two variants, sm_o_achg and sm_i_achg. The other three variants (including sm_achg itself) do not exist.

RETURNS

-1 if the field isn't found or isn't scrollable, or if item_id is invalid;
0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_o_achg (field_number, item_id, disp_attr);
sm_i_achg (field_name, item_id, disp_attr);
sm_chg_attr (field_number, disp_attr);
```

EXAMPLE

```
/* Highlight the data item under the cursor in a scrolling array,
* so that the highlight will move with the item rather than
* staying on the field. */

#include "smdefs.h"

int base;
int occur;

base = sm_base_fldno (sm_getcurno ());
occur = sm_occurno ();
sm_o_achg (base, occur, RED | REVERSE);
```

NAME

sm_allget - load screen from the LDB

SYNOPSIS

```
int sm_allget (respect_flag)
int respect_flag;
```

DESCRIPTION

Copies data from the local data block to screen fields with matching names. In its loop, this function makes use of a data structure set up during screen display that identifies which fields have LDB entries.

If respect_flag is nonzero, this function does not write to fields that already contain data, or that have their MDT bits set. If the flag is zero, all fields are initialized. When this function is called by the JAM run-time system, or by your screen entry function, it does not set MDT bits for the fields it initializes.

This function is similar to the Release 3 function ldb_merge. and is called automatically by the JAM screen-display logic, unless LDB processing has been turned off using sm_dd_able. Application code should not normally need to call it.

RETURNS

Always zero.

VARIANTS AND RELATED FUNCTIONS

```
sm_lstore (item_name, value);
sm_dd_able (flag);
```

EXAMPLE

```
#include "smkeys.h"

/* If you open a window using sm_r_window, you want named
 * fields initialized from the LDB, and LDB processing is
 * off, you will need to call sm_allget explicitly. You
 * could use this, e.g., to make the LDB read-only during
 * a certain transaction. */

sm_dd_able (0);
...

if (sm_r_window ("popup", 5, 24) == 0)
{
     sm_allget (0);
     while (sm_openkeybd () != EXIT)
     {
          ...
     }
     sm_close_window ();
}
```

NAME

sm_amt_format - write data to a field, applying currency editing

SYNOPSIS

```
int sm_amt_format (field_number, buffer)
int field_number;
char *buffer;
```

DESCRIPTION

If the specified field has an amount edit, it is applied to the data in buffer. If the resulting string is too long for the field, an error message is displayed. Otherwise, sm_putfield is called to write the edited string to specified field.

If the field has no amount edit, sm_putfield is called with the unedited string.

RETURNS

-1 if the field is not found or the item ID is out of range; -2 if the edited string will not fit in the field; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_amt_format (field_name, element, buffer);
sm_i_amt_format (field_name, occurrence, buffer);
sm_n_amt_format (field_name, buffer);
sm_o_amt_format (field_number, occurrence, buffer);
sm_strip_amt_ptr (field_number, text);
sm_dtofield (field_number, real_value);
```

EXAMPLE

```
#include "smdefs.h"

/* Write a list of real numbers to the screen. The first
 * and last fields in the list are tagged with special names.
 * Actually, sm_dtofield () would be more convenient.
 */
int k, first, last;
char buf[256];
double values[];
     /* set up elsewhere... */

last = sm_n_fldno ("last");
first = sm_n_fldno ("first");
for (k = first; k && k <= last; ++k)
{
     sprintf (buf, "%lf", values[k - first]);
     sm_amt_format (k, buf);
}
```

NAME

sm_aprotect - protect an array

SYNOPSIS

```
#include "smdefs.h"

int sm_aprotect (field_number, mask)
int field_number;
int mask;
```

DESCRIPTION

sm_aprotect sets protection bits for every field in the array containing field_number. The fields are then protected from some combination of data entry, tabbing into, clearing, or validation, according to mask. If the field is scrolling, all offscreen items are protected as well.

Mnemonics for mask are defined in smdefs.h , and are listed below. Multiple sets can be done by oring the mnemonics.

Mnemonic	Meaning
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into (or from entering via any other key)
CPROTECT	protect from clearing
VPROTECT	protect from validation routines

sm_aunprotect clears protection for an array. sm_protect and variants set all protect bits for a single field. sm_1protect sets protection bits for a single field; such changes will supersede the array protection for onscreen elements, but the array protection will remain in effect for offscreen items.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_aprotect (field_name, mask);
sm_aunprotect(field_number, mask);
sm_1protect(field_number, mask);
sm_protect(field_number, mask);
```

EXAMPLE

```
#include "smdefs.h"

/* Postpone calculations by protecting the totals column from
* validation; this will prevent execution of its math edit. */

sm_n_aprotect ("subtotals", VPROTECT);
```

NAME

sm_ascroll - scroll to a given occurrence

SYNOPSIS

```
int sm_ascroll (field_number, occurrence)
int field_number;
int occurrence;
```

DESCRIPTION

This function scrolls the designated field so that the indicated occurrence appears there. The field need not be the first element of a scrolling array; you could use this function, for instance, to place the nineteenth item in the third onscreen element of a five-element scrolling array.

The validity of certain combinations of parameters depends on the exact nature of the field. For instance, if field number 7 is the third element of a scrolling array, the call

```
sm_ascroll (7, 1);
```

will fail on a regular scrolling array but succeed if scrolling is circular.

Parallel arrays or fields will, of course, scroll along with the target array or field.

RETURNS

-1 if field or occurrence specification is invalid, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
int sm_n_ascroll (field_name, occurrence);
int sm_rscroll (field_number, count);
int sm_t_scroll (field_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Scroll the "records" array (and those parallel to it)
 * to the line indicated in another field on the screen. */

#define GOTO_LINE PF4

if (sm_openkeybd () == GOTO_LINE)
{
     sm_n_ascroll ("records", sm_n_intval ("line");
}
```

NAME

sm_aunprotect - unprotect an array

SYNOPSIS

```
#include "smdefs.h"

int sm_aunprotect (field_number, mask)
int field_number;
int mask;
```

DESCRIPTION

sm_aunprotect clears protection bits for every field in the array containing field_number, and for scrolling items if the array is scrolling. The fields are then unprotected from some combination of data entry, tabbing into, clearing, or validation, according to mask.

Mnemonics for mask are defined in smdefs.h , and are listed below. Multiple sets can be done by oring the mnemonics.

Mnemonic	Meaning
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into (or from entering via any other key)
CPROTECT	protect from clearing
VPROTECT	protect from validation routines

sm_aprotect sets protection bits. sm_protect and related functions set all protect bits for a single field. sm_1protect sets protection bits for a single field; such changes will supersede the array protection for onscreen elements, but the array protection will remain in effect for offscreen items.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_aunprotect (field_name, mask);
sm_aprotect(field_number);
sm_1unprotect(field_number);
sm_unprotect(field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* open all fields in array number 14 to data entry and clearing */

sm_aunprotect (14, EPROTECT | CPROTECT);
```

NAME

sm_backtab - backtab to the previous start of an unprotected field

SYNOPSIS

```
void sm_backtab ();
```

DESCRIPTION

If the cursor is in an unprotected field, but not in the first enterable position, it is moved to the first enterable position of that field. Otherwise, it is moved to the first enterable position of the previous unprotected field. If the cursor is in the first position of the first unprotected field on the form, or before the first unprotected field on the form, it wraps backward into the last unprotected field. If there are no unprotected fields, the cursor doesn't move.

If the destination field is shiftable, it is reset according to its justification. Note that the first enterable position depends on the justification of the field and, in digits-only fields, on the presence of punctuation.

The previous field here means the field with the next lower number; field numbers increase from left to right within a display line, and from top to bottom. This function disregards next field edits.

EXAMPLE

```
#include "smkeys.h"

/* Back the cursor up if the user strikes a key indicating
 * s/he has made a particular mistake. */
int key;

do {
     key = sm_openkeybd ();
     if (key == PF5)
     {
          sm_quiet_err ("OK, start over");
          sm_backtab ();
     }
} while (key != EXIT && key != XMIT);
```

NAME

sm_base_fldno - get the field number of the first element of an array

SYNOPSIS

```
int sm_base_fldno (field_number)
int field_number;
```

DESCRIPTION

If the field specified by field_number is an array element, this function returns the field number of the first element (base) of the array.

If the field is not an array element, it returns field_number.

RETURNS

The field number of the base element of the array containing the specified field, or 0 if the field number is out of range.

VARIANTS AND RELATED FUNCTIONS

EXAMPLE

```
#include "smdefs.h"

/* Highlight the data item under the cursor in a scrolling array,
 * so that the highlight will move with the item rather than
 * staying on the field. */

int base;
int occur;

base = sm_base_fldno (sm_getcurno ());
occur = sm_occur_no ();
sm_o_achg (base, occur, RED | REVERSE);
sm_gofield (base);
```

NAME

sm_bel - beep!

SYNOPSIS

```
void sm_bel ()
```

DESCRIPTION

Causes the terminal to beep, ordinarily by transmitting the ASCII BEL code to it. If there is a BELL entry in the video file, sm_bel will transmit that instead, usually causing the terminal to flash instead of beeping.

Even if there is no BELL entry, use this function instead of sending a BEL, because certain displays use BEL as a graphics character.

Including a %B at the beginning of a message displayed on the status line will cause this function to be called.

EXAMPLE

```
#include "smkeys.h"

/* Beep at unwanted function keys. */
int key;

switch (key = sm_openkeybd ())
{
     case PF1:
          save_something ();
          break;
     case PF2:
          discard_something ();
          break;
     default:
          sm_bel();
          break;
}
```

NAME

sm_bitop - manipulate validation and data editing bits

SYNOPSIS

```
#include "smbitops.h"

int sm_bitop(field_number, action, bit)
int field_number;
int action;
int bit;
```

DESCRIPTION

You can use this function to inspect and modify validation and data editing bits of screen fields, without reference to internal data structures. The first parameter identifies the field to be operated upon.

Action can include a test and at most one manipulation, from the following table of mnemonics:

Mnemonic	Meaning
BIT_CLR	Turn bit off BIT_SET
	Turn bit on BIT_TOGL
	Flip state of bit BIT_TST
	Report state of bit

The third parameter is a bit identifier, drawn from the following table:

Character edits

N_ALL	N_DIGIT N_YES_NO
	N_ALPHA N_NUMERIC
	N_ALPHNUM N_FCMASK

Field edits

N_RTJUST	N_REQD N_VALIDED
	N_MDT N_CLRINP
	N_MENU N_UPPER
	N_LOWER N_RETENTRY
	N_FILLED N_NOTAB
	N_WRAP N_EPROTECT
	N_TPROTECT N_CPROTECT
	N_VPROTECT N_ALLPROTECT
	N_ADDLEDS

The character edits are not, strictly speaking, bits; you cannot toggle them, but the other functions work as you would expect. N_ALLPROTECT is a special value meaning all four protect bits at once.

N_VALIDED and N_MDT are the only bit operations that can apply offscreen. All other bit operations are attached to fixed onscreen positions.

This function has two variants, sm_a_bitop and sm_t_bitop, which perform the requested bit operation on all elements of an array. Their synopses appear below. If you include BIT_TST, these variants return 1 only if bit is set for every element of the array. The variants sm_i_bitop and sm_o_bitop are restricted to N_VALIDED and N_MDT.

RETURNS

-1 if the field or occurrence cannot be found; -2 if the action or bit identifiers are invalid; 1 if there was no error, the action included a test operation, and bit was set; -3 if sm_i_bitop or sm_o_bitop were called with bit set to something other than N_VALIDED or N_MDT; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_e_bitop (array_name, element, action, bit);
     sm_i_bitop (array_name, occurrence, action, bit);
     sm_n_bitop (field_name, action, bit);
     sm_o_bitop (field_number, occurrence, action, bit);

     sm_a_bitop (array_name, action, bit);
     sm_t_bitop (array_number, action, bit);
```

EXAMPLE

```
#include "smbitops.h"

/* Check whether a field is valid. If not, place the
 * cursor there. */

if (! sm_n_bitop ("operation", BIT_TST, N_VALIDED))
{
     sm_n_gofield ("operation");
}

/* Make the array "quantities" required. */

sm_a_bitop (sm_n_fldno ("quantities"), BIT_SET, N_REQD);
```

NAME

sm_c_off - turn the cursor off

SYNOPSIS

```
void sm_c_off ();
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is off. The normal setting is in effect except when a block cursor is in use, as during menu processing (cursor off); while screen manager functions are writing to the display (cursor off); and within certain error message display functions (cursor on).

If the display cannot turn its cursor on and off (CON and COF entries are not defined in the video file), this function will have no effect.

VARIANTS AND RELATED FUNCTIONS

```
sm_c_on ();
```

EXAMPLE

```
sm_err_reset("Verify that the cursor is turned ON");
sm_c_off();
sm_emsg("Verify that the cursor is turned OFF");
sm_c_on();
sm_emsg("Verify that the cursor is turned ON");
```

NAME

sm_c_on - turn the cursor on

SYNOPSIS

```
void sm_c_on ();
```

DESCRIPTION

This function notifies JAM that the normal cursor setting is on. The normal setting is in effect except when a block cursor is in use, as during menu processing (cursor off); while screen manager functions are writing to the display (cursor off); and within certain error message display functions (cursor on).

If the display cannot turn its cursor on and off (CON and COF entries are not defined in the video file), this function will have no effect.

VARIANTS AND RELATED FUNCTIONS

```
sm_c_off ();
```

EXAMPLE

```
sm_err_reset("Verify that the cursor is turned ON");
sm_c_off();
sm_emsg("Verify that the cursor is turned OFF");
sm_c_on();
sm_emsg("Verify that the cursor is turned ON");
```

NAME

sm_c_vis - turn cursor position display on or off

SYNOPSIS

```
void sm_c_vis (disp)
int disp;
```

DESCRIPTION

If disp is non-zero, subsequent messages displayed on the status line, including background status messages, will include the cursor's screen position. If the message would overlap the cursor position display, it is truncated. If disp is zero, subsequent messages displayed on the status line will not include the cursor's position.

If the CURPOS entry in the video file is not defined, this function will have no effect; the cursor position will not appear. JAM uses an asynchronous function and a status line function to perform the cursor position display; if the application has previously installed either of those, this function will clobber it.

VARIANTS AND RELATED FUNCTIONS

```
sm_u_async ();
sm_u_statfnc ();
sm_install (which_hook, what_func, howmany);
```

EXAMPLE

```
#include "smkeys.h"

/* Toggle the cursor position display on or off when the
* PF10 key is struck. The first time the key is struck,
* it will go on. */

static int cpos_on = 0;

switch (sm_openkeybd ())
{
...
case PF10:
     sm_c_vis (cpos_on ^= 1);
...
}
```

NAME

sm_calc - perform a calculation

SYNOPSIS

```
int sm_calc (field_number, item_id, expression)
int field_number;
int item_id;
char *expression;
```

DESCRIPTION

The field_number and item_id parameters identify the field and item with which the calculation is associated. (If the field is not scrollable, item_id should be set to zero.) Expression is a calculation, written as specified in the JAM Author's Guide. Briefly, a calculation contains an optional precision specifier, %m.n; a destination field identifier; an equal sign; and a math expression. The expression uses conventional arithmetic operators and parentheses in infix notation, with a few special unary operators. It and the destination field identifier may specify fields by name, absolute number, or relative number.

If the calculation begins with % the rounding information is extracted. Otherwise, rounding information is taken from the float or double data type edit, if any, attached to the destination field; or from the amount edit, if any, attached to the destination field. If none of the above are available, the default rounding to 2 decimal places is performed.

If the destination field is a date field, the value of the expression is formatted as a date. sm_calc provides a way of placing arbitrary dates in fields, through the @date expression. You should call sm_calc with an argument in the following form:

```
destination-field = @date (your-date)
```

where destination-field identifies a field by name or number as defined in the Author's Guide, and your-date is formatted as MM/DD/YYYY; assuming that the destination field is a date field, it will be written out in the proper format. This is presently the only way of getting an arbitrary date, properly formatted, into a date field.

If a math error such as divide by zero or wrong date format occurs, a message is presented to the operator, and the function returns -1.

RETURNS

-1 is returned if a math error occurred. 0 is returned otherwise.

EXAMPLE

```
/* Place a famous date in a field. */

sm_calc ("day1 = @date(07/04/1776)");
```

NAME

sm_cancel - reset the display and exit

SYNOPSIS

```
void sm_cancel ();
```

DESCRIPTION

This routine is installed by sm_initcrt to be executed if a keyboard interrupt occurs. It calls sm_resetcrt to restore the display to the operating system's default state, and exits to the operating system.

If your operating system supports it, you can also install this function to handle conditions that normally cause a program to abort. If a program aborts without calling sm_resetcrt, you may find your terminal in an odd state; sm_cancel can prevent that.

EXAMPLE

```
/* the following program segment could be found in some
 * error routines */

if (error)
{
     sm_quiet_err("fatal error -- can't continue!\n");
     sm_cancel();
}


/* The following code can be used on a UNIX system to
 * install sm_cancel() as a signal handler. */

#include <signal.h>

extern void sm_cancel ();

signal (SIGTERM, sm_cancel);
```

NAME

sm_ch_emsgatt - change the standard error message attributes

SYNOPSIS

```
     #include "smdefs.h"

     void sm_ch_emsgatt (noisy_att, quiet_att)
     int noisy_att, quiet_att;
```

DESCRIPTION

Changes the display attributes used by sm_err_reset and sm_quiet_err. Noisy_att is used for the message by sm_err_reset and for the message prefix (normally "ERROR:") displayed by sm_quiet_err. Quiet_att is used for the message body displayed by sm_quiet_err.

If either argument is zero, the corresponding display attribute is unchanged. If an argument is nonzero but no color is specified, and the display does not support background color, the color is made WHITE.

The following display attribute mnemonics may be used in the arguments to this function:

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

If sm_ch_emsgatt is never called, sm_err_reset uses WHITE with BLINK, and HILIGHT if it is available. The default attribute for the message body displayed by sm_quiet_err is just WHITE.

If you define the SMCHEMSGATT variable in your setup file, this function will automatically be called with the parameters you provide there.

VARIANTS AND RELATED FUNCTIONS

```
     sm_err_reset (message);
     sm_quiet_err(message);
     sm_er_options (key, discard);
```

EXAMPLE

```
#include "smdefs.h"

sm_ch_emsgatt(NORMAL_ATTR, NORMAL_ATTR);
sm_err_reset("Verify this message is displayed in white.");

sm_ch_emsgatt(REVERSE, NORMAL_ATTR);
sm_err_reset("Verify this message is in reverse video.");
```

NAME

sm_ch_form_atts - change the standard JAM library window attributes

SYNOPSIS

```
#include "smdefs.h"

void sm_ch_form_atts (bord_style, bord_attr,
     protect_attr, entry_attr)
     int bord_style;
     int bord_attr;
     int protect_attr;
     int entry_attr;
```

DESCRIPTION

Changes the display characteristics of windows that are part of the library. Currently, there are four such windows: the system calls (SPF2) window; the go-to-screen (SPF3) window; the error window, used to display a message too long to fit on the status line; and the hit space window, which pops up if you hit the wrong key to acknowledge an error message.

This function is intended to be called once, at the beginning of an application, to set the display characteristics of the library windows to harmonize with the application's own forms.

If bord_style is less than 0, the windows are made borderless. Otherwise, it is made the border style number (0 through 9), and border_attribute, if nonzero, is made the border attribute.

If protect_attr is nonzero, it is used for protected fields that contain messages. If entry_attr is nonzero, it is used for the unprotected data entry fields.

If you define the SMCHFORMATTS variable in your setup file, this function will automatically be called with the parameters you provide there.

EXAMPLE

```
#include "smdefs.h"

/* Give the library windows a colorless graphics border
* (conventionally style 1), with yellow message fields and
* green data entry field. */

sm_ch_form_atts (1, NORMAL_ATTR, YELLOW, GREEN);
```

NAME

sm_ch_qmsgatt - change the standard query message attribute

SYNOPSIS

```
#include "smdefs.h"

void sm_ch_qmsgatt (disp_attr)
int disp_attr;
```

DESCRIPTION

Changes the display attribute used by sm_query_msg. If no color is specified, it is set to WHITE.

The argument disp_attr is the logical sum of one or more of the following:

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

If sm_ch_qmsgatt is never called, sm_query_msg uses WHITE and REVERSE, with HILIGHT if it is available.

If you define the SMCHQMSGATT variable in your setup file, this function will automatically be called with the parameters you provide there.

VARIANTS AND RELATED FUNCTIONS

```
sm_query_msg (question);
```

EXAMPLE

```
#include "smdefs.h"

sm_ch_qmsgatt(NORMAL_ATTR);
sm_query_msg("Verify that this message is displayed in white.");

sm_ch_qmsgatt(GREEN);
sm_query_msg("Verify that this message is displayed in green.");
```

NAME

sm_ch_stextatt - change the status text display attribute

SYNOPSIS

```
#include "smdefs.h"

void sm_ch_stextatt (disp_attr)
int disp_attr;
```

DESCRIPTION

Changes the display attribute used for displaying status text associated with a field. If no color is specified, it is set to WHITE.

The argument disp_attr is the logical sum of one or more of the following:

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

If sm_ch_stextatt is never called, the display attribute used for the status text associated with a field is normal intensity WHITE.

If you define the SMCHSTEXTATT variable in your setup file, this function will automatically be called with the parameters you provide there.

EXAMPLE

```
#include "smdefs.h"

/* Change the default status text attribute to bright green. */

sm_ch_stextatt (GREEN | HILIGHT);
```

NAME

sm_chg_attr - change the display attribute of a field

SYNOPSIS

```
#include "smdefs.h"

void sm_chg_attr (field_number, disp_attr)
int field_number, disp_attr;
```

DESCRIPTION

Changes the display attribute of a field. If the field is scrolling, each data item may also have a display attribute, which will override the field display attribute when the item arrives onscreen; use sm_achg to change scrolling attributes.

Disp_attr is the logical sum of one or more of the following. If no color is specified, and BLACK is not a valid color, sm_chg_attr will automatically include the color WHITE.

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_chg_attr (field_name, disp_attr);
sm_e_chg_attr (field_name, element, disp_attr);
sm_o_chg_attr (field_number, occurrence, disp_attr);
sm_i_achg (field_name, item_id, disp_attr);
sm_o_achg (field_number, item_id, disp_attr);
```

EXAMPLE

```
#include "smdefs.h"

sm_chg_attr (1, NORMAL_ATTR);
sm_err_reset ("Verify that the text in field 1 is white.");

sm_chg_attr (1, BLANK);
sm_err_reset ("Verify that the text in field 1 is invisible.");
```

NAME

sm_choice - get item selection

SYNOPSIS

```
#include "smdefs.h"

char *sm_choice (type)
int type;
```

DESCRIPTION

This is a menu-handling function, similar in some respects to sm_menu_proc. It enables you to tab, backtab, arrow and scroll through a screen, in order to select the contents of one of the fields or scrolling items. The entry at which the cursor is positioned is shown in reverse video; fields that are blank in their first position, and those without a MENU field edit, will be skipped.

Hitting a key that matches the first character of a screen entry causes the cursor to be positioned there; if more than one entry begins with that character, the cursor is positioned to the first entry following its current location. Entries are searched by field number. Arrays and scrolls, however, are searched in their entirety following their first field, and scrolling occurs automatically. If type is UPPER (or LOWER), an alphabetic key is translated to upper (or lower) case before a match is attempted; if it is UPPER | LOWER, both are tried; and if type has any other value, the entry is not translated.

sm_choice returns to the calling program only when you hit the TRANSMIT or EXIT key. It ignores menu return codes attached to fields; the returned value is the result of a call to sm_fptr.

The functions sm_mp_options and sm_mp_string, which control the behavior of sm_menu_proc, do not affect sm_choice.

Menu control strings are not executed within this function, but at a higher level of the JAM run-time system. If you call this function, do not expect your selection's control string to be executed.

RETURNS

The contents of the selected field if TRANSMIT was hit, or 0 if EXIT was hit.

VARIANTS AND RELATED FUNCTIONS

```
sm_menu_proc (type);
sm_fptr (field_number);
```

EXAMPLE

```
#include "smdefs.h"
#define WHI    WHITE | HILIGHT
#define NOT_UL ~(UPPER|LOWER)

#define INS1 "Move cursor to 2nd field and press TRANSMIT"
#define INS5 "Press 'c' and TRANSMIT"

/* Move the cursor to the second field
 * and press the TRANSMIT key. Verify that a pointer
 * to the text of the second field is returned. */

sm_d_msg_line(INS1,WHI);
if (strcmp("bcdefgh", sm_choice(NOT_UL)))
     sm_err_reset ("Bad choice");

/* Press the first letter of the first item of
 * the third field ("c" or "C") and verify that the
 * cursor is located correctly. */

sm_d_msg_line(INS5,WHI);
if (strcmp("CDE", sm_choice(UPPER)))
     sm_err_reset ("Bad choice");
```

NAME

     sm_cl_all_mdts - clear all MDT bits

SYNOPSIS

```
     void sm_cl_all_mdts ();
```

DESCRIPTION

Clears the MDT (modified data tag) of every data item, both onscreen and off.

JAM sets the MDT bit of an occurrence to indicate that it has been modified, either by keyboard entry or by a call to a function like sm_putfield, since the screen was first displayed.

VARIANTS AND RELATED FUNCTIONS

```
     sm_mdt_clear (field_number);
     sm_mod_test (field_number);
     sm_tst_all_mdts (field_number, occurrence);
```

EXAMPLE

```
#include "smdefs.h"

/* Clear MDT for all fields on the form; then write
 * data to the last field, and check that its MDT is
 * the first one set. */

int occurrence;

sm_cl_all_mdts();
sm_putfield (sm_numflds, "Hello");
if (sm_tst_all_mdts (&occurrence) != sm_numflds)
     sm_err_reset ("Something is rotten in the state of Denmark.");
```

NAME

    sm_cl_everyfield - clear all fields, protected or not

SYNOPSIS

```
     void sm_cl_everyfield ();
```

DESCRIPTION

Erases all fields on the current screen, including protected fields and offscreen data. Date and time fields that take system values are re-initialized.

VARIANTS AND RELATED FUNCTIONS

```
     sm_cl_unprot ();
```

EXAMPLE

```
/* The following code effectively binds sm_cl_everyfield
 * to the CLEAR ALL key, instead of sm_cl_unprot (the
 * normal binding). */

#include "smdefs.h"
#include "smkeys.h"

int key;

/* Make the CLEAR ALL key returnable and NOT executable. */
sm_route_table[CLR] = RETURN;

while ((key = sm_openkeybd ()) != EXIT)
{
     if (key == CLR)
     {
          sm_cl_everyfield ();
          continue;
     }
     ...
}
```

NAME

     sm_cl_unprot - clear all unprotected fields

SYNOPSIS

```
     void sm_cl_unprot ();
```

DESCRIPTION

Erases onscreen and offscreen data from all fields that are not protected from clearing (CPROTECT). Date and time fields that take system values are re-initialized.

This function is normally bound to the CLEAR ALL key.

VARIANTS AND RELATED FUNCTIONS

```
     sm_cl_everyfield ();
     sm_1protect (field_number);
```

EXAMPLE

```
/* The following code clears all unprotected fields and puts
 * the cursor into the first one. */

sm_cl_unprot ();
sm_home ();
```

NAME

sm_clear_array - erase all data from an array

SYNOPSIS

```
int sm_clear_array (field_number)
int field_number;
```

DESCRIPTION

Clears onscreen and offscreen data of the specified array or field. If there are scrollable arrays or fields parallel to the one specified, they are also cleared.

The array indicated by the argument will be cleared regardless of protection; the protection of parallel scrolling arrays will, however, be respected.

The buffers that held the offscreen data are freed and are no longer accessible.

RETURNS

-1 if the field is not found; -2 if memory allocation fails; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_clear_array (field_name);
sm_1clear_array (field_number);
sm_aprotect (field_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Make the ERASE key clear the entire array of "names",
 * first ensuring that it will be returned to us. */

int key;

sm_route_table[FERA] |= RETURN;

while ((key = sm_openkeybd ()) != EXIT)
{
     if (key == FERA)
          sm_n_clear_array ("names");
}
```

NAME

sm_close_window - close current window

SYNOPSIS

```
int sm_close_window ();
```

DESCRIPTION

The currently open window is erased, and the screen is restored to the state before the window was opened. All data from the window being closed is lost unless LDB processing is active, in which case named fields are copied to the LDB using sm_lstore. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the position it had before the window was opened.

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

-1 is returned if there is no window open, i.e. if the currently displayed screen is a form (or if there is no screen up). 0 is returned otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_r_window (screen_name, line, column);
sm_d_window (address, line, column);
sm_l_window (library_descriptor, screen_name, line, column);
sm_wselect (window_num);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* In a validation routine, if the field contains a
 * special value, open up a window to prompt for a
 * second value and save it in another field. */

int validate (field, data, occur, bits)
char *data;
{
     char buf[256];

     if (bits & VALIDED)
          return 0;

     if (strcmp(data, "other") == 0)
     {
          sm_r_at_cur ("getsecval");
          if (sm_openkeybd () != EXIT)
               sm_getfield (buf, 1);
          else buf[0] = 0;
          sm_close_window ();
          sm_n_putfield ("secval", buf);
     }

     return 0;
}
```

NAME

```
     sm_d_at_cur - display a memory-resident window at the current cursor
                   position
```

SYNOPSIS

```
     int sm_d_at_cur (mr_screen)
     char *mr_screen;
```

DESCRIPTION

Displays a memory-resident window at the current cursor position, offset by one line to avoid hiding that line's current display. Mr_screen is the address of the screen in memory.

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the top most, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

You can use JYACC bin2c to convert screens from disk files, which you can modify using jxform, to program data structures you can compile into your application. A memory-resident screen is never altered at run-time, and may therefore be made shareable on systems that provide for sharing read-only data. sm_r_at_cur can also display memory-resident screens, if they are properly installed using sm_formlist. Memory-resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (e.g. MS-DOS).

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

0 is returned if no error occurred during display of the form; -3 is returned if no memory was available; -7 is returned if the screen contained fields that would not fit within the physical display. The screen is always restored to its previous condition.

VARIANTS AND RELATED FUNCTIONS

```
     sm_d_form (mr_screen);
     sm_d_window (mr_screen, line, column);
     sm_r_at_cur (name);
```

EXAMPLE

```
/* Display a warning message in a memory-resident window,
 * and wait for acknowledgement before continuing.
 * The warning should include the instruction,
 * "Press any key to continue." */

extern char warning[];

if (over_threshold ())         /* Externally defined condition */
{
     sm_d_at_cur (warning);
     sm_getkey ();
     sm_close_window ();
}
...
```

NAME

sm_d_form - display a memory-resident form

SYNOPSIS

```
int sm_d_form (mr_screen)
char *mr_screen;
```

DESCRIPTION

This function displays a memory-resident screen as a base form. Mr_screen is the address of the screen.

Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new screen is displayed with its upper left-hand corner at the extreme upper left of the display (position (0, 0)). Any error in this function leaves the display and JAM internals in an undefined state.

If the form contains display data that are too big for the physical display, they are truncated without warning. However, if there are fields that won't fit within the physical display, this function returns an error without displaying the form.

You can use JYACC bin2c to convert screens from disk files, which you can modify using jxform, to program data structures you can compile into your application. A memory-resident screen is never altered at run-time, and may therefore be made shareable on systems that provide for sharing read-only data. sm_r_at_cur can also display memory-resident screens, if they are properly installed using sm_formlist. Memory-resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (e.g. MS-DOS).

This function should be called by JAM applications only under unusual circumstances, as it does not update the control stack. You should execute a control string to display the form instead.

RETURNS

0 is returned if no error occurred during display of the screen. -5 is returned if, after the screen was cleared, the system ran out of memory. -7 is returned if the screen contained fields that would not fit within the display.

VARIANTS AND RELATED FUNCTIONS

```
sm_d_window (mr_screen, line, column)
sm_d_at_cur (mr_screen)
sm_r_form (screen_name)
sm_l_form (library_descriptor, screen_name)
```

EXAMPLE

```
/* Display a memory-resident form to provide a
 * blank background for what follows. */

extern char blank[];

if (sm_d_form (blank) < 0)
{
     sm_err_reset ("Error in form display - goodbye!");
     sm_cancel ();
}
...
```

NAME

sm_d_msg_line - display a message on the status line

SYNOPSIS

```
#include "smdefs.h"

void sm_d_msg_line (message, attrib)
char *message;
int attrib;
```

DESCRIPTION

The message in message is displayed on the status line, with an initial display attribute of attrib. This message overrides background status text and field status text; it will itself be overwritten by sm_err_reset and related functions, or by the ready/wait message enabled by sm_setstatus.

Several percent escapes provide control over the content and presentation of status messages. They are interpreted by sm_d_msg_line, which is eventually called by everything that puts text on the status line (including field status text). The character following the percent sign must be in upper-case; this is to avoid conflict with the percent escapes used by printf and its variants. Certain percent escapes (%W, for instance; see below) must appear at the beginning of the message, i.e. before anything except perhaps another percent escape.

- If a string of the form %Annnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table below gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear immediately after the attribute change, pad the attribute to 4 digits with leading zeroes; if the following character is not a legal hex digit, leading zeroes are unnecessary.
- If a string of the form %KKEYNAME appears anywhere in the message, KEYNAME is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the %K is stripped out and the mnemonic remains. Key mnemonics are defined in smkeys.h ; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.
- If %N appears anywhere in the message, the latter will be presented in a pop-up window rather than on the status line, and all occurrences of %N will be replaced by newlines.
- If the message begins with a %B, JAM will beep the terminal (using sm_bel) before issuing the message.
- If the message begins with %W, it will be presented in a pop-up window instead of on the status line. The window will appear near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it will appear near the top. If the message begins with %MU or %MD, and is passed to one of the error message display functions, JAM will ignore the default error message acknowledgement flag and process (for %MU) or discard (for %MD) the next character typed.

Note that, if a message containing percent escapes - that is, %A, %B, %K, %N or %W - is displayed before sm_initcrt or after %W is called, the percent escapes will show up in it.

Attribute	Hex value
BLACK	0 BLUE
	1 GREEN
	2 CYAN
	3 RED
	4 MAGENTA
	5 YELLOW
	6 WHITE
	7
B_BLACK	0 B_BLUE
	100 B_GREEN
	200 B_CYAN
	300 B_RED
	400 B_MAGENTA
	500 B_YELLOW
	600 B_WHITE
	700
BLANK	8 REVERSE
	10 UNDERLN
	20 BLINK
	40 HILIGHT
	80 DIM
	1000

If the cursor position display has been turned on (see sm_c_vis), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead.

VARIANTS AND RELATED FUNCTIONS

```
     sm_err_reset (message);
     sm_msg (message, start_column);
     sm_mwindow (text, line, column);
```

EXAMPLE

```
/* The following prompt uses labels for the EXIT and
 * return keys, and underlines crucial words. */

sm_d_msg_line ("Press %KEXIT to %A0027abort%A7, or %KNL to %A0027continue%A7.");

/* To clear the message line, use: */

sm_d_msg_line ("", 0);
```

NAME

sm_d_window - display a memory-resident window at a stated position

SYNOPSIS

```
int sm_d_window (mr_screen, line, column)
char *mr_screen;
int line, column;
```

DESCRIPTION

The memory-resident screen whose address is in mr_screen is brought up with its upper left-hand corner at (line, column). The line and column are counted from zero: if line is 1, the screen is displayed starting at the second line of the screen. Note that the window coordinates you place in JAM control strings are counted from 1 as usual.

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the top most, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

You can use JYACC bin2c to convert screens from disk files, which you can modify using jxform, to program data structures you can compile into your application. A memory-resident screen is never altered at run-time, and may therefore be made shareable on systems that provide for sharing read-only data. sm_r_at_cur can also display memory-resident screens, if they are properly installed using sm_formlist. Memory-resident screens are particularly useful in applications that have a limited number of screens, or in environments that have a slow disk (e.g. MS-DOS).

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

0 is returned if no error occurred during display of the screen. -5 is returned if, after the screen was cleared, the system ran out of memory. -7 is returned if the screen contained fields that would not fit within the display.

VARIANTS AND RELATED FUNCTIONS

```
sm_d_at_cur (mr_screen);
sm_d_form (mr_screen);
sm_r_window (screen_name, line, column);
sm_l_window (library_descriptor, screen_name, line, column);
```

EXAMPLE

```
/* Display a warning message in a memory-resident
 * window centered on the screen, and
 * wait for acknowledgement before continuing. */

extern char warning[];

if (over_threshold ())         /* Externally defined condition */
{
     sm_d_window (warning, 8, 18);
     sm_err_reset ("Press any key to continue");
     sm_close_window ();
}
...
```

NAME

sm_dblval - get the value of a field as a real number

SYNOPSIS

```
#include "smdefs.h"

double sm_dblval (field_number)
int field_number;
```

DESCRIPTION

This function returns the contents of field_number as a real number. It calls sm_strip_amt_ptr to remove superfluous amount editing characters before converting the data.

RETURNS

The real value of the field is returned. If the field is not found, the function returns 0.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_dblval (field_name, element);
sm_i_dblval (field_name, occurence);
sm_n_dblval (field_name);
sm_o_dblval (field_number, occurrence);
sm_dtofield (field_number, value, format);
sm_strip_amt_ptr (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Retrieve the value of a starting parameter. */

double param1;

param1 = sm_n_dblval ("param1");
```

NAME

sm_dd_able - turn LDB write-through on or off

SYNOPSIS

```
void sm_dd_able (flag)
int flag;
```

DESCRIPTION

During normal JAM processing, named fields in the screen and local data block are kept in sync. When a screen is brought up, values are copied in from the LDB; when control passes from the screen, values are copied back to the LDB. When application code reads or writes a value to or from a name that is not in the screen, JAM accesses the LDB instead.

sm_dd_able turns that feature off or on, according to the value of flag. It is on by default; when it is off, the LDB is never accessed. Refer to Section 9 for a full explanation.

EXAMPLE

```
/* Turn LDB write-through off. */

sm_dd_able (0);
```

NAME

sm_dicname - set data dictionary name

SYNOPSIS

```
int sm_dicname (dictionary_name)
char *dictionary_name;
```

DESCRIPTION

This function names the application's data dictionary, which is data.dic by default. It must be called before JAM initialization, in particular before sm_ldb_init is called to initialize the local data block from the data dictionary. The argument dictionary_name is a character string giving the filename; JAM will search for it in all the directories in the SMPATH variable.

You can achieve the same effect by defining the SMDICNAME variable in your setup file equal to the data dictionary name. See the section on setup files in the Configuration Guide.

RETURNS

-1 if it fails to allocate memory to store the name, 0 otherwise.

EXAMPLE

```
/* Set the name of the application's data dictionary to
* /usr/app/common.dic .*/

sm_dicname ("/usr/app/common.dic");
```

NAME

sm_disp_off - get displacement of cursor from start of field

SYNOPSIS

```
int sm_disp_off ();
```

DESCRIPTION

Returns the difference between the first column of the current field and the current cursor location. This routine ignores offscreen data; use sm_sh_off to obtain the total cursor offset of a shiftable field.

RETURNS

The difference between cursor position and start of field, or -1 if the cursor is not in a field.

VARIANTS AND RELATED FUNCTIONS

```
sm_sh_off ();
sm_getcurno ();
```

EXAMPLE

```
/* Retrieve the contents of the current field, up to
 * the cursor position, discarding the rest. This
 * example assumes the field is non-shifting and
 * left-justified. */

char buf[256];
int index;

sm_getfield (buf, sm_getcurno ());
if ((index = sm_disp_off ()) >= 0)
     buf[index] = '\0';
```

NAME

     sm_dlength - get the length of a field's contents

SYNOPSIS

```
     int sm_dlength (field_number)
     int field_number;
```

DESCRIPTION

Returns the length of data stored in field_number. The length does not include leading blanks in right justified fields, or trailing blanks in left-justified fields (which are also ignored by sm_getfield). It does include data that have been shifted offscreen.

RETURNS

     Length of field contents, or -1 if the field is not found.

VARIANTS AND RELATED FUNCTIONS

```
     sm_e_dlength (field_name, element);
     sm_i_dlength (field_name, occurrence);
     sm_n_dlength (field_name);
     sm_o_dlength (field_number, occurrence);
     sm_length (field_number);
```

EXAMPLE

```
/* Save the contents of the "rank" field in a buffer
 * of the proper size. */

char *save_rank;

if ((save_rank = malloc (sm_n_dlength ("rank") + 1)) == 0)
     punt ("Malloc error");
sm_n_getfield (save_rank, "rank");
```

NAME

sm_do_region - rewrite part or all of a screen line

SYNOPSIS

```
     #include "smdefs.h"

     void sm_do_region (line, column, length, attribute,
          text)
     int line, column, length, attribute;
     char *text;
```

DESCRIPTION

The screen region defined by line, column, and length is rewritten. Line and column are counted from zero, with (0, 0) the upper left-hand corner of the screen. If text is zero, the actual text that is written is taken from the screen buffer; if text is shorter than length, it is padded out with blanks. In any case, the display attribute of the whole area is changed to attribute. A table of attribute mnemonics follows.

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

The fifth argument, if passed as zero, must be cast, as in

```
     sm_do_region (line, col, length, attrib, (char *)0);
```

EXAMPLE

```
#include "smdefs.h"
#include "smvideo.h"

/* Place a centered text string in a part of the screen where
 * there is (hopefully) no field. The line number is
 * made zero-relative. */

void centerstring (text, line)
char *text;
{
     int offset, length = strlen (text);

     offset = (*sm_video[V_COLMS] - length) / 2;
     if (offset < 0)
          return;
     sm_do_region (line - 1, offset, length, REVERSE | WHITE, text);
}
```

NAME

    sm_doccur - delete occurrences

SYNOPSIS

```
int sm_o_doccur (field_number, occurrence, count);
int field_number;
int occurrence;
int count;
```

DESCRIPTION

Deletes count data items beginning with the specified occurrence, moving all following occurrences up. If there are not enough occurrences, fewer than the requested number will be deleted. The memory associated with the deleted data items is released. If count is negative, occurrences are inserted instead, subject to limitations explained at sm_ioccur.

If occurrence is zero, the occurrence used is that of field_number. If occurrence is nonzero, however, it is taken relative to the first field of the array in which field_number occurs.

This function is normally bound to the DELETE LINE key. It has only the o_ and i_ variants; the others, including sm_doccur itself, do not exist.

RETURNS

-1 if the field or occurence number was out of range; -3 if insufficient memory was available; otherwise, the number of occurrences actually deleted (zero or more).

VARIANTS AND RELATED FUNCTIONS

```
sm_i_doccur(field_name, occurrence, count);
sm_i_ioccur(field_name, occurrence, count);
sm_o_ioccur(field_number, occurrence, count);
```

NAME

sm_dtofield - write a real number to a field

SYNOPSIS

```
int sm_dtofield (field_number, value, format)
int field_number;
double value;
char *format;
```

DESCRIPTION

The real number value is converted to human-readable form, according to format, and moved into field_number via a call to sm_amt_format. If the format string is null, the number of decimal places will be taken from a data type edit, if one exists; failing that, from a currency edit, if one exists; or failing that, will default to 2.

The format string is in the style of the C library function printf, q.v. It should have the form %m.nf, where m is the total output width, n is the number of decimal places, and both are optional.

RETURNS

-1 is returned if the field is not found. -2 is returned if the output would be too wide for the destination field. 0 is returned otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_dtofield (field_name, element, value, format);
sm_i_dtofield (field_name, occurrence, value, format);
sm_n_dtofield (field_name, value, format);
sm_o_dtofield (field_number,occurrence, value, format);
sm_dblval (field_number)
sm_amt_format (field_number, text);
```

EXAMPLE

```
/* Place the value of pi on the screen, using the formatting
 * attached to the field. */

sm_n_dtofield ("pi", 3.141596, (char *)0);

/* Do it again, using only three decimal places.

sm_n_dtofield ("pi", 3.141596, "%5.3f");
```

NAME

```
     sm_dw_options - turn delayed write on or off
```

SYNOPSIS

```
     #include "smdefs.h"

     void sm_dw_options (flag)
     int flag;
```

DESCRIPTION

This function turns the delayed-write feature of the JAM library on or off. It is on by default. The value of flag should be either DW_ON or DW_OFF.

When delayed write is on, output from library functions is not sent immediately to the display, but is used to update a screen image in memory. When it becomes necessary to update the display (usually when the keyboard is opened), output is sent to the display a line at a time, and a check is made for keyboard input between each line. If you press a key before the screen has been fully updated, JAM processes the key before doing any more output. This scheme make JAM more responsive, especially at low baud rates.

You may find it advantageous to turn delayed write off while debugging an application, so that you can see the output produced by each function call. In this case you should also investigate the BUFSIZ video file entry, which controls the output buffer size; see the video manual in the Configuration Guide. When delayed write is off, the display will still not be flushed until the keyboard is opened; however, JAM will not check for input while writing to the display.

If you define the SMDWOPTIONS variable in your setup file, it will cause this function to be called automatically during start-up with the parameter you specify there.

VARIANTS AND RELATED FUNCTIONS

```
     sm_getkey ();
     sm_ungetkey (key);
     sm_flush ();
```

EXAMPLE

```
#include "smdefs.h"

/* Turn delayed write off for debugging. */

#ifdef DEBUG
sm_dw_options (DW_OFF);
#endif
```

newpage NAME

sm_e_1protect selectively protect an array element sm_e_1unprotect
selectively unprotect an array element sm_e_amt_format
format data and write to an array element sm_e_bitop
manipulate field edit bits sm_e_chg_attr
change display attribute of an array element
sm_e_dblval
get decimal value of array element sm_e_dlength
get length of data stored in an array element
sm_e_dtofield
write decimal value to array element sm_e_fldno
* see next page * sm_e_fptr
get copy of array element's data sm_e_fval
force validation of an array element sm_e_getfield
copy data from array element into buffer sm_e_gofield
position cursor to an array element sm_e_intval
get integer value of data in an array element
sm_e_is_yes
test array element for yes sm_e_itofield
write integer to an array element sm_e_lngval
get long integer value of data in an array element
sm_e_ltofield
write long integer to an array element sm_e_mdt_clear
reset the MDT bit of an array element sm_e_mod_test
test the MDT bit of an array element sm_e_novalbit
reset the validated bit of an array element
sm_e_off_gofield
move cursor to specified offset within an array element
sm_e_protect
protect an array element from data entry sm_e_putfield
write data string to an array element sm_e_unprotect
allow data entry into an array element

DESCRIPTION

Each of the above functions accesses one element of an array by field name and element number. For the description of sm_e_fldno, see next page. For a description of any other particular function, look under the related function without e_ in its name. For example, sm_e_amt_format is described under sm_amt_format.

Despite the fact that they take a field name as argument, these functions do not search the LDB for names not found in the screen.

NAME

sm_e_fldno - get the field number of an array element

SYNOPSIS

```
int sm_e_fldno (field_name, element)
char *field_name;
int element;
```

DESCRIPTION

Returns the field number of an array element specified by field_name and element.

If element is zero, returns the field number of the named field, or of the base element of the named array.

RETURNS

0 if the name is not found, if the element number exceeds 1 and the named field is not an array, or if the element number exceeds the size of the array. Otherwise, returns an integer between 1 and the maximum number of fields on the current form.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_fldno (field_name);
```

EXAMPLE

```
/* Retrieve the field numbers of the first three elements of the
* "horses" array. */

int winnum, placenum, shownum;

winnum = sm_e_fldno ("horses", 1);
placenum = sm_e_fldno ("horses", 2);
shownum = sm_e_fldno ("horses", 3);
```

NAME

sm_edit_ptr - get special edit string

SYNOPSIS

```
#include "smdefs.h"
char *sm_edit_ptr (field_number, command)
int field_number, command;
```

DESCRIPTION

This function searches the special edits area of a field or array for an edit of type command. The command should be one of the following values, which are defined in smdefs.h :

```
     Command          Contents of edit string

     NAMED     Field name RANGEL
               Low bound on range; up to 9 permitted RANGEH
               High bound on range; up to 9 permitted NEXTFLD
               Next field (contains both primary and
               alternate fields) DOLLARS
               Amount field formatting parameters TEXT
               Status line prompt CPROG
               Name of field exit function HELPSCR
               Name of help screen CALC
               Math expression executed at field exit DATEFLD
               Format string for system-supplied date TIMEFLD
               Format string for system-supplied time CKDIGIT
               Flag and parameters for check digit FTYPE
               Data type for inclusion in structure USRDATE
               Format string for user-supplied date USRTIME
               Format string for user-supplied time ITEMSCR
               Name of item selection screen HARDHLP
               Name of automatic help screen HARDITM
               Name of automatic item selection screen MEMO1
               Nine arbitrary user-supplied text strings ...
               ... MEMO9
               ... FE_CPROG
               Name of field entry function EDT_BITS
               For internal use: bit string showing what
               other edits are present. Always first. RETCODE
               Return value for menu or return entry field JPLTEXT
               Attached JPL code, or JPL file name SUBMENU
               Name of pull-down menu screen CMASK
               Regular expression for field validation CCMASK
               Regular expression for character validation TABLOOK
               Name of screen for table-lookup validation
```

The string returned by this function has the command code in its second byte, and the total length of the string (including the two overhead bytes and any terminators) in its first; the body of the edit follows. If the field has no edit of type command, this function returns a null string. If a field has multiple edits of one type, such as RANGEH or RANGEL, the first one is returned; the rest follow it.

This function is especially useful for retrieving user-defined information contained in MEMO edits.

RETURNS

A pointer to the first (length) byte of the special edit for this field is returned. Zero is returned if the field or edit is not found.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_edit_ptr (field_name, command);
```

EXAMPLE

```
#include "smdefs.h"

/* Useful little function to retrieve the name of a field. */

char *field_name (fieldnum)
int fieldnum;
{
     char *name;

     if (fieldnum < 1 || fieldnum > sm_numflds)
          return 0;

     if ((name = sm_edit_ptr (fieldnum, NAMED)) == 0)
          return 0;
     return name + 2;
}
```

NAME

sm_emsg - display an error message and reset the message line, without turning on the cursor

SYNOPSIS

```
void sm_emsg (message)
char *message;
```

DESCRIPTION

This function displays message on the status line, if it fits, or in a window if it is too long; it remains visible until the operator presses a key. The function's exact behavior in dismissing the message is subject to the error message options; see sm_er_options.

sm_emsg is identical to sm_err_reset, except that it does not attempt to turn the cursor on before displaying the message. It is similar to sm_qui_msg, which inserts a constant string (normally "ERROR:") before the message. That string may be altered by changing the SM_ERROR entry in the message file.

Several percent escapes provide control over the content and presentation of status messages. They are interpreted by sm_d_msg_line, which is eventually called by everything that puts text on the status line (including field status text). The character following the percent sign must be in upper-case; this is to avoid conflict with the percent escapes used by printf and its variants. Certain percent escapes (%W, for instance; see below) must appear at the beginning of the message, i.e. before anything except perhaps another percent escape.

- If a string of the form %Annnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table below gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear immediately after the attribute change, pad the attribute to 4 digits with leading zeroes; if the following character is not a legal hex digit, leading zeroes are unnecessary.
- If a string of the form %KKEYNAME appears anywhere in the message, KEYNAME is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the %K is stripped out and the mnemonic remains. Key mnemonics are defined in smkeys.h ; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.
- If %N appears anywhere in the message, the latter will be presented in a pop-up window rather than on the status line, and all occurrences of %N will be replaced by newlines.
- If the message begins with a %B, JAM will beep the terminal (using sm_bel) before issuing the message.
- If the message begins with %W, it will be presented in a pop-up window instead of on the status line. The window will appear near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it will appear near the top. If the message begins with %MU or %MD, and is passed to one of the error message display functions, JAM will ignore the default error message acknowledgement flag and process (for %MU) or discard (for %MD) the next character typed.

Note that, if a message containing percent escapes - that is, %A, %B, %K, %N or %W - is displayed before sm_initcrt or after %W is called, the percent escapes will show up in it.

Attribute	Hex value
BLACK	0 BLUE
	1 GREEN
	2 CYAN
	3 RED
	4 MAGENTA
	5 YELLOW
	6 WHITE
	7
B_BLACK	0 B_BLUE
	100 B_GREEN
	200 B_CYAN
	300 B_RED
	400 B_MAGENTA
	500 B_YELLOW
	600 B_WHITE
	700
BLANK	8 REVERSE
	10 UNDERLN
	20 BLINK
	40 HILIGHT
	80 DIM
	1000

If the cursor position display has been turned on (see sm_c_vis), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead.

VARIANTS AND RELATED FUNCTIONS

```
     sm_er_options (key, discard);
     sm_err_reset (message);
     sm_quiet_err (message);
     sm_qui_msg (message);
```

EXAMPLE

```
sm_emsg ("%MDYou goofed. Press %A0017any%A7 key to continue";
```

NAME

sm_er_options - set error message options

SYNOPSIS

```
#include "smdefs.h"

void sm_er_options (acknowledge_key, flags)
int acknowledge_key;
int flags;
```

DESCRIPTION

This function affects the behavior of the error message display functions: sm_err_reset, sm_emsg, sm_quiet_err, and sm_qui_msg. By default, an error message remains on the display until you acknowledge it by pressing the space bar; this function changes both the key and the requirements for error acknowledgement.

Flags specifies whether the acknowledgement key is to be discarded, and (if so) whether the hit-space window is to be displayed. The following two pairs of flags are recognized:

ER_DISCARD	Default. All error messages must be acknowledged by the acknowledge_key, which is then discarded. Any other key struck before the acknowledgement key will also be discarded.
ER_USE_KEY	Error messages may be acknowledged by any key, which is then treated as ordinary keyboard input; for instance, an invalid input message would be acknowledged by the first character of the corrected input. However, the keyboard typeahead buffer is flushed, so that anything you typed before the message was displayed is still discarded.
ER_YES_WIND	Default. If ER_DISCARD is in effect and another key is hit when the acknowledgement key is expected, a reminder window pops up. (The text in the window is obtained from the message file entries SM_SP1 and SM_SP2.) Both the reminder window and the message disappear when the acknowledgement key is struck.
ER_NO_WIND	If ER_DISCARD is in effect and another key is hit when the acknowledgement key is expected, the terminal beeps.

If neither of one pair is specified, the corresponding option remains unchanged. These options will be overridden by a %MD or %MU string at the beginning of a message.

If you define the SMEROPTIONS variable in your setup file, it will cause this function to be called automatically during start-up with the parameters you specify there.

VARIANTS AND RELATED FUNCTIONS

```
sm_emsg (message);
sm_err_reset (message);
sm_quiet_err (message);
sm_qui_msg (message);
```

EXAMPLE

```
#include "smdefs.h"

/* Reset error message options to their defaults. */

sm_er_options (" ", ER_DISCARD|ER_YES_WIND);
```

NAME

sm_err_reset - display an error message and reset the status line

SYNOPSIS

```
void sm_err_reset (message)
char *message;
```

DESCRIPTION

The message is displayed on the status line until you acknowledge it by pressing a key. If message is too long to fit on the status line, it is displayed in a window instead. The exact behavior of error message acknowledgement is governed by sm_er_options. The initial message attribute is set by sm_ch_emsgatt, and defaults to blinking.

This function turns the cursor on before displaying the message, and forces off the global flag sm_do_not_display. It is similar to sm_emsg, which does not turn on the cursor, and to sm_quiet_err, which inserts a constant string (normally "ERROR:") before the message.

Several percent escapes provide control over the content and presentation of status messages. They are interpreted by sm_d_msg_line, which is eventually called by everything that puts text on the status line (including field status text). The character following the percent sign must be in upper-case; this is to avoid conflict with the percent escapes used by printf and its variants. Certain percent escapes (%W, for instance; see below) must appear at the beginning of the message, i.e. before anything except perhaps another percent escape.

- If a string of the form %Annnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table below gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear immediately after the attribute change, pad the attribute to 4 digits with leading zeroes; if the following character is not a legal hex digit, leading zeroes are unnecessary.
- If a string of the form %KKEYNAME appears anywhere in the message, KEYNAME is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the %K is stripped out and the mnemonic remains. Key mnemonics are defined in smkeys.h ; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.
- If %N appears anywhere in the message, the latter will be presented in a pop-up window rather than on the status line, and all occurrences of %N will be replaced by newlines.
- If the message begins with a %B, JAM will beep the terminal (using sm_bel) before issuing the message.
- If the message begins with %W, it will be presented in a pop-up window instead of on the status line. The window will appear near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it will appear near the top. If the message begins with %MU or %MD, and is passed to one of the error message display functions, JAM will ignore the default error message acknowledgement flag and process (for %MU) or discard (for %MD) the next character typed.

Note that, if a message containing percent escapes - that is, %A, %B, %K, %N or %W - is displayed before sm_initcrt or after %W is called, the percent escapes will show up in it.

Attribute	Hex value
BLACK	0 BLUE
	1 GREEN
	2 CYAN
	3 RED
	4 MAGENTA
	5 YELLOW
	6 WHITE
	7
B_BLACK	0 B_BLUE
	100 B_GREEN
	200 B_CYAN
	300 B_RED
	400 B_MAGENTA
	500 B_YELLOW
	600 B_WHITE
	700
BLANK	8 REVERSE
	10 UNDERLN
	20 BLINK
	40 HILIGHT
	80 DIM
	1000

If the cursor position display has been turned on (see sm_c_vis), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead.

VARIANTS AND RELATED FUNCTIONS

```
     sm_er_options (key, discard);
     sm_emsg (message);
     sm_quiet_err (message);
     sm_qui_msg (message);
```

EXAMPLE

```
#include "smdefs.h"

/* Let somebody know that his name isn't in the database. */

int validate (field, name, occur, bits)
char *name;
{
     char buf[128];

     if (getrec (name) == 0)
     {
          sprintf (buf, "%s is not in the database.", name);
          sm_err_reset (buf);
          return -1;
     }

     return 0;
}
```

NAME

sm_fcase - set case sensitivity when searching for screens

SYNOPSIS

```
     #include "smdefs.h"

     int sm_fcase (case)
     int case;
```

DESCRIPTION

Controls whether the matching of screen names in the form stack, form libraries, and memory resident form list is case-sensitive. Case must be either CASE_SENS or CASE_INSENS; those values are defined in smdefs.h . Refer to sm_r_window and Section 12.1 for descriptions of the searches affected by this function.

RETURNS

-1 if case was invalid ; 0 otherwise.

EXAMPLE

```
#include "smdefs.h"

/* Use case-insensitive search on case-insensitive
 * operating systems. */

#if (defined MSDOS || defined VMS)
     sm_fcase (CASE_INSENS);
#endif
```

NAME

sm_fextension - set default screen file extension

SYNOPSIS

```
int sm_fextension (extension)
char *extension;
```

DESCRIPTION

This function makes extension the default file extension for screen files. When searching for a screen, JAM will append it to any name that does not already contain an extension. Refer to sm_r_window and Section 12.1 for a description of the searches affected by this function, and to the introduction to the Configuration Guide for details on how JAM handles file extensions generally.

Extension should not contain any separator, such as a period. That and the placement of the extension are controlled by the SMUSEEXT setup variable; by default, extensions are placed at the end of the filename and are separated from it by a period.

The same effect may be achieved by defining the SMFEXTENSION variable in your setup file. Refer to the section on setup files in the Configuration Guide.

RETURNS

-1 if insufficient memory is available to store the extension; 0 otherwise.

EXAMPLE

```
/* Change the default extension to "form". */

sm_fextension ("form");

/* Declare that screen files should have no default extension. */

sm_fextension ("");
```

NAME

sm_flush - flush delayed writes to the display

SYNOPSIS

```
void sm_flush ();
```

DESCRIPTION

This function performs delayed writes and flushes all buffered output to the display. It is called automatically whenever the keyboard is opened and there are no keystrokes available, i.e. typed ahead.

Calling this routine indiscriminately can significantly slow execution. As it is called whenever the keyboard is opened, the display is always guaranteed to be in sync before data entry occurs; however, if you want timed output or other non-interactive display, use of this routine will be necessary.

sm_flush does two sorts of flushing: first it does output that has been delayed, then it calls a system-dependent routine that empties display output buffers.

VARIANTS AND RELATED FUNCTIONS

```
sm_rescreen ();
```

EXAMPLE

```
/* Update a system time field once per second, until a key
 * is pressed. */

while (!sm_keyhit (10))
{
     sm_n_putfield ("time_now", "");
     sm_flush ();
}

/* ...process the key */
```

NAME

sm_formlist - update list of memory-resident forms

SYNOPSIS

```
     #include "smdefs.h"

     int sm_formlist (ptr_to_form_list)
     struct form_list *ptr_to_form_list;
```

DESCRIPTION

This function adds to a list of memory-resident forms. Each member of the list is a structure giving the name of the form, as a character string, and its address in memory.

The library functions sm_r_form, sm_r_window, and sm_r_at_cur, which are all called with a screen name as parameter, search for it in the memory-resident form list before attempting to read the screen from disk.

This function is called once from sm_initcrt to pick up the global list sm_memforms; this is for compatibility with Release 3. It can be called any number of times from an application program to add forms to the list.

Since no count is given with the list, care must be taken to end the new list with a null entry.

RETURNS

-1 if insufficient memory is available for the new list; 0 otherwise.

EXAMPLE

```
#include "smdefs.h"

/* The following code adds two screens to the
 * memory-resident form list. */

struct form_list new_list[] =
{
     "new_form1",  new_form1,
     "new_form2",  new_form2,
     "",        0
};

sm_formlist (new_list);
```

NAME

sm_fptr - get the contents of a field

SYNOPSIS

```
#include "smdefs.h"

char *sm_fptr (field_number)
int field_number;
```

DESCRIPTION

Returns the contents of the field specified by field_number. Leading blanks in right-justified fields, and trailing blanks in left-justified fields, are stripped.

This function shares with several others a pool of buffers where it stores returned data. The value returned by any of them should therefore be processed quickly or copied. sm_getfield is not subject to this restriction.

RETURNS

The field contents, or 0 if the field cannot be found.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_fptr (field_name, element);
sm_i_fptr (field_name, occurrence);
sm_n_fptr (field_name);
sm_o_fptr (field_number, occurrence);
sm_getfield (buffer, field_number);
sm_putfield (field_number, text);
```

EXAMPLE

```
#include "smdefs.h"

/* Little function to tell somebody something s/he
 * already knows. */

void report (fieldname)
char *fieldname;
{
     char buf[256], *stuf;
     if ((stuf = sm_n_fptr (fieldname)) == 0)
          return;

     sprintf (buf, "You have typed %s in the %s field.",
          stuf, fieldname);
     sm_emsg (buf);
}
```

NAME

sm_fval - force field validation

SYNOPSIS

```
int sm_fval (field_number)
int field_number;
```

DESCRIPTION

This function performs all validations on the indicated field or occurrence, and returns the result. If the field is protected against validation, the checks are not performed and the function returns 0; see sm_1protect. Validations are done in the order listed below. Some will be skipped if the field is empty, or if its VALIDED bit is already set (implying that it has already passed validation).

Validation	Skip if valid	Skip if empty
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check-digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n
exit function	n	n
jpl function	n	n

*
The currency format edit contains a skip-if-empty flag; see the Author's Guide.

If you need to force a skip-if-empty validation, make the field required. A digits-only field must contain at least one digit in order to be considered nonempty; for other character edits, any nonblank character makes the field nonempty.

Math expressions and field exit functions are never skipped, since they can alter fields other than the one being validated. If you are planning to use this function, be careful to make no assumptions about the cursor position when writing field exit functions. Often it is assumed that the cursor is in the field being validated, so that sm_tab and similar functions will work properly.

Field validation is performed automatically within sm_openkeybd when the cursor exits a field. Application programs need call this function only to force validation of other fields.

RETURNS

-2 if the field or occurrence specification is invalid; -1 if the field fails any validation; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_fval (array_name, element);
sm_i_fval (field_name, occurrence);
sm_n_fval (field_name);
sm_o_fval (field_number, occurrence);
sm_s_val ();
```

EXAMPLE

```
#include "smdefs.h"

/* Make sure that the previous field has been validated
* before checking the current one. */

validate (fieldnum, data, occurrence, bits)
char *data;
{
     if (sm_fval (fieldnum - 1))
     {
          /* Place cursor in the previous field and indicate error */
          sm_gofield (fieldnum - 1);
          return 1;
     }
     ...
}
```

NAME

sm_getcurno - get current field number

SYNOPSIS

```
int sm_getcurno ();
```

DESCRIPTION

Returns the number of the field in which the cursor is currently positioned. The field number ranges from 1 to the total number of fields in the screen.

RETURNS

Number of the current field, or 0 if the cursor is not within a field.

VARIANTS AND RELATED FUNCTIONS

```
sm_occur_no ();
```

EXAMPLE

```
/* Imagine that the screen contains an 8 by 8 array
 * of fields, like a checkerboard. The following code
 * gets the number of the current field and returns
 * the corresponding row and column. */

void get_location (row, column)
int *row, *column;
{
     int fieldnum;

     if ((fieldnum = sm_getcurno ()) == 0)
          *row = *column = -1;
     else
     {
          *row = (fieldnum - 1) / 8 + 1;
          *column = (fieldnum - 1) % 8 + 1;
     }
}
```

NAME

sm_getfield - copy the contents of a field

SYNOPSIS

```
int sm_getfield (buffer, field_number)
char *buffer;
int field_number;
```

DESCRIPTION

Copies the data found in field_number to buffer. Leading blanks in right-justified fields, and trailing blanks in left-justified fields, are not copied.

Responsibility for providing a buffer large enough for the field's contents rests with the calling program. This should be at least one greater than the maximum length of the field, taking shifting into account.

Note that the order of arguments to this function is different from that to the related function sm_putfield.

RETURNS

The total length of the field's contents, or -1 if the field cannot be found.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_getfield (buffer, field_name, element);
sm_i_getfield (buffer, field_name, occurrence);
sm_n_getfield (buffer, field_name);
sm_o_getfield (buffer, field_number, occurrence);
sm_fptr (field_number);
sm_putfield (field_number, text);
```

EXAMPLE

```
#include "smdefs.h"

/* Save the contents of the "rank" field in a buffer
 * of the proper size. */

int size;
char *save_rank;

size = sm_n_length ("rank");
if ((save_rank = malloc (size + 1)) == 0)
     punt ("Malloc error");

if (sm_n_getfield (save_rank, "rank") > size)
     punt ("Bug in sm_length or sm_getfield!");
```

NAME

sm_getkey - get translated value of the key hit

SYNOPSIS

```
#include "smkeys.h"

int sm_getkey ();
```

DESCRIPTION

Gets and interprets keyboard input according to an algorithm described elsewhere, and returns the interpreted value to the calling program. Normal characters are returned unchanged; function keys are interpreted according to a key translation file for the particular computer or terminal you are using.

Function keys include TRANSMIT, EXIT, HELP, LOCAL PRINT, arrows, data modification keys like INSERT and DELETE CHAR, user function keys PF1 through PF24, shifted function keys SPF1 through SPF24, and others. Defined values for all are in smkeys.h . A few function keys, such as LOCAL PRINT and RESCREEN, are processed locally in sm_getkey and not returned to the caller.

There is another function called sm_ungetkey, which pushes logical key values back on the input stream for retrieval by sm_getkey. Since all JAM input routines call sm_getkey, you can use it to generate any input sequence automatically. When you use it, calls to sm_getkey will not cause the display to be flushed, as they do when keys are read from the keyboard.

There is a key-change hook in sm_getkey. Before returning a translated key to its caller, it passes the key to a user-installed function which may alter the key value, delete it from the input stream, or whatever. See sm_u_keychange and sm_install.

There is another pair of hooks, for recording and playing back sequences of keys. These are a recording function, which is passed the key just typed, and a playback function, which is called to obtain a key. See sm_u_record and sm_u_play.

Finally, there is a mechanism for detecting an externally established abort condition, essentially a flag, which causes JAM input functions to return to their callers immediately. The present function checks for that condition on each iteration, and returns the ABORT key if it is true. See sm_isabort.

Application programmers should be aware that JAM control strings are not executed within this function, but at a higher level within the JAM run-time system. If you call this function, do not expect function key control strings to work.

RETURNS

The standard ASCII value of a displayable key; a value greater than 255 (FF hex) for a key sequence in the key translation file.

VARIANTS AND RELATED FUNCTIONS

```
sm_ungetkey (key);
sm_u_keychange (key);
sm_keyfilter (flag);
sm_u_play ();
sm_u_record (key);
sm_isabort (flag);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Alternate version of sm_query_msg, which makes up
 * its mind right away. */

int query (text)
char *text;
{
     int key;

     sm_d_msg_line (text, REVERSE);
     for (;;)
     {
          switch (key = sm_getkey ())
          {
          case XMIT:
          case 'y':
          case 'Y':
               sm_d_msg_line ("", WHITE");
               return 1;
          default:
               sm_emsg ("%MU I take that for a 'no'");
               sm_d_msg_line ("", WHITE");
               return 0;
          }
     }
}
```

PSEUDOCODE

The multiplicity of hooks in sm_getkey makes it a little difficult to see how they interact, which take precedence, and so forth. In an effort to clarify the process, we present an outline of sm_getkey. The process of key translation is deliberately omitted, for the sake of clarity; that algorithm is presented separately, toward the end of this chapter.

```
-- Step 1

If an abort condition exists,
     return the ABORT key.

If there is a key pushed back by ungetkey,
     return that.

If playback is active and a key is available,
     take it directly to Step 2; otherwise read and translate input from the
     keyboard.


-- Step 2

Pass the key to the keychange function. If that function says to discard the
key, go back to Step 1; otherwise if an abort condition exists,
     return the ABORT key.

If recording is active,
     pass the key to the recording function.


-- Step 3

If the routing table says the key is to be processed locally,
     do so.

If the routing table says to return the key,
     return it; otherwise, go back to Step 1.
```

NAME

sm_gofield - move the cursor into a field

SYNOPSIS

```
     int sm_gofield (field_number)
     int field_number;
```

DESCRIPTION

Positions the cursor to the first enterable position of field_number. If the field is shiftable, it is reset.

In a right-justified field, the cursor is placed in the rightmost position; in a left-justified field, in the leftmost. In either case, if the field is digits-only, the cursor goes to the nearest position not occupied by a punctuation character. Use sm_off_gofield to place the cursor elsewhere than in the first position.

When called to position the cursor in a scrollable field or array, sm_o_gofield and sm_i_gofield return an error if the occurrence number passed exceeds by more than 1 the number of items in the specified field or array.

RETURNS

-1 if the field is not found; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_e_gofield (field_name, element)
     sm_i_gofield (field_name, occurrence)
     sm_n_gofield (field_name)
     sm_o_gofield (field_number, occurrence)
     sm_getcurno ();
     sm_off_gofield (field_number, offset);
```

EXAMPLE

```
#include "smdefs.h"

/* If the combination of this field and the previous one
 * is invalid, go back to the previous for data entry. */

int validate (field, data, occur, bits)
char *data;
{
     if (bits & VALIDED)
          return 0;

     if (!lookup (data, sm_fptr (field - 1)))
     {
          sm_novalbit (field - 1);
          sm_gofield (field - 1);
          sm_quiet_err ("Lookup failed - please re-enter both items.");
          return 1;
     }
     return 0;
}
```

NAME

    sm_hlp_by_name - present help window

SYNOPSIS

```
     int sm_hlp_by_name (help_screen)
     char *help_screen;
```

DESCRIPTION

The named screen is displayed and processed as a normal help screen, including input processing for the current field (if any). Four types of help screens are automatically recognized and processed:

1. A help screen with one unprotected field, which has no attached function. This function will change the help field's length to that of the underlying field, if necessary by making the help field shiftable; and it will copy the underlying field's contents and most of its edits into the help field. The operator may then change the help field's contents, and, by hitting TRANSMIT, have the new contents copied back to the underlying field.
2. A help screen with two or more unprotected fields, all of which are menu fields. The screen will be treated as a menu, except that menu entries will be used only to bring up lower level help screens.
3. A help screen with entry-protected fields only. The screen is displayed until the operator hits either EXIT or TRANSMIT. Hitting the HELP key while the cursor is in a field will bring up any help screen associated with that field.
4. A help screen with display data only. The screen is displayed until the operator hits any returnable key (excluding such keys as LOCAL PRINT, RESCREEN, or HELP), at which point the window is closed.

Refer to the Author's Guide for further instructions on how to create each kind of help screen.

RETURNS

    -1 if screen is not found or other error; 1 if data copied from help screen to underlying field; 0 otherwise.

EXAMPLE

```
#include "smdefs.h"

/* If user tabs out of empty field, find the field's
 * help screen and execute it. Implemented as a validation
 * function. */

nonempty (field, data, occur, bits)
char *data;
{
     char *helpscreen;

     if (*data == 0)
     {
          if ((helpscreen = sm_edit_ptr (field, HELP)) != 0 ||
               (helpscreen = sm_edit_ptr (field, HARDHELP)) != 0)
               sm_hlp_by_name (helpscreen + 2);
     }

     return 0;
}
```

NAME

     sm_home - home the cursor

SYNOPSIS

```
     int sm_home ();
```

DESCRIPTION

This routine moves the cursor to the first enterable position of the first tab-unprotected field on the screen; or, if the form has no tab-unprotected fields, to the first line and column of the topmost window.

The cursor will be put into a tab-protected field if it occupies the first line and column of the window and there are no tab-unprotected fields.

RETURNS

     The number of field in which the cursor is left, or 0 if the form has no
          unprotected fields and the home position is not in a protected field.

VARIANTS AND RELATED FUNCTIONS

```
     sm_gofield (field_number);
     sm_last ();
```

EXAMPLE

```
#include "smdefs.h"

/* Suppose that at some point the data entry process
 * has gotten fouled up beyond all repair. The following
 * code fragment could be used to start it over. */

/* ... */
sm_cl_unprot ();
sm_home ();
sm_err_reset ("%MUI'm confused! Let's start over.");
/* ... */
```

NAME

sm_i_achg change the display attribute of a scrolling item
sm_i_amt_format
format data and write to occurrence sm_i_bitop
manipulate edit bits of an occurrence sm_i_dblval
get decimal value of occurrence sm_i_dlength
get length of data in occurrence sm_i_doccur
delete an occurrence sm_i_dtofield
write decimal value to occurrence sm_i_fptr
get copy of occurrence's data sm_i_fval
force validation of occurrence sm_i_getfield
copy data from occurrence into buffer sm_i_gofield
position cursor to occurrence sm_i_intval
get integer value of occurrence sm_i_ioccur
insert a blank occurrence into a scroll or array
sm_i_itofield
write integer to occurrence sm_i_lngval
get long integer value of occurrence sm_i_ltofield
write long integer to occurrence sm_i_mdt_clear
reset MDT bit of an occurrence sm_i_mod_test
test MDT bit of an occurrence sm_i_novalbit
reset validated bit of an occurrence sm_i_off_gofield
place cursor in the middle of an occurrence
sm_i_putfield
write data string to occurrence

DESCRIPTION

Each of the above functions refers to data by field name and occurrence number. As used in the above functions, occurrence means

1. item, if the field or array is scrollable;
2. element, if the specified field is part of a non-scrollable array; or
3. the specified field, if neither scrollable nor an array.

If occurrence is zero, the reference is always to the current contents of the named field, or of the base field of the named array.

For the description of a particular function, look under the related function without i_ in its name. For example, sm_i_amt_format is described under sm_amt_format.

If the named field is not on the screen, these functions will attempt to retrieve or change its value in the local data block.

NAME

sm_inbusiness - tell whether the screen manager is up and running or not

SYNOPSIS

```
int sm_inbusiness ();
```

DESCRIPTION

This function inspects internal screen manager flags and data structures to determine whether screen manager I/O is in progress, and returns a nonzero value if it is. The library functions sm_initcrt and sm_return turn on that state; sm_resetcrt and sm_leave turn it off.

Applications may find this function useful in deciding how to display error messages, or how to handle error conditions generally.

RETURNS

1 if the screen manager is running, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_initcrt ();
sm_resetcrt ();
sm_leave ();
sm_return ();
```

EXAMPLE

```
#define ERRMSG1 "Insufficient memory available"
char *buf;

if ((buf = malloc (4096)) == 0)
{
     if (sm_inbusiness ())
          sm_quiet_err (ERRMSG1);
     else fprintf (stderr, "%s\n", ERRMSG1);
}
```

NAME

sm_ind_set - control onscreen shifting and scrolling indicators

SYNOPSIS

```
int sm_ind_set (flag)
int flag;
```

DESCRIPTION

This function controls the presence and style of shifting and scrolling indicators, which JAM uses to indicate the presence of offscreen data in a field or array. Flag is restricted to the following values, which are defined in smdefs.h . The where codes are to be ored with the whether codes:

Whether to display indicators

IND_NONE	no indicators
IND_SHIFT	shifting indicators only
IND_SCROLL	scrolling indicators only
IND_BOTH	shift and scroll

Where to display scrolling indicators

IND_FULL	full width of field
IND_FLDLEFT	left corner of field
IND_FLDCENTER	center of field
IND_FLDRIGHT	right side of field
IND_FLDENTRY	left or right, according to the field's justification

If flag is IND_NONE, the existing indicators are erased from the display and the indicator table is discarded; otherwise, an indicator table is allocated if needed, and the new indicators are displayed. The default setting is IND_BOTH and IND_FULL.

This function should be called before reading the screen for which indicators are desired; normally, it is called once at the beginning of the program. Closing a window does not perform a recalculation, but restores the underlying screen's indicators; to avoid confusion, this function should be called when no windows are displayed.

If you define the SMINDSET variable in your setup file, it will cause this function to be called automatically during start-up with the parameters you specify there.

RETURNS

-1 if sufficient memory for a new table was not available; 0 otherwise.

EXAMPLE

```
#include "smdefs.h"
/* Set indicator display back to defaults */
sm_ind_set (IND_BOTH | IND_FULL);
```

NAME

sm_ininames - record names of initial data files for local data block

SYNOPSIS

```
int sm_ininames (name_list)
char *name_list;
```

DESCRIPTION

Sets up a list of initialization files for local data block items. The file names in the single string name_list should be separated by commas, semicolons or blanks; there may be up to ten of them. The files themselves contain pairs of names and values, which are used to initialize local data block items by sm_ldb_init, q.v. That function is called automatically during JAM initialization, so sm_ininames should be called before then. White space in the initialization files is ignored, but we suggest a format like the following:

```
"emperor"           "Julius Caesar" "lieutenant"
                    "Mark Antony" "assassin[1]"
                    "Brutus" "assassin[2]"
                    "Cassius"
```

You may achieve the same effect by defining the SMININAMES variable in your setup file to the list of names. See the section on setup files in the Configuration Guide for details.

Items of all scopes may be freely mixed within all files. We recommend, however, that items be grouped in files by scope if you are planning to use sm_lreset, q.v. That function clears all items of a given scope before reinitializing them from a single file.

RETURNS

-5 if insufficient memory is available to store the names; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_lreset (scope, filename);
sm_ldb_init ();
```

EXAMPLE

```
/* Set up five initialization files. */

sm_ininames ("scope0.ini, scope1.ini, scope2.ini,\
scope3.ini, scope4.ini");
```

NAME

sm_initcrt - initialize the display and JAM data structures

SYNOPSIS

```
void sm_initcrt (path)
char *path;
```

DESCRIPTION

This function must be called at the beginning of screen handling, that is, before any screens are displayed or the keyboard opened for input to a JAM screen. Functions that set options, such as sm_ok_options, and those that install functions or configuration files, such as sm_install or sm_vinit, are the only kind that may be called before sm_initcrt.

Path is a directory to be searched for screen files by sm_r_window and variants. First the file is sought in the current directory; if it is not there, it is sought in the path supplied to this function. If it is not there either, the paths specified in the environment variable SMPATH (if any) are tried. The path argument must be supplied. If all forms are in the current directory, or if (as JYACC suggests) all the relevant paths are specified in SMPATH, an empty string may be passed. After setting up the search path, sm_initcrt performs several initializations:

1. It calls a user-defined initialization routine (see sm_u_uinit).
2. It determines the terminal type, if possible by examining the environment (TERM or SMTERM), otherwise by asking you.
3. It executes the setup files defined by the environment variables SMVAGS and SMSETUP, and reads in the binary configuration files (message, key, and video) specific to the terminal.
4. It allocates memory for a number of data structures shared among JAM library functions.
5. If supported by the operating system, keyboard interrupts are trapped to a routine that clears the screen and exits.
6. It initializes the operating system screen and keyboard channels, and clears the display.

This function is called automatically during JAM start-up, and should not be called by application programs.

VARIANTS AND RELATED FUNCTIONS

```
sm_smsetup ();
sm_uinit (term_type);
sm_msgread (prefix, range, address, name);
sm_vinit (video_file);
sm_keyinit (key_file);
```

EXAMPLE

```
/* To initialize the screen manager without supplying a path
* for screens: */

     sm_initcrt ("");
```

NAME

sm_install - attach application functions to JAM library hooks

SYNOPSIS

```
#include "smdefs.h"

struct fnc_data *sm_install (which_hook,
     what_funcs, howmany)
int which_hook;
struct fnc_data what_funcs[];
int *howmany;
```

DESCRIPTION

This function places an application routine on one of the screen manager library hooks; this enables JAM to pass control to your code in the proper context. Each hook is documented separately in this chapter; refer to the table below.

Which_hook must be drawn from the following list. It identifies the hook your routine is to be attached to.

Which	Purpose	Refer to
UINIT_FUNC	initialization	sm_u_uinit
URESET_FUNC	exit-time cleanup	sm_u_ureset
VPROC_FUNC	video processing	sm_u_vproc
CKDIGIT_FUNC	check digit computation	sm_u_ckdigit
KEYCHG_FUNC	translated key peek/poke	sm_u_keychg
INSCRSR_FUNC	insert/overwrite toggle	sm_u_inscrsr
PLAY_FUNC	play back saved keys	sm_u_play
RECORD_FUNC	record keys for playback	sm_u_record
AVAIL_FUNC	check for recorded keys	sm_u_avail
ASYNC_FUNC	asynchronous function	sm_u_async
STAT_FUNC	status line function	sm_u_statfnc
ATTCH_FUNC	field attached function list	Section 1.3
FORM_FUNC	screen entry/exit list	Section 1.4
CARET_FUNC	invoked function list	Section 1.2

The last three hooks mentioned are not for single functions, but for lists of functions. When you install these, names are required in the fnc_data structures; what_funcs should be an array of structures; and howmany should hold the number of functions in the list. Your functions are added to any already in the list.

The second parameter, what_funcs, is the address of a structure describing an application routine. If which_hook is a list, it is the address of an array of such structures. Here is a definition of the structure:

```
struct fnc_data {
     char *fnc_name;
     int (*fnc_addr)();
     char language;
     char intrn_use;
     char appl_use;
     char reserved;
};
```

- Fnc_name is a character string naming your routine. It is required only if which_hook is ATTCH_FUNC, CARET_FUNC, or FORM_FUNC, and should match the function name used in your screens.
- Fnc_addr is always required. It is the address of your routine.
- Language is a language identifier, drawn from smdefs.h . C is always 0.
- Intrn_use serves as an installation parameter. Currently it is used with ASYNC_FUNC and the lists; see below.
- Appl_use is reserved for your own use.
- reserved is reserved for future use by JYACC.

The third parameter, howmany, is required only if which_hook is a list. It is the address of an integer variable giving the number of entries in your list. The count is passed by reference so that sm_install can return a count for the new list with its return value. If the value pointed to by howmany is zero, all functions in the list are removed, except those having a non-zero value in the intrn_use field of the structure. Built-in functions supplied with JAM are protected from removal in this fashion.

When you are installing an asynchronous function using ASYNC_FUNC, the intrn_use field of the structure should be set to the timeout, in tenths of a second. While the keyboard is idle, the asynchronous function will be called at that interval.

RETURNS

The address of the old function data structure(s), or zero if no function was previously installed. For lists of functions, also places the number of entries in the new list in howmany.

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/************* Example 1 *************/

/* Here is a function to change the RETURN key to TAB, and
 * chain to the old keychange function, which is stored in
 * prev_fix.
 */
static struct fnc_data *prev_fix;

int keyfix (key)
{
     int k = key == NL ? TAB : key;
     if (prev_fix)
          return (*(prev_fix->fnc_addr)) (k);
     else return k;
}

/* Install the new function, storing the old one in prev_fix.
 * You must take the address of the fnc_data structure.
 */

static struct fnc_data fix = {
     0, keyfix, 0, 0, 0, 0
};

prev_fix = sm_install (KEYCHG_FUNC, &fix, (int *)0);

/************* Example 2 *************/

/* Install two attached functions, defined elsewhere.
 */

extern int atch1(), atch2();

static struct fnc_data atch[] = {
     { "atch1", atch1, 0, 0, 0, 0 },
     { "atch2", atch2, 0, 0, 0, 0 },
};
int count;

count = sizeof(atch) / sizeof(struct fnc_data);
sm_install (ATTCH_FUNC, atch, &count);
```

NAME

sm_intval - get the integer value of a field

SYNOPSIS

```
int sm_intval (field_number)
int field_number;
```

DESCRIPTION

Returns the integer value of the data contained in the field specified by field_number. Any punctuation characters in the field (except, of course, a leading plus or minus sign) are ignored.

RETURNS

The integer value of the specified field, or zero if the field is not found.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_intval (field_name, element);
sm_i_intval (field_name, occurrence);
sm_n_intval (field_name);
sm_o_intval (field_number, occurrence);
sm_itofield (field_number, value);
```

EXAMPLE

```
/* Retrieve the integer value of the "sequence" field. */

int sequence;

sequence = sm_n_intval ("sequence");
```

NAME

sm_ioccur - insert blank occurrences

SYNOPSIS

```
int sm_o_ioccur (field_number, occurrence, count)
int field_number;
int occurrence;
int count;
```

DESCRIPTION

Inserts count blank data items before the specified occurrence, moving that occurrence and all following occurrences down. If inserting that many would move some occurrence past the end of its array or scrolling array, fewer will be inserted. This function never increases the maximum number of items a scroll can contain; sm_sc_max does that. If count is negative, occurrences will be deleted instead, subject to limitations described in the page for sm_doccur.

If occurrence is zero, the occurrence used is that of field_number. If occurrence is nonzero, however, it is taken relative to the first field of the array in which field_number occurs.

This function is normally bound to the INSERT LINE key. It has only two variants, sm_i_ioccur and sm_o_ioccur; the other three, including sm_ioccur itself, do not exist.

RETURNS

-1 if the field or occurrence number is out of range; -3 if insufficient memory is available; otherwise, the number of occurrences actually inserted (zero or more).

VARIANTS AND RELATED FUNCTIONS

```
sm_i_ioccur(field_name, occurrence, count);
sm_i_doccur(field_name, occurrence, count);
sm_o_doccur(field_number, occurrence, count);
```

EXAMPLE

```
#include "smkeys.h"

/* As a shortcut, make the PF5 key insert five blank
 * lines in the "amounts" array. */

int field, key;

while ((key = sm_openkeybd ()) != EXIT)
{
     if (key == PF5)
     {
          /* Make sure we're in the right place */
          field = sm_base_fldno (sm_getcurno ());
          if (field == sm_n_fldno ("amounts"))
               sm_o_ioccur (field, 0, 5);
     }
     ...
}
```

NAME

sm_is_yes - boolean value of a yes/no field

SYNOPSIS

```
int sm_is_yes (field_number)
int field_number;
```

DESCRIPTION

The first character of the specified field_number is compared with the first letter of the SM_YES entry in the message file, ignoring case, and the resulting logical value is returned.

This function is ordinarily used with one-letter fields restricted to yes or no by the appropriate character edit. Unlike the field data retrieval functions (sm_fptr, etc.), it does not ignore leading blanks.

RETURNS

1 if the field is found, and its contents match as described above; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_is_yes (field_name);
sm_e_is_yes (field_name, element);
sm_i_is_yes (field_name, occurrence);
sm_o_is_yes (field_number, occurrence);
```

EXAMPLE

```
/* Keep processing until the user enters "n" in
 * a flag field. This is an alternative for the
 *& usual checking against the EXIT key. */

while (sm_n_is_yes ("continue"))
{
     sm_openkeybd ();
}
```

NAME

sm_isabort - test and set the abort control flag

SYNOPSIS

```
#include "smdefs.h"

int sm_isabort (flag)
int flag;
```

DESCRIPTION

This function sets the abort flag to the value of flag, and returns the old value. Flag must be one of the following:

Flag	Meaning
ABT_ON	set abort flag ABT_OFF clear abort flag ABT_DISABLE turn abort reporting off ABT_NOCHANGE do not alter the flag

Abort reporting is intended to provide a quick way out of processing in the JAM library, which may involve nested calls to sm_openkeybd and other input functions. The triggering event is the detection of an abort condition by sm_getkey, either an ABORT keystroke or a call to this function with ABT_ON.

This function enables application code to verify the existence of an abort condition by testing the flag, as well as to establish one. You may need to verify it because certain functions, such as sm_choice, cannot return the ABORT key directly. Abort processing is described in detail later in this chapter.

RETURNS

The previous value of the abort flag.

EXAMPLE

```
#include "smdefs.h"

/* Establish an abort condition */

sm_isabort (ABT_ON);

/* Verify that an abort condition exists, without
 * altering it. */

if (sm_isabort (ABT_NOCHANGE) == ABT_ON)
     ...
```

NAME

sm_itofield - write an integer value to a field

SYNOPSIS

```
int sm_itofield (field_number, value)
int field_number, value;
```

DESCRIPTION

The integer passed to this routine is converted to characters and placed in the specified field. A number longer than the field will be truncated (on the left or right, according to the field's justification) without warning.

RETURNS

-1 if the field is not found; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_itofield (field_name, element, value);
sm_i_itofield (field_name, occurrence, value);
sm_n_itofield (field_name, value);
sm_o_itofield (field_number, occurence, value);
sm_intval (field_number);
```

EXAMPLE

```
/* Find the length of the data in field number 12, and
 * tell somebody about it. */

sm_n_itofield ("count", sm_dlength (12));
```

NAME

sm_jclose - close current window under JAM control

SYNOPSIS

```
int sm_jclose ();
```

} DESCRIPTION

The currently open window is erased, and the screen is restored to the state before the window was opened. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the position it had before the window was opened.

Note that this function closes a window regardless of whether it was opened via a control string, or sm_jwindow. If the last window was opened through a call to sm_r_window the results are unpredictable.

RETURNS

-1 is returned if there is no window open, i.e. if the currently displayed screen is a form (or if there is no screen up). 0 is returned otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_jwindow (screen_arg);
```

EXAMPLE

```
/* This is an example of a caret function attached to the  XMIT */
/* key. It validates login and password information. If the  */
/* login and password are incorrect, the program proceeds to */
/* close three of the four "security" windows used for getting*/
/* a user's login and password information, and the user may */
/* again attempt to enter the information. If the password */
/* passes, the welcome screen is displayed, and the user may */
/* proceed.

int complete_login(jptr);
char *jptr;
{
     char pass[10];
     sm_n_getfield("pass", "password");
     if(!check_password(pass)) /*call routine to validate password*/
     {
          sm_jclose();      /*close current password window*/
          sm_jclose();      /*close 3rd underlying login window*/
          sm_jclose();      /*close 2nd underlying login window*/
          sm_emsg("Please reenter login and password");
     }                                    /*in bottom window*/
     else
     {
          sm_d_msg_line("Welcome to Security Systems, Inc.");
          sm_jform("Welcome");
                           /*open welcome screen*/
     }
     return (0);
}
```

NAME

sm_jform - display a screen as a form under JAM control

SYNOPSIS

```
int sm_jform (screen_name)
char *screen_name;
```

DESCRIPTION

This function displays the named screen as a base form. The form's opening and closing (with the EXIT key) are under JAM control. The function is similar to sm_r_form.

Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new form is displayed with its upper left-hand corner at the extreme upper left of the screen.

If the form contains display data that are too big for the physical display, they are truncated without warning. However, if there are fields that won't fit within the physical display, this function returns an error without displaying the form.

The named form is sought on disk in the current directory; then under the path supplied to sm_initcrt; then in all the paths in the setup variable SMPATH. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

In the case of a return of -1, -2 or -7 the previously displayed form is still displayed and may be used. Other negative return code indicate that the display is undefined; the caller should display another form before using screen manager functions. The return code -2 typically means that the named screen does not exist; however, it may occur because the maximum allowable number of files is already open.

RETURNS

0 if no error occurred; -1 if the screen file's format is incorrect; -2 if the form cannot be found; -4 if, after the screen has been cleared, the form cannot be successfully displayed because of a read error; -5 if, after the screen was cleared, the system ran out of memory; -7 if the screen was larger than the physical display, and there were fields that would have fallen outside the display.

VARIANTS AND RELATED FUNCTIONS

```
sm_r_form (screen_name);
sm_jwindow (name, line, column);
```

EXAMPLE

```
/* This exemplifies a caret function attached to the XMIT key. */
/* Here we have completed entering data on the second of several*/
/* security screens. If the user entered "bypass" into the login, */
/* he bypasses the other security screens, and the "welcome" */
/* screen is displayed. If the user login is incorrect, the */
/* current window is closed, and the user is back at the */
/* initial screen (below). Otherwise, the next security window */
/* is displayed. */


int getlogin(jptr)
char *jptr;
{
     char password[10];
     sm_n_getfield(password, "password");
     /* check if "bypass" has been entered into login */
     if (strcmp(password,"bypass"))
          sm_jform("welcome");
     /* check if login is valid */
     else if (check_password(password))
     {
          sm_jclose();
                   /*close current (2nd) login window */
          sm_emsg("Please reenter login");
     }
     else
          sm_jwindow("login3");
     return (0);
}


}
```

NAME

sm_jwindow - display a window at a given position under JAM control

SYNOPSIS

```
int sm_jwindow (screen_arg)
char *screen_arg;
```

DESCRIPTION

Displays screen_arg with its upper left-hand corner at the current cursor position, if no line and column are specified. The function's argument can include a specification of the display position (for instance, ("wind1 10 20"). The line and column are counted from one. The display takes place under JAM control. JAM can also close the window through a call to sm_jclose.

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the topmost, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

RETURNS

0 if no error occurred during display of the screen; -1 if the screen file's format is incorrect; -2 if the form cannot be found; -3 if the system ran out of memory but the previous screen was restored; -7 if the screen was larger than the physical display, and there were fields that would have fallen outside the display.

VARIANTS AND RELATED FUNCTIONS

```
sm_r_window (name);
sm_jform (name);
sm_jclose ();
```

EXAMPLE

```
/* This is an example of a caret function attached to the XMIT key.*/
/* Here we have completed entering data on the second of several */
/* security screens. If the user entered "bypass" into the login, */
/* he bypasses the other security screens, and the "welcome" */
/* screen is displayed. If the user login is incorrect, the */
/* current window is closed, and the user is back at the */
/* initial screen (below). Otherwise, the next security window */
/* is displayed. */

int getlogin(jptr)
char *jptr;
{
     char password[10];
     sm_n_getfield(password, "password");
     /* check if "bypass" has been entered into login */
     if (strcmp(password,"bypass"))
          sm_jform("welcome");
     /* check if login is valid */
     else if (check_password(password))
     {
          sm_jclose();
                   /*close current (2nd) login window */
          sm_emsg("Please reenter login");
     }
     else
          sm_jwindow("login3");
     return (0);
}
```

NAME

sm_keyfilter - control keystroke record/playback filtering

SYNOPSIS

```
int sm_keyfilter (flag)
int flag;
```

DESCRIPTION

This function turns the keystroke record/playback mechanism of sm_getkey on (flag = 1) or off (flag = 0). If none of the recording hooks have functions on them, turning them on has no effect.

It returns a flag indicating whether recording was previously on or off.

RETURNS

The previous value of the filter flag.

VARIANTS AND RELATED FUNCTIONS

```
sm_getkey ();
sm_u_avail (interval);
sm_u_play ();
sm_u_record (key);
```

EXAMPLE

```
/* Disable key recording and playback. */

sm_keyfilter (0);
```

NAME

sm_keyhit - test whether a key has been typed ahead

SYNOPSIS

```
int sm_keyhit (interval)
int interval;
```

DESCRIPTION

This routine checks whether a key has already been hit; if so, it returns 1 immediately. If not, it waits for the indicated interval and checks again. The key (if any is struck) is not read in, and is available to the usual keyboard input routines.

Interval is in tenths of seconds; the exact length of the wait depends on the granularity of the system clock, and is hardware- and operating-system dependent. JAM uses this function to decide when to call the user-supplied asynchronous function.

If the operating system does not support reads with timeout, this function ignores the interval and only returns 1 if a key has been typed ahead.

RETURNS

1 if a key is available, 0 if not.

VARIANTS AND RELATED FUNCTIONS

```
sm_getkey ();
sm_u_async ();
sm_u_avail (interval);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* The following code adds one asterisk per second to a
 * danger bar, until somebody presses EXIT. */

static char *danger_bar = "*************************";
int k;

sm_d_msg_line ("You have 25 seconds to find the EXIT key.", WHITE);
sm_do_region (5, 10, 25, WHITE, "");  /* Clear the danger bar area */
sm_flush ();

for (k = 1; k <= 25; ++k)
{
     if (sm_keyhit (10))
     {
          if (sm_getkey () == EXIT)
               break;
     }
     sm_do_region (5, 10, k, WHITE, danger_bar);
     sm_flush ();
}
if (k <= 25)
     sm_d_msg_line ("%BCongratulations! you win!");
else sm_err_reset ("Sorry, you lose.");
```

NAME

     sm_keyinit - initialize key translation table

SYNOPSIS

```
     int sm_keyinit (key_file)
     char key_file[];
```

DESCRIPTION

This routine is called by sm_initcrt as part of the initialization process, but it can also be called by an application program (either before or after sm_initcrt) to install a memory-resident key translation file.

To install a memory-resident key translation file, key_file must contain the address of a key translation table created using the JYACC key2bin and bin2c utilities, q.v. If it is zero, a disk file whose name is obtained from the SMKEY variable will be used instead.

If no memory-resident file is supplied and an error occurs while reading the disk file, this function prints out an error message and exits to the operating system.

RETURNS

     0 if the key file is successfully installed; -1 if a disk file is requested and the terminal type is unknown; program exit if an error occurs while reading a disk file.

EXAMPLE

```
/* Install a memory-resident key file */

extern char keyfile[];

sm_keyinit (keyfile);
```

NAME

sm_keylabel - get the printable name of a logical key

SYNOPSIS

```
#include "smkeys.h"

char *sm_keylabel (key)
int key;
```

DESCRIPTION

Returns the label defined for key in the key translation file; the label is usually what is printed on the key on the physical keyboard. If there is no such label, returns the name of the logical key from the following table. Here is a list of key mnemonics:

EXIT	XMIT	HELP	BKSP	TAB
NL	BACK	HOME	DELE	INS
LP	FERA	CLR	SPGU	SPGD
LARR	RARR	DARR	UARR	REFR
EMOH	INSL	DELL	ZOOM	FHLP
CAPS	ABORT			

If the key code is invalid (not one defined in smkeys.h), this function returns the empty string.

RETURNS

A string naming the key, or the empty string if it has no name.

VARIANTS AND RELATED FUNCTIONS

```
sm_keyname (key);
```

EXAMPLE

```
#include "smkeys.h"

/* Put the name of the TRANSMIT key into a field
 * for help purposes. */

char buf[80];

sprintf (buf, "Press %s to commit the transaction.",
     sm_keylabel (XMIT));
sm_n_putfield ("help", buf);
```

NAME

sm_l_at_cur - display a library-resident window at the current cursor position

SYNOPSIS

```
int sm_l_at_cur (lib_desc, screen_name)
int lib_desc;
char *screen_name;
```

DESCRIPTION

Displays a library-resident window at the current cursor position, offset by one line to avoid hiding that line's contents.

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the top most, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

A screen library is a single file containing many screens. You can assemble one from individual screen files using the screen librarian, JYACC formlib. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on filesystem path searches.

The library descriptor is an integer returned by sm_l_open, which you must call before trying to read any screens from a library. Note that sm_r_window and related functions also search any open screen libraries.

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

0 is returned if no error occurred during display of the window. -2 is returned if lib_desc is invalid, or if that library does not contain screen_name. -3 is returned if a malloc failed to find available memory before the screen area was cleared, or if a malloc or read error occurred after the area was cleared but the screen was subsequently restored. -6 is returned if the screen library is corrupted.

VARIANTS AND RELATED FUNCTIONS

```
sm_l_open (library_name);
sm_l_form (library_descriptor, screen_name);
sm_l_window (library_descriptor, screen_name, line, column);
sm_l_close (library_descriptor);
```

EXAMPLE

```
/* Bring up a window from a library. */

int ld;

if ((ld = sm_l_open ("myforms")) < 0)
     sm_cancel ();
...
sm_l_at_cur (ld, "popup");
```

NAME

sm_l_close - close a screen library

SYNOPSIS

```
     int sm_l_close (lib_desc)
     int lib_desc;
```

DESCRIPTION

Closes the screen library indicated by lib_desc and frees all associated memory. The library descriptor is a number returned by a previous call to sm_l_open.

RETURNS

-1 is returned if the library file could not be closed; -2 is returned if the library was not open; 0 is returned if the library was closed successfully.

VARIANTS AND RELATED FUNCTIONS

```
     sm_l_open (library_name);
     sm_l_at_cur (library_descriptor, screen_name);
     sm_l_form (library_descriptor, screen_name);
     sm_l_window (library_descriptor, screen_name, line, column);
```

EXAMPLE

```
/* Bring up a window from a library. */

int ld;

if ((ld = sm_l_open ("myforms")) < 0)
     sm_cancel ();
...
sm_l_at_cur (ld, "popup");
...
sm_l_close (ld);
```

NAME

sm_l_form - display a library-resident screen as a base form

SYNOPSIS

```
int sm_l_form (lib_desc, screen_name)
int lib_desc;
char *screen_name;
```

DESCRIPTION

This function displays a library-resident screen as a base form.

Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new screen is displayed with its upper left-hand corner at the extreme upper left of the display (position (0, 0)). Any error in this function leaves the display and JAM internals in an undefined state.

If the form contains display data that are too big for the physical display, they are truncated without warning. However, if there are fields that won't fit within the physical display, this function returns an error without displaying the form.

A screen library is a single file containing many screens. You can assemble one from individual screen files using the screen librarian, JYACC formlib. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on filesystem path searches.

The library descriptor is an integer returned by sm_l_open, which you must call before trying to read any screens from a library. Note that sm_r_window and related functions also search any open screen libraries.

This function should be called by JAM applications only under unusual circumstances, as it does not update the control stack. You should execute a control string to display the form instead.

RETURNS

0 is returned if no error occurred during display of the screen. -2 is returned if lib_desc is invalid, or does not contain screen_name. -4 is returned if, after the screen has been cleared, the form cannot be displayed successfully because of a read error. -5 is returned if, after the screen has been cleared,not enough memory is available. -6 is returned if the library is corrupt.

VARIANTS AND RELATED FUNCTIONS

```
sm_l_window (library_descriptor, screen_name, line, column);
sm_l_at_cur (library_descriptor, screen_name);
sm_l_open (library_name);
sm_l_close (library_descriptor);
```

EXAMPLE

```
/* Put up a base form from a previously opened library. */

extern int formlib1;

if (sm_l_form (formlib1, "background"))
     sm_cancel ();
```

NAME

sm_l_open - open a screen library

SYNOPSIS

```
int sm_l_open (lib_name)
char *lib_name;
```

DESCRIPTION

Opens a screen library created by JYACC formlib, preparatory to displaying screens therein. It allocates space to store information about the library, leaves the library file open, and returns a descriptor identifying the library. That descriptor may subsequently be used by sm_l_window and related functions, to display screens stored in the library; or the library can be referenced implicitly by sm_r_window and related functions, which search all open screen libraries.

The library file is sought in all the directories identified by SMPATH and the parameter to sm_initcrt. Defining the SMFLIBS variable in your setup file to a list of library names will cause this function to be called on the libraries; all will then be searched automatically by sm_r_window and variants.

Several libraries may be kept open at once. This may cause problems on systems with severe limits on memory or simultaneously open files.

RETURNS

-1 if the library cannot be opened or read; -2 if too many libraries are already open; -3 if the named file is not a library; -4 if insufficient memory is available; Otherwise, a non-negative integer that identifies the library file.

VARIANTS AND RELATED FUNCTIONS

```
sm_l_form (library_descriptor, screen_name);
sm_l_at_cur (library_descriptor, screen_name);
sm_l_window (library_descriptor, screen_name, line, column);
sm_l_close (library_descriptor);
sm_r_window (screen_name, line, column);
```

EXAMPLE

```
/* Prompt for the name of a screen library until a
 * valid one is found. Assume the memory-resident
 * screen contains one field for entering the library name,
 * with suitable instructions. */

int ld;
extern char libquery[];

if (sm_d_form (libquery) < 0)
     sm_cancel ();
sm_d_msg_line ("Please enter the name of your screen library.");

do {
     sm_openkeybd ();
} while ((ld = sm_l_open (sm_fptr (1))) < 0);
```

NAME

sm_l_window - display a library-resident window at a given location

SYNOPSIS

```
int sm_l_window (lib_desc, screen_name, start_line, start_column)
int lib_desc, start_line, start_column;
char *screen_name;
```

DESCRIPTION

The screen screen_name is read from the library indicated by lib_desc, and displayed with its upper left-hand corner at (line, column). The line and column are counted from zero: if line is 1, the screen is displayed starting at the second line of the screen.

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the top most, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

A screen library is a single file containing many screens. You can assemble one from individual screen files using the screen librarian, JYACC formlib. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on filesystem path searches.

The library descriptor is an integer returned by sm_l_open, which you must call before trying to read any screens from a library. Note that sm_r_window and related functions also search any open screen libraries.

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

0 is returned if no error occurred during display of the form. -1 is returned if the window could not be displayed successfully because the format was incorrect. -2 is returned if lib_desc is invalid, or does not contain screen_name. -3 is returned if the system ran out of memory but the screen was restored successfully.

VARIANTS AND RELATED FUNCTIONS

```
sm_l_open (library_name);
sm_l_form (library_descriptor, screen_name);
sm_l_at_cur (library_descriptor, screen_name);
sm_l_close (library_descriptor);
sm_r_window (screen_name, line, column);
```

EXAMPLE

```
/* Bring up a window from a library. */

int ld;

if ((ld = sm_l_open ("myforms")) < 0)
     sm_cancel ();
...
sm_l_window (ld, "popup", 5, 22);
...
sm_l_close (ld);
```

NAME

sm_label_key - put a function key block label onto the screen

SYNOPSIS

```
     int sm_label_key (label_number, line1_text,
          line2_text)
     int label_number;
     char *line1_text, *line2_text;
```

DESCRIPTION

Certain terminals provide special areas on the screen to hold descriptions of function key actions. This routine places a label into one of those blocks. Label_number tells which block to use; the two strings will be placed on line 1 and line 2 of the block. (For a blank line, pass the empty string.)

This function assumes that there are two lines available per block. The KPAR and KSET entries must be present in the video file in order for this routine to work. Refer to the video manual in the Configuration Guide for instructions.

RETURNS

-1 if the label number is out of range, 0 otherwise.

EXAMPLE

```
/* Put any old label into the first two function key blocks. */

     sm_label_key(1, "blk1, ln1", "blk1, ln2");
     sm_label_key(2, "blk2, ln1", "blk2, ln2");
```

NAME

sm_last - position the cursor in the last field

SYNOPSIS

```
     void sm_last ();
```

DESCRIPTION

Places the cursor at the first enterable position of the last unprotected field of the current form. The first enterable position is the leftmost in a left-justified field, and the rightmost in a right-justified field; furthermore, if the field is digits-only, punctuation characters will be skipped.

Unlike sm_home, this function will not reposition the cursor if the screen has no unprotected fields.

VARIANTS AND RELATED FUNCTIONS

```
     sm_home ();
     sm_gofield (field_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Assume the last field must be entered for confirmation.
 * This code puts the cursor there, after the TRANSMIT key
 * is pressed. */

while (sm_openkeybd () != XMIT)
     ;
sm_unprotect (sm_numflds);
sm_last ();
sm_openkeybd ();
if (sm_is_yes (sm_numflds))
     commit ();  /* Finish whatever it is */
```

NAME

sm_lclear - erase LDB entries of one scope

SYNOPSIS

```
int sm_lclear (scope)
int scope;
```

DESCRIPTION

This function erases the values stored in the local data block for all names having a scope of scope; legal values are between 1 and 9. Constant variables having scope 1 can be erased.

Refer to the Author's Guide for a discussion of the scope of LDB entries.

VARIANTS AND RELATED FUNCTIONS

```
sm_lreset (file_name, scope);
```

EXAMPLE

```
/* Clear out LDB entries of scope 6, which has been
 * assigned to customer information. */

#define CUSTOMER_SCOPE 6

sm_lclear (CUSTOMER_SCOPE);
```

NAME

    sm_ldb_init - initialize (or reinitialize) the local data block

SYNOPSIS

```
     void sm_ldb_init ();
```

DESCRIPTION

This function creates an empty index of named data items by reading the data dictionary, then loads values into them from initialization files. There is no LDB prior to the execution of this function.

Selected parts of the LDB, namely those assigned a certain scope, can be reinitialized using sm_lclear or sm_lreset.

This function is called automatically during JAM initalization. Other functions that affect its behavior, such as sm_dicname and sm_ininames, should be called first.

VARIANTS AND RELATED FUNCTIONS

```
     sm_lreset (filename, scope);
     sm_ininames (file_list);
     sm_dicname (dictionary_name);
```

EXAMPLE

```
/* After a catastrophic application failure, reboot the index. */
if (bad_data ())
{
     sm_ldb_init ();
     ...
}
```

NAME

sm_leave - prepare to leave a JAM application temporarily

SYNOPSIS

```
void sm_leave ();
```

DESCRIPTION

It may at times be necessary to leave a JAM application temporarily: to escape to the command interpreter, to execute some graphics functions, and so on. In such a case, the terminal and its operating system channel need to be restored to their normal states.

This function should be called before leaving. It clears the physical screen (but not the internal screen image); resets the operating system channel; and resets the terminal (using the RESET sequence found in the video file).

VARIANTS AND RELATED FUNCTIONS

```
sm_return ();
```

EXAMPLE

```
#include "smdefs.h"

/* Escape to the UNIX shell for a directory listing */

sm_leave ();
system ("ls -l");
sm_return ();
sm_c_off ();
sm_d_msg_line ("Hit any key to continue", BLINK | WHITE);
sm_getkey ();
sm_d_msg_line ("", WHITE);
sm_rescreen ();
```

NAME

```
     sm_length - get the maximum length of a field
```

SYNOPSIS

```
     int sm_length (field_number)
     int field_number;
```

DESCRIPTION

Returns the maximum length of the field specified by field_number. If the field is shiftable, its maximum shifting length is returned. This length is as defined in jxform, and has no relation to the current contents of the field; use sm_dlength to get the length of the contents.

RETURNS

     Length of the field, or 0 if the field is not found.

VARIANTS AND RELATED FUNCTIONS

```
     sm_n_length (field_name);
     sm_dlength (field-number);
```

EXAMPLE

```
/* Compute the number of blanks left in a
 * right-justified field (number 6), and fill them
 * with asterisks. */

int blanks, k;
char buf[256];

blanks = sm_length (6) - sm_dlength (6);
for (k = 0; k < blanks; ++k)
     buf[k] = '*';
sm_getfield (buf + blanks, 6);
sm_putfield (6, buf);
```

NAME

sm_lngval - get the long integer value of a field

SYNOPSIS

```
#include "smdefs.h"

long sm_lngval (field_number)
int field_number;
```

DESCRIPTION

Returns the contents of field_number, converted to a long integer. All non-digit characters are ignored, except of course for a leading plus or minus sign.

RETURNS

The long value of the field; 0 if the field is not found.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_lngval (field_name, element);
sm_i_lngval (field_name, occurrence);
sm_n_lngval (field_name);
sm_o_lngval (field_number, occurrence);
sm_intval (field_number);
sm_ltofield (field_number, value);
```

EXAMPLE

```
#include "smdefs.h"

/* Retrieve the number of fish in one particular sea
 * (a big number) from the screen. */

#define MEDITERRANEAN 4
long fish;

fish = sm_e_lngval ("seas", MEDITERRANEAN);
```

NAME

sm_lreset - reinitialize LDB entries of one scope

SYNOPSIS

```
int sm_lreset (file_name, scope)
char *file_name;
int scope;
```

DESCRIPTION

This function sets local data block entries to values read from file_name. Scope must be between 1 and 9; references in the file to LDB entries not belonging to scope are ignored. All variables belonging to scope are cleared before reinitializing; therefore, this function erases variables that are not in the file.

The file may be in the current directory, or in any of the directories listed in the SMPATH environment variable. It contains pairs of names with values, each enclosed in quotes. While all whitespace outside the quotes is ignored, we recommend for readability that the file have one name-value pair per line. If an entry has multiple occurrences, it may be subscripted in the file. Here are a few sample pairs:

```
"husband" "Ronald Reagan"
"wife[1]" "Jane Wyman"
"wife[2]" "Nancy Davis"
```

If you plan to use this function, we recommend that you group your variables in separate files by scope. You can use sm_ininames to list a number of files for initialization.

RETURNS

-1 if error (file not found or scope out of range), 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_lclear (scope);
```

EXAMPLE

```
/* Reinitialize LDB entries of scope 6, which has been
 * assigned to customer information. */

#define CUSTOMER_SCOPE 6
#define CUSTOMER_INIT  "customers.ini"

sm_lreset (CUSTOMER_INIT, CUSTOMER_SCOPE);
```

NAME

sm_lstore - copy everything from screen to LDB

SYNOPSIS

```
int sm_lstore ();
```

DESCRIPTION

Copies data from the screen to local data block entries with matching names. In its loop, this function makes use of a data structure set up during screen display that identifies which fields have LDB entries.

This function is similar to the Release 3 function ldb_store. It is called automatically by the JAM screen display logic, when bringing up a new screen, and need not be called by application code except under special circumstances.

RETURNS

-3 if sufficient memory is not available, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_allget (respect_flag);
```

NAME

sm_ltofield - place a long integer in a field

SYNOPSIS

```
int sm_ltofield (field_number, value)
int field_number;
long value;
```

DESCRIPTION

The long integer passed to this routine is converted to human-readable form and placed in field_number. If the number is longer than the field, it is truncated without warning, on the right or left depending on the field's justification.

RETURNS

-1 if the field is not found. 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_ltofield (field_name, element, value);
sm_i_ltofield (field_name, occurrence, value);
sm_n_ltofield (field_name, value);
sm_o_ltofield (field_number, occurrence, value);
sm_lngval (field_number);
sm_itofield (field_number, value);
```

EXAMPLE

```
#include "smdefs.h"

/* Set the number of fish in the sea to a smallish number. */

#define MEDITERRANEAN 4

sm_e_ltofield ("seas", MEDITERRANEAN, 14L);
```

NAME

sm_m_flush - flush the message line

SYNOPSIS

```
void sm_m_flush ();
```

DESCRIPTION

Forces updates to the message line to be written to the display. This is useful if you want to display the status of an operation with sm_d_msg_line, without flushing the entire display as sm_flush does.

VARIANTS AND RELATED FUNCTIONS

```
sm_flush ();
```

EXAMPLE

```
#include "smdefs.h"

/* Process a big pile of records, providing status as we go. */
char buf[80];
int k;

k = 0;
do {
     sprintf (buf, "Processing record %d", k + 1);
     sm_d_msg_line (buf, REVERSE | WHITE);
     sm_m_flush ();
} while (process (records[k++]) >= 0);
```

NAME

sm_max_occur - get the maximum number of occurrences that can be entered into an array or field

SYNOPSIS

```
int sm_max_occur (field_number)
int field_number;
```

DESCRIPTION

If the field or array designated by field_number is scrollable, returns the maximum number of items the scroll can hold. Note that this is the maximum as defined by jxform or a call to sm_sc_max, not the greatest number actually entered so far.

If the field is an element of a non-scrollable array, the function returns the number of elements in the array. If it is a non-scrollable single field, the function returns 1.

RETURNS

0 if the field designation is invalid; 1 for a non-scrollable single field; The number of elements in a non-scrollable array; The maximum number of items in a scrollable array or field.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_max_occur (field_name);
```

EXAMPLE

```
#include "smdefs.h"

/* Find the number of occurrences in an array of
 * whole numbers, say numbers of children, and
 * allocate some memory to hold them. */

int *children, howmany;

if ((howmany = sm_n_max_occur ("children")) > 0)
     children = (int *)calloc(howmany, sizeof(int));
```

NAME

sm_menu_proc - get a menu selection

SYNOPSIS

```
#include "smdefs.h"

int sm_menu_proc (type)
int type;
```

DESCRIPTION

Allows you to tab, backtab, arrow, and scroll through a menu screen, and select an item from it. The entry under the cursor is displayed in reverse video. The routine returns to the calling program when you hit EXIT or a function key (PF, SPF, or APP), or make a selection. A selection is made when you hit the TRANSMIT key, or a sequence of characters that uniquely match a menu entry (see mp_string).

Hitting a key that matches the first character of a menu entry on the screen causes the cursor to be positioned to that entry. If type is UPPER (or LOWER), any alphabetic keyboard entry is translated to upper (or lower) case before a match is attempted. If type contains both UPPER and LOWER, both translations are tried; the search is totally insensitive to case. Any other value yields a case-sensitive search. The search always starts at the beginning of the menu, and ignores off-screen data; to see off-screen menu items you must use the scrolling keys.

Each menu selection must be defined as initial data in a tab-unprotected menu field. Furthermore, unless you change the default setting by calling mp_string, each selection must begin with a unique character.(The JAM run-time system does so during its standard initialization sequence. You can define arbitrary return codes for each field using jxform; the default is to use the first character of the menu entry itself. See the JAM Author's Guide for a detailed discussion of menu creation and return values.

Two auxiliary functions, mp_options and mp_string, can alter the behavior of the cursor; refer to their definitions.

Menu control strings are not executed within this function, but at a higher level of the JAM run-time system. If you call this function, do not expect your selection's control string to be executed.

RETURNS

If the cursor is not within a field, returns 0. The translated value (see smkeys.h) of EXIT, or of any other function key except TRANSMIT. If a selection is made with TRANSMIT or a menu character, the menu return code defined in jxform, or the first character of the selected entry if there is no return code.

EXAMPLE

See sm_mp_string for an example.

NAME

sm_mp_options - define cursor motion for sm_menu_proc

SYNOPSIS

```
#include "smdefs.h"

int sm_mp_options (wrap, vertical_arrow,
     horizontal_arrow)
int wrap, vertical_arrow, horizontal_arrow;
```

DESCRIPTION

This function takes three parameters. Wrap determines whether the arrow keys wrap, that is, whether the cursor procedes from the rightmost field around to the leftmost on right arrow (and so forth). The TAB and BACKTAB keys always wrap. Vertical_arrow and horizontal_arrow influence which field the arrow keys land you in when wrapping is not imminent. If you want to leave any of the settings unchanged, pass the special value OK_NOCHANGE.

The mnemonics listed below are defined in smdefs.h ; they are the same as those used by ok_options.

wrap:

OK_NOWRAP	No wrapping. The terminal beeps if an attempt is made to arrow past the edge of the current form (or window). OK_NOWRAP is overridden by OK_TAB and equivalent settings.
OK_WRAP	Default. The arrow keys wrap. Vertical arrows wrap straight from bottom to top and vice versa; right arrow wraps to the beginning of the next (or first) line, left arrow to the end of the previous (or last) line.

vertical arrow (up- and down-arrow keys):

OK_NXTLINE	The cursor will be positioned to the closest field whose line is closest to the current line.
OK_FREE	Default. Same as OK_NXTLINE.
OK_RESTRICT	The arrow keys are not operative; the terminal will beep if they are pressed.
OK_SWATH	The cursor will be positioned to the closest field that overlaps the "swath" containing the current field.
OK_COLM	Same as OK_SWATH.
OK_NXTFLD	The cursor will be positioned to the field closest to the current line and column. The calculation uses the diagonal distance, assuming that the terminal has a 5 to 2 aspect ratio.
OK_TAB	The arrow keys behave like TAB and BACKTAB.

horizontal arrow (left- and right-arrow keys):

OK_TAB	The arrow keys behave like TAB and BACKTAB.
OK_FREE	Default. Same as OK_TAB.
OK_RESTRICT	The arrow keys are not operative.
OK_COLM	The cursor will be positioned to the closest field on the current line.
OK_SWATH	Same as OK_COLM.
OK_NXTLINE	The cursor will be positioned to the closest of those fields whose column is closest to the current column.
OK_NXTFLD	The cursor will be positioned to the field closest to the current line and column. The calculation uses the diagonal distance, assuming that the terminal has a 5 to 2 aspect ratio.

If you define the SMMPOPTIONS variable in your setup file, it will cause this function to be called automatically during start-up with the parameters you specify there.

RETURNS

-1 if any parameter is invalid (nothing is changed); 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_menu_proc (type);
```

EXAMPLE

```
#include "smdefs.h"

/* Restore the menu_proc options to their default values. */

if (sm_mp_options (OK_WRAP,OK_FREE,OK_FREE))
     sm_cancel ();
}
```

NAME

    sm_mp_string - set string option for sm_menu_proc

SYNOPSIS

```
     #include "smdefs.h"

     int sm_mp_string (option)
     int option;
```

DESCRIPTION

Sets the string option for sm_menu_proc to one of the following values. The mnemonics are defined in smdefs.h .

If option is OK_NOSTRING (the default), each data key struck is compared against the initial character of each menu item, beginning with the first. As soon as a match is found that entry is selected, regardless of whether there are other items that begin with the same character; therefore, the second and subsequent duplicate entries can never be selected by a data key.

If option is OK_STRING, data keys are collected until the saved sequence is long enough to match one menu item unambiguously. As keys are collected, the cursor moves to the item closest to the top that matches everything typed so far.

If you define the SMMPSTRING variable in your setup file, it will cause this function to be called automatically during start-up with the parameter you specify there.

Suppose a menu contains the following items, and the string option is on:

```
                              ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
                              º                 º
                              º AAA Auto        º
                              º Ace Body Work   º
                              º Acme Auto Parts º
                              º                 º
                              ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼
```

Typing a positions the bounce bar to the first item; typing c moves to the second; typing m moves to the third and selects it. Typing aa selects the first item.

RETURNS

    -1 if the option is invalid, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_menu_proc (type);
```

EXAMPLE

See above.

NAME

sm_msg - display a message at a given column on the status line

SYNOPSIS

```
void sm_msg (column, length, text)
int column, length;
char *text;
```

DESCRIPTION

The message is merged with the current contents of the status line, and displayed beginning at column. Length gives the number of characters in text.

On terminals with onscreen attributes, the column position may need to be adjusted to allow for attributes embedded in the status line. Refer to sm_d_msg_line for an explanation of how to embed attributes and function key names in a status line message.

This function is called, for example, by the function that updates the cursor position display (see sm_c_vis).

VARIANTS AND RELATED FUNCTIONS

```
sm_d_msg_line (msg);
```

EXAMPLE

```
#include "smdefs.h"

/* This code displays a message, then chops out
 * part of it. */

char *text0 = "        ";
char *text1 = "Message is displayed on the status line at col 1.";

sm_msg(1, strlen(text1), text1);
sm_msg(12, strlen(text0), text0);
```

NAME

sm_msg_get - find a message given its number

SYNOPSIS

```
#include "smdefs.h"
#include "smerror.h"

void *sm_msg_get (number)
int number;
```

DESCRIPTION

The messages used by JAM library routines are stored in binary message files, which are created from text files by the JYACC msg2bin utility. Use sm_msgread to load message files for use by this function.

This function takes the number of the message desired and returns the message, or a less informative string if the message number cannot be matched.

Messages are divided into classes based on their numbers, with up to 4096 messages per class. The message class is the message number divided by 4096, and the message offset within the class is the message number modulo 4096. Predefined JAM message numbers and classes are defined in smerror.h .

RETURNS

The desired message, if found; the message class and number, as a string, otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_msgfind (number);
sm_msgread ();
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"

/* Assume that an anxious programmer has just typed
 * in the question, "Will my boss like my new program?"
 * This code fragment answers the question. */

sm_n_putfield ("answer", rand() & 1 ?
     sm_msg_get (SM_YES) :
     sm_msg_get (SM_NO));
```

NAME

sm_msgfind - find a message given its number

SYNOPSIS

```
     #include "smdefs.h"
     #include "smerror.h"

     char *sm_msgfind (number)
     int number;
```

DESCRIPTION

This function takes the number of a screen manager message, and returns the message string. It is identical to sm_msg_get, except that it returns zero if the message number is not found.

Screen manager message numbers are defined in smerror.h .

RETURNS

The message, or 0 if the message number is out of range.

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"

/* print out message #4 */

sprintf (buf, "The message reads: %s\n", sm_msgfind (SM_BADKEY));
sm_quiet_err (buf);
```

NAME

sm_msgread - read message file into memory

SYNOPSIS

```
#include "smerror.h"

int sm_msgread (code, class, mode, arg)
char *code;
int class;
int mode;
char *arg;
```

DESCRIPTION

Reads a single set of messages from a binary message file into memory, after which they can be accessed using sm_msg_get and sm_msgfind. Code selects a single message class from a file that may contain several classes:

Code	Class	Contents
SM	SM_MSGS	Screen manager messages
FM	FM_MSGS	Formaker (xform) messages
JM	JM_MSGS	JAM run-time messages
JX	JX_MSGS	JAM Formaker (jxform) messages
(blank)		Undesignated user messages

Class identifies a class of messages. Classes 0-7 are reserved for user messages, and several classes are reserved to JAM; see smerror.h . As messages with the prefix code are read from the file, they are assigned numbers sequentially beginning at 4096 times class.

Mode is a mnemonic or mnemonics drawn from the following list. The first five indicate where to get the message file; at least one of these must be supplied. The other four modify the basic action.

Mnemonic	Action
MSG_DELETE	Delete the message class and recover its memory.
MSG_DEFAULT	Use the default file defined by the SMMSGS setup variable.
MSG_FILENAME	Use a file named in arg.
MSG_ENVIRON	Use the file named in an environment variable named by arg.
MSG_MEMORY	Use a memory-resident file whose address is given by arg.
MSG_NOREPLACE	Modifier: do not overwrite previously installed messages.
MSG_DSK	Modifier: leave file open, do not read into memory
MSG_INIT	Modifier: do not use screen manager error reporting.
MSG_QUIET	Modifier: do not report errors.

You can or MSG_NOREPLACE with any mode except MSG_DELETE, to prevent overwriting messages read previously. Error messages will be displayed on the status line, if the screen has been initialized by sm_initcrt; otherwise, they will go to the standard error output. You can or MSG_INIT with the mode to force error messages to standard error. Combining the mode with MSG_QUIET suppresses error reporting altogether.

If you or MSG_DSK with the mode, the messages are not read into memory. Instead the file is left open, and sm_msg_get and sm_msgfind fetch them from disk when requested. If your message file is large, this can save substantial memory; but you should remember to account for operating system file buffers in your calculations.

Arg contains the environment variable name for MSG_ENVIRON; the file name for MSG_FILENAME; or the address of the memory-resident file for MSG_MEMORY. It may be passed as zero for other modes.

RETURNS

0 if the operation completed successfully; 1 if the message class was already in memory and the mode included MSG_NOREPLACE; 2 if the mode was MSG_DELETE and the message file was not in memory; -1 if the mode was MSG_ENVIRON or MSG_DEFAULT and the environment variable was undefined; -2 if the mode was MSG_ENVIRON, MSG_FILENAME, or MSG_DEFAULT and the message file could not be read from disk; other negative values if the message file was bad or insufficient memory was available.

VARIANTS AND RELATED FUNCTIONS

```
     sm_msg_get (number);
     sm_msgfind (number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"

/****************** Example 1 *********************
 * These calls are issued by sm_initcrt() to load
 * standard messages and one set of user messages. */

sm_msgread ("SM", SM_MSGS, MSG_DEFAULT, (char *)0);
sm_msgread ("", 0, MSG_DEFAULT, (char *)0);

/***************** Example 2 **********************
 * This code fragment duplicates the Release 3 routine
 * sm_msginit(). */

int sm_msginit (memfile)
char memfile[];
{
int sm_msginit (msg_file)
char * msg_file;
{
     int mode = (msg_file ? MSG_MEMORY : MSG_DEFAULT | MSG_NOREPLACE)
          | MSG_INIT;

     if ( sm_msgread ("SM", SM_MSGS, mode, msg_file) < 0 ||
          sm_msgread ("JM", JM_MSGS, mode, msg_file) < 0 ||
          sm_msgread ("FM", FM_MSGS, mode, msg_file) < 0 ||
          sm_msgread ("JX", JX_MSGS, mode, msg_file) < 0)
     {
          exit (RET_FATAL);
     }
     sm_msgread ((char *)0, 0, mode & ~MSG_INIT | MSG_QUIET, msg_file);
     return (0);
}
```

NAME

sm_mwindow - display a status message in a window

SYNOPSIS

```
int sm_mwindow (text, line, column)
char *text;
int line;
int column;
```

DESCRIPTION

This function displays text in a pop-up window, whose upper left-hand corner appears at line and column. The line and column are counted from zero: if line is 1, the top of the window will be on the second line of the display. The window itself is constructed on the fly by the run-time system; no data entry is possible in it, nor is data entry possible in underlying screens as long as it is displayed. Make sure that sm_close_window is called after this function.

All the percent escapes for status messages, except %M, are effective; refer to sm_d_msg_line for a list and full description. If either line or column is negative, the window will be displayed according to the rules given at sm_r_at_cur.

RETURNS

-1 if there was a malloc failure. 1 if the text had to be truncated to fit in a window 0 otherwise

VARIANTS AND RELATED FUNCTIONS

```
sm_d_msg_line (text, attribute);
```

EXAMPLE

```
/* By judicious use of %N's, it is possible to get your
 * messages centered on the screen when you call
 * sm_mwindow().
 */

void poem ()
{
     sm_mwindow ("The world is too much with us. Late and soon,%N\
Getting and spending, we lay waste our powers.%N\
Little we see in Nature that is ours;%N\
We have given our hearts away, a sordid boon!%N%N\
The sea that bares her bosom to the Moon,%N\
The winds that will be raging at all hours,%N\
And are up-gathered now like sleeping flowers,%N\
For this, for everything, we are out of tune;%N\
It moves us not. Great God! I'd rather be%N\
A pagan, suckled in a creed outworn;%N\
So might I, standing on this pleasant lea,%N\
Have glimpses that would make me less forlorn:%N\
Catch sight of Proteus rising from the sea,%N\
Or hear old Triton blow his wreathed horn.",
     6, 16);
}
```

NAME

sm_n_1clear_array clear all data from a single scrolling array
sm_n_1protect
selectively protect a field sm_n_1unprotect
selectively unprotect a field sm_n_amt_format
format data and write to a field sm_n_aprotect
protect an entire array or scroll sm_n_aunprotect
unprotect an entire array or scroll sm_n_bitop
manipulate field edit bits sm_n_chg_attr
change the display attribute(s) of a field
sm_n_clear_array
clear all data from a scrolling array and parallel
arrays sm_n_dblval
get the decimal value of a field sm_n_dlength
get the length of data stored in a field sm_n_dtofield
write decimal value to a field sm_n_edit_ptr
get special edit sm_n_fldno
* see next page * sm_n_fptr
get copy of data in field sm_n_fval
force validation of a field sm_n_getfield
copy data from field into buffer sm_n_gofield
position cursor to a field sm_n_intval
get integer value of data in field sm_n_is_yes
test field for yes sm_n_itofield
write integer value to field sm_n_length
get length of field sm_n_lngval
get long integer value of data in field sm_n_ltofield
write long integer value to field sm_n_max_occur
get maximum occurrence of field sm_n_mdt_clear
reset field's MDT bit sm_n_mod_test
test field's MDT bit sm_n_novalbit
reset field's validated bit sm_n_num_items
get number of items entered in scrollable field or
array sm_n_off_gofield
move cursor to specified offset in a field
sm_n_protect
protect field from data entry sm_n_putfield
write data string to field sm_n_size_of_array
get number of elements in an array sm_n_unprotect
allow data entry into field sm_n_wselect
select a hidden window by name

DESCRIPTION

Each or the above functions accesses a field by means of the field name. For a description of sm_n_fldno, see the next page. For a description of any other function listed above, look under the related function without n_ in its name. For example, sm_n_amt_format is described under sm_amt_format. If the named field is not on the screen, these functions will attempt to access a similarly named entry in the local data block.

NAME

sm_n_fldno - get the field number of a named field

SYNOPSIS

```
int sm_n_fldno (field_name)
char *field_name;
```

DESCRIPTION

Returns the field number of a field specified by name, or the base field number of an array specified by name.

RETURNS

An integer ranging from 1 to the maximum number of fields on the current form. 0 if the field name is not found.

VARIANTS AND RELATED FUNCTIONS

sm_e_fldno (field_name, element)

EXAMPLE

```
#include "smdefs.h"

/* Write a list of real numbers to the screen. The first
 * and last fields in the list are tagged with special names.
 * Actually, sm_dtofield () would be more convenient.
 */
int k, first, last;
char buf[256];
double values[];
     /* set up elsewhere... */

last = sm_n_fldno ("last");
first = sm_n_fldno ("first");
for (k = first; k && k <= last; ++k)
{
     sprintf (buf, "%lf", values[k - first]);
     sm_amt_format (k, buf);
}
```

NAME

sm_nl - tab to the first unprotected field beyond the current line

SYNOPSIS

```
void sm_nl ();
```

DESCRIPTION

This function moves the cursor to the next item of a scrolling field or array, scrolling if necessary. Unlike the down-arrow, it will open up a blank scrolling item if there are no more below (but the maximum has not yet been exceeded).

If the current field is not scrolling, the cursor is positioned to the first unprotected field, if any, following the current line of the form. If there are no unprotected fields beyond there, the cursor is positioned to the first unprotected field of the form.

If the form has no unprotected fields at all, the cursor is positioned to the first column of the line following the current line; if the cursor is on the last line of the form, it goes to the top left-hand corner of the form.

This function is ordinarily bound to the RETURN key.

VARIANTS AND RELATED FUNCTIONS

```
sm_tab ();
sm_home ();
sm_last ();
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Scuttle down a scrolling array until we come
 * to a nonblank item, or run out of array. */

char buf[256];

while (sm_t_scroll (sm_fldnumber + 1) &&
     sm_getfield (buf, sm_fldnumber + 1) == 0)
{
     sm_nl ();
}
```

NAME

sm_novalbit - forcibly invalidate a field

SYNOPSIS

```
int sm_novalbit (field_number)
int field_number;
```

DESCRIPTION

Resets the VALIDED bit of the specified field, so that the field will again be subject to validation when it is next exited, or when the screen is validated as a whole.

JAM sets a field's VALIDED bit automatically when the field passes all its validations. The bit is initially clear, and is cleared whenever the field is altered by keyboard input or by a library function such as sm_putfield.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_novalbit (field_name, element)
sm_i_novalbit (field_name, occurrence)
sm_n_novalbit (field_name)
sm_o_novalbit (field_number, occurrence)
sm_fieldval ();
sm_s_val ();
```

EXAMPLE

```
#include "smdefs.h"

/* Here is a validation function for a "last_name"
 * field. When it is changed, it marks the
 * "first_name" field, which depends on it, invalid. */

int validate (field, data, occur, bits)
char *data;
{
     if (bits & VALIDED) /* Not really changed */
          return 0;

     sm_n_novalbit ("first_name");
     return 0;
}
```

NAME

sm_num_occurs - count the occurrences in a scrollable field or array

SYNOPSIS

```
int sm_num_occurs (field_number)
int field_number;
```

DESCRIPTION

Returns the number of items actually entered so far into the scrollable field or array identified by field_number. They may have been entered either through the keyboard, or through calls to sm_putfield or similar functions.

This count is different from the maximum capacity of a scroll, which you can retrieve by calling sm_max_occur.

RETURNS

-1 if the field is not found; otherwise, the number of items entered, possibly 0.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_num_occurs (field_name);
sm_max_occur (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Compute the number of unused items in this scroll. */

int unused;

unused = sm_n_max_occur ("hatpins") -
     sm_n_num_occurs ("hatpins");
```

NAME

```
     sm_o_achg            change the display attribute of a scrolling item
                          sm_o_amt_format
                          format a currency value and write to occurrence
                          sm_o_bitop
                          manipulate an occurrence's edit bits sm_o_chg_attr
                          change the display attribute of a field sm_o_dblval
                          get decimal value of occurrence sm_o_dlength
                          get length of data in occurrence sm_o_doccur
                          delete an occurrence from an array or scroll
                          sm_o_dtofield
                          write decimal value to occurrence sm_o_fptr
                          get copy of occurrence's data sm_o_fval
                          force validation of an occurrence sm_o_getfield
                          copy data from occurrence into buffer sm_o_gofield
                          position cursor to occurrence sm_o_intval
                          get integer value of occurrence sm_o_ioccur
                          insert a blank occurrence into an array or scroll
                          sm_o_itofield
                          write integer value to occurrence sm_o_lngval
                          get long integer value of occurrence sm_o_ltofield
                          write long integer value to occurrence sm_o_mdt_clear
                          reset MDT bit of an occurrence sm_o_mod_test
                          test MDT bit of an occurrence sm_o_novalbit
                          reset validated bit of an occurrence sm_o_off_gofield
                          place the cursor in the middle of an occurrence
                          sm_o_putfield
                          write data string to occurrence
```

DESCRIPTION

Each of the above functions refers to data by field number and occurrence number. As used in the above functions, occurrence means

1. item, if the field or array is scrollable;
2. element, if the specified field is part of a non-scrollable array; or
3. the specified field, if neither scrollable nor an array.

If the occurrence is zero, the reference is always to the current contents of the specified field.

For the description of a particular function, look under the related function without o_ in its name. For example, sm_o_amt_format is described under sm_amt_format.

NAME

sm_occur_no - get the occurrence number of data in the current field

SYNOPSIS

```
int sm_occur_no ();
```

DESCRIPTION

Returns the occurrence number of the data item in the current field. If the current field is scrollable, the occurrence number is the item ID, that is, the item's index in the whole scroll. If the field is an element of a non-scrollable array, the occurrence number is the field's element number. Otherwise, the occurrence number is 1.

RETURNS

0 if the cursor is not in a field; 1 if the cursor is in a field that is neither scrollable nor an array element; The element number if the cursor is in a non-scrollable array; The item id if the cursor is in a scrollable field or array.

VARIANTS AND RELATED FUNCTIONS

```
sm_getcurno ();
```

EXAMPLE

```
#include "smdefs.h"

/* Find the number of the scrolling item under the cursor,
 * and scroll down to the next higher multiple of 5. */

int thisn;

thisn = sm_occur_no ();
sm_rscroll (sm_getcurno (), 5 - (thisn % 5));
```

NAME

sm_off_gofield - move the cursor into a field, offset from the left

SYNOPSIS

```
int sm_off_gofield (field_number, offset)
int field_number, offset;
```

DESCRIPTION

Moves the cursor into field_number, at position offset within the field's contents, regardless of the field's justification. The field's contents will be shifted if necessary to bring the appropriate piece onscreen.

If offset is larger than the field length (or the maximum length if the field is shiftable), the cursor will be placed in the rightmost position.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_off_gofield (field_name, element, offset);
sm_n_off_gofield (field_name, offset);
sm_i_off_gofield (field_name, item_id, offset);
sm_o_off_gofield (field_number, item_id, offset);
sm_gofield (field_number);
sm_disp_off ();
sm_sh_off ();
```

EXAMPLE

```
#include "smdefs.h"
#include <ctype.h>

/* Place the cursor over the first embedded blank in the
 * "names" field. */

char buf[256], *p;
int length;

length = sm_n_getfield (buf, "names");
for (p = buf; p < length; ++p)
     if (isspace (*p))
          break;
sm_n_off_gofield ("names", p - buf);
```

NAME

sm_ok_options - set openkeybd options

SYNOPSIS

```
     #include "smdefs.h"

     int sm_ok_options (block, wrap, fld_reset,
          vert_arrow, horiz_arrow, val_opt)
     int block, wrap, fld_reset;
     int vert_arrow, horiz_arrow, val_opt;
```

DESCRIPTION

The six options set by this function control how the function sm_openkeybd responds to cursor motion keys and other keyboard input.

The first defines the appearance of the cursor. The second determines whether wrapping occurs on arrow keys. The third determines whether the arrow keys can be used to enter a field in the middle. The next two options define cursor movement in response to the arrow keys. The last determines when field validation is performed.

Six arguments must be passed, even if you want default values for some of them. Use the special value OK_NOCHANGE to leave a setting unaltered.

Mnemonics for setting the options are given in smdefs.h and are listed below. These mnemonics are also used for sm_mp_options.

block:

OK_NOBLOCK	Default. The cursor occupies one character position.
OK_BLOCK	The cursor fills the field. (Actually, the cursor is turned off and the current field made reverse video.)

wrap:

OK_NOWRAP	Default. No wrapping. The terminal beeps if an attempt is made to arrow past the edge of the current form (or window).
OK_WRAP	The arrow keys wrap. Vertical arrows wrap straight from bottom to top and vice versa; right arrow wraps to the beginning of the next (or first) line, left arrow to the end of the previous (or last) line.

fld_reset:

OK_NORESET	Default. The arrow keys can be used to enter the middle of a field.
OK_RESET	When a field is entered, the cursor always goes to the first data position (allowing for justification, and punctuation in digits-only fields).
OK_ENDCHAR	Default off. When a character is typed at the last position of a no-autotab field, beep the terminal instead of overwriting the last character.

vert_arrow (up and down arrow keys):

OK_FREE	Default. Free cursor movement.
OK_RESTRICT	The arrow keys will not take the cursor out of the current field.
OK_COLM	The cursor will be positioned to the closest field (observing wrapping, if set) that overlaps the current column.
OK_SWATH	The cursor will be positioned to the closest field (observing wrapping, if set) that overlaps the "swath" containing the current field.

OK_NXTLINE	The cursor will be positioned to the closest of those fields (observing wrapping, if set) whose line is closest to the current line.
OK_NXTFLD	The cursor will be positioned to the field closest to the current line and column. The calculation uses the diagonal distance, assuming that the terminal has a 5 to 2 aspect ratio.
OK_TAB	The arrows behave like TAB and BACKTAB, except that the up-arrow goes to the end of the field instead of the beginning.

horiz_arrow (left and right arrow keys):

OK_FREE	Default. Free cursor movement.
OK_RESTRICT	The arrow keys will not take the cursor out of the current field.
OK_COLM	The cursor will be positioned to the closest field on the current line.
OK_SWATH	Same as OK_COLM.
OK_NXTLINE	The cursor will be positioned to the closest field (observing wrapping, if set) whose column is closest to the current column.
OK_NXTFLD	The cursor will be positioned to the field closest to the current line and column. The calculation uses the diagonal distance, assuming that the terminal has a 5 to 2 aspect ratio.
OK_TAB	The arrows behave like TAB and BACKTAB, except that the left-arrow goes to the end of the field instead of the beginning.

val_opt:

OK_NOVALID	Default. Field validations are performed only on TAB and RETURN keys.
OK_VALID	Validations are performed whenever the field is exited (by arrows, backtab, etc.)

If you define the SMOKOPTIONS variable in your setup file, this function will automatically be called with the parameters you provide there.

RETURNS

0 is returned if all the arguments are valid; -1 is returned if they are not. In this case, no options are affected.

VARIANTS AND RELATED FUNCTIONS

```
     sm_openkeybd ();
```

EXAMPLE

```
#include "smdefs.h"

/* Restore the default sm_openkeybd options. */

sm_ok_options(OK_NOBLOCK, OK_NOWRAP, OK_NORESET,
     OK_FREE, OK_NOVALID);
```

NAME

sm_openkeybd - open the keyboard for data entry

SYNOPSIS

```
#include "smdefs.h"
#include "smkeys.h"

int sm_openkeybd ();
```

DESCRIPTION

This routine calls sm_getkey to read the keyboard and translate function keys to logical values. Ordinary keys, such as letters, numbers, and punctuation, are entered into fields on the screen, subject to restrictions and edits defined during screen authoring. Cursor motion keys, such as TAB, PAGE UP, and arrows, move the cursor between data items; exiting a field causes its contents to be validated against conditions defined during authoring. Data editing keys, such as ERASE, INSERT, and BACKSPACE, make it easier to alter existing data. The Author's Guide, in the section on "data entry," lists all the function keys and describes their actions.

The processing of TRANSMIT, EXIT, HELP, and the cursor positioning keys is determined by a routing table. For each key, if the EXECUTE bit is set, the appropriate action is performed (tab, field erase, etc.) If the RETURN bit is set, sm_openkeybd returns the key to its caller. Since the bits are examined independently, four combinations of actions are possible. The default setting is EXECUTE only (although normal execution may cause the key to be returned, as in the case of the EXIT key).

You may change key actions "on the fly". For example, to disable the BACKTAB key, the application program would execute the following:

```
sm_route_table[BACK] = 0;
```

while to make the ERASE key return to the application program without erasing the field's contents:

```
sm_route_table[FERA] = RETURN;
```

The mnemonics are defined in smkeys.h . The PF, SPF, and APP function keys are not in the routing table; they behave as though their entries were RETURN only. A few keys, including RESCREEN and LOCAL PRINT, are processed locally in sm_getkey and not returned to sm_openkeybd, unless their RETURN bits are set Note that data keys may be made returnable too.

sm_openkeybd returns to its caller either when you press a returnable key, or when you exit a return entry field (by completing or tabbing out). In the former case, it returns a code for the function key. When you exit a return entry field, it returns the field's return code, if one has been defined. If one has not, it returns the last key struck in the field (if the field is autotab) or the rightmost data character in the field (if it is no-autotab).

Another function, sm_ok_options, defines the specific behavior of certain keys under sm_openkeybd.

Application programmers should be aware that JAM control strings are not executed within this function, but at a higher level within the JAM run-time system. If you call this function, do not expect function key control strings to work.

RETURNS

See above.

EXAMPLE

```
#include "smkeys.h"

/* Beep at unwanted function keys. */
int key;

switch (key = sm_openkeybd ())
{
     case PF1:
          save_something ();
          break;
     case PF2:
          discard_something ();
          break;
     default:
          sm_bel();
          break;
}
```

VARIANTS AND RELATED FUNCTIONS

```
     sm_ok_options (block, wrap, field_reset, varrow, harrow, val_opt);
```

NAME

sm_oshift - shift a field by a given amount

SYNOPSIS

```
     int sm_oshift (field_number, offset)
     int field_number;
     int offset;
```

DESCRIPTION

Shifts the contents of field_number by offset positions. If offset is negative, the contents are shifted right (data past the left-hand edge of the field become visible); otherwise, the contents are shifted left. Shifting indicators, if displayed, are adjusted accordingly.

The field may be shifted by fewer than offset positions if the maximum shifting width is reached thereby.

RETURNS

The number of positions actually shifted, or 0 if the field is not found or is not shifting.

VARIANTS AND RELATED FUNCTIONS

```
     sm_n_oshift (field_name, offset);
```

EXAMPLE

```
#include "smdefs.h"

/* Shift the Republicans gently toward the left,
 * and the Democrats toward the right.
 * For extra credit, speculate on which shift is positive. */

sm_n_oshift ("GOP", 1);
sm_n_oshift ("DEM", -1);
```

NAME

```
     sm_plcall - execute a JPL procedure
```

SYNOPSIS

```
     int sm_plcall (procedure_name)
     char *procedure_name;
```

DESCRIPTION

Executes the JPL procedure contained in the file procedure_name. The file will be sought in all the directories listed in SMPATH.

Although JPL procedures can return values through this function, their usual use is to modify named fields on the screen and in the LDB.

JPL, the JYACC Procedural Language, is the subject of a separate chapter.

RETURNS

-1 if the procedure could not be loaded; otherwise, the value returned by the JPL procedure.

EXAMPLE

```
#include "smdefs.h"

/* Imagine that you want to validate a field using a JPL
* procedure, but only on certain conditions that are awkward
* to test in JPL. You could use the following field exit
* function: */

int validate (field, data, occur, bits)
char *data;
{
     char *procname;

     if (bits & VALIDED)
          return;
     if (field_needs_validating (field, data) &&
          (procname = sm_edit_ptr (field, JPLTEXT)))
     {
          return sm_plcall (procname + 2);
     }
}
```

NAME

sm_protect - protect a field completely

SYNOPSIS

```
#include "smdefs.h"

int sm_protect (field_number)
int field_number;
```

DESCRIPTION

Protects the specified field from all of the following.

Mnemonic	Meaning
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into (or from entering via any other key)
CPROTECT	protect from clearing
VPROTECT	protect from validation routines

To protect a field selectively, use sm_1protect; to protect an entire array, use sm_aprotect.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_protect (field_name, element)
sm_n_protect (field_name)
sm_unprotect (field_number);
sm_1protect (field_number);
sm_aprotect (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* If the executive has a PC, unprotect a field to
 * hold its make; otherwise, protect that field. */

if (sm_n_is_yes ("owns_pc"))
     sm_n_unprotect ("pc_make");
else sm_n_protect ("pc_make");
```

NAME

sm_putfield - put a string into a field

SYNOPSIS

```
int sm_putfield (field_number, data)
int field_number;
char *data;
```

DESCRIPTION

The string data is moved into the field specified by field_number. If the string is too long, it is truncated without warning. If it is shorter than the destination field, it is blank filled (to the left if the field is right justified, otherwise to the right). If data is empty or zero the field's contents are erased; if the field is a date or time field taking system values, the date or time is refreshed.

This function sets the field's MDT bit to indicate that it has been modified, and clears its VALIDED bit to indicate that the field must be revalidated upon exit. sm_n_putfield and sm_i_putfield will store data in the LDB if the named field is not present in the screen. However, if the LDB item has a scope of 1 (constant), its contents will be unaltered and the function will return -1.

Notice that the order of arguments to this function is different from that of arguments to the related function sm_getfield.

RETURNS

-1 if the field is not found, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_putfield (field_name, element, data)
sm_i_putfield (field_name, occurrence, data)
sm_n_putfield (field_name, data)
sm_o_putfield (field_number, occurence)
sm_getfield (buffer, field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Perform sm_putfield specifying a field and a string the size
 * of the field. Verify that the field matches the string
 * and that 0 is returned. */

sm_putfield (1, "This string has 29 characters");
```

NAME

sm_putjctrl - associate a control string with a key

SYNOPSIS

```
int sm_putjctrl (key, control_string, default)
int key;
char *control_string;
int default;
```

DESCRIPTION

Each JAM screen contains a table of control strings associated with function keys. JAM also maintains a default table of keys and control strings, which take effect when the current screen has no control string for a function key you press. This table enables you to define system-wide actions for keys. It is initialized from INICTRL setup variables; see the section on setup in the Configration Guide.

This function associates control_string with key in one of the tables, replacing the control string previously associated with key (if there was one). If the flag default is zero, the control string will be installed in the current screen, and will disppear when you exit the screen; otherwise, it will go into the system-wide default table.

If you install a default control string for a key that is defined in the current screen, the definition in the screen will be used. Note also that JAM will not search back up the form or window stack for function key definitions; only the current screen and the default table are consulted.

Mnemonic values for key are in smkeys.h . The syntax for control strings is defined in the Author's Guide.

RETURNS

-5 if insufficient memory is available; 0 otherwise.

EXAMPLE

```
#include "smkeys.h"

/* These three calls duplicate the default associations for
* the JAM run-time system. */

sm_putjctrl (SPF1, "^jm_gotop");
sm_putjctrl (SPF2, "^jm_system");
sm_putjctrl (SPF3, "^jm_goform");
```

NAME

sm_query_msg - display a question, and return a yes or no answer

SYNOPSIS

```
     int sm_query_msg (message)
     char *message;
```

DESCRIPTION

The message is displayed on the status line, until you type a yes or a no key. A yes key is the first letter of the SM_YES entry in the message file, and a no key is the first letter of the SM_NO entry; case is ignored. At that point, this function returns a lower case 'y' or 'n' (English!) to its caller. All keys other than yes and no keys are ignored.

The initial attribute for the message defaults to highlighted reverse video; you may alter it by calling sm_ch_qmsgatt. Refer to sm_d_msg_line for an explanation of how to embed changed attributes and function key names in your message. The old status line is redisplayed as soon as you answer the question.

RETURNS

Lower-case ASCII 'y' or 'n', according to the response.

VARIANTS AND RELATED FUNCTIONS

```
     sm_d_msg_line (message, attribute);
     sm_is_yes (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Ask a couple of straightforward questions. Be careful of
 * the dangling else, which has ruined many relationships. */

if (sm_query_msg("Are you single?") == 'y')
     if (sm_query_msg("Will you go out with me?") == 'y')
          if (sm_query_msg("Do you like Clint Eastwood movies?") == 'n')
               ...
```

NAME

sm_qui_msg - display a message preceded by a constant tag, and reset the message line

SYNOPSIS

```
void sm_qui_msg (message)
char *message;
```

DESCRIPTION

This function prepends a tag (normally "ERROR:") to message, and displays the whole on the status line (or in a window if it is too long). The tag may be altered by changing the SM_ERROR entry in the message file. The message remains visible until the operator presses a key; refer to sm_er_options for an exact description of error message acknowledgement. If the message is longer than the status line, it will be displayed in a window instead.

This function is identical to sm_quiet_err, except that it does not turn the cursor on. It is similar to sm_emsg, which does not prepend a tag.

Several percent escapes provide control over the content and presentation of status messages. They are interpreted by sm_d_msg_line, which is eventually called by everything that puts text on the status line (including field status text). The character following the percent sign must be in upper-case; this is to avoid conflict with the percent escapes used by printf and its variants. Certain percent escapes (%W, for instance; see below) must appear at the beginning of the message, i.e. before anything except perhaps another percent escape.

- If a string of the form %Annnn appears anywhere in the message, the hexadecimal number nnnn is interpreted as a display attribute to be applied to the remainder of the message. The table below gives the numeric values of the logical display attributes you will need to construct embedded attributes. If you want a digit to appear immediately after the attribute change, pad the attribute to 4 digits with leading zeroes; if the following character is not a legal hex digit, leading zeroes are unnecessary.
- If a string of the form %KKEYNAME appears anywhere in the message, KEYNAME is interpreted as a logical key mnemonic, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the %K is stripped out and the mnemonic remains. Key mnemonics are defined in smkeys.h ; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.
- If %N appears anywhere in the message, the latter will be presented in a pop-up window rather than on the status line, and all occurrences of %N will be replaced by newlines.
- If the message begins with a %B, JAM will beep the terminal (using sm_bel) before issuing the message.
- If the message begins with %W, it will be presented in a pop-up window instead of on the status line. The window will appear near the bottom center of the screen, unless it would obscure the current field by so doing; in that case, it will appear near the top. If the message begins with %MU or %MD, and is passed to one of the error message display functions, JAM will ignore the default error message acknowledgement flag and process (for %MU) or discard (for %MD) the next character typed.

Note that, if a message containing percent escapes - that is, %A, %B, %K, %N or %W - is displayed before sm_initcrt or after %W is called, the percent escapes will show up in it.

Attribute	Hex value
BLACK	0 BLUE
	1 GREEN
	2 CYAN
	3 RED
	4 MAGENTA
	5 YELLOW
	6 WHITE
	7
B_BLACK	0 B_BLUE
	100 B_GREEN
	200 B_CYAN
	300 B_RED
	400 B_MAGENTA
	500 B_YELLOW
	600 B_WHITE
	700
BLANK	8 REVERSE
	10 UNDERLN
	20 BLINK
	40 HILIGHT
	80 DIM
	1000

If the cursor position display has been turned on (see sm_c_vis), the end of the status line will contain the cursor's current row and column. If the message text would overlap that area of the status line, it will be displayed in a window instead.

VARIANTS AND RELATED FUNCTIONS

```
     sm_er_options (key, discard);
     sm_emsg (message);
     sm_err_reset (message);
     sm_quiet_err (message);
     sm_await_space (message);
```

EXAMPLE

```
#include "smdefs.h"

sm_qui_msg ("Be %A17veewwwwy%A7 quiet. I'm hunting wabbits.");
```

NAME

sm_quiet_err - display error message preceded by a constant tag, and reset the status line

SYNOPSIS

```
     void sm_quiet_err (message)
     char *message;
```

DESCRIPTION

This function prepends a tag (normally "ERROR") to message, turns the cursor on, and displays the whole message on the status line (or in a window if it is too long). The tag may be altered by changing the SM_ERROR entry in the message file. The message remains visible until the operator presses a key; refer to sm_er_options for an exact description of error message acknowledgement. If the message is longer than the status line, it will be displayed in a window instead.

This function is identical to sm_qui_msg, except that it turns the cursor on. It is similar to sm_err_reset, which does not prepend a tag. Refer to sm_d_msg_line for an explanation of how to change display attributes and insert function key names within a message.

VARIANTS AND RELATED FUNCTIONS

```
     sm_er_options (key, discard);
     sm_emsg (message);
     sm_err_reset (message);
     sm_qui_msg (message);
     sm_await_space (message);
```

EXAMPLE

```
/* Display an error message that is surely long
 * enough to be put into a window. */

char *buf;

if ((buf = malloc (8192)) == 0)
{
     sm_quiet_err ("Sorry, guy, I'm %A0017all%A7 out of memory. Here's \
500 bucks, why don't you just run down to the corner dealer and \
pick me up a meg?");
     sm_cancel ();
}
```

NAME

sm_r_at_cur - display a window at the current cursor position

SYNOPSIS

```
int sm_r_at_cur (screen_name)
char *screen_name;
```

DESCRIPTION

Displays a window at the current cursor position, offset by one line to avoid hiding that line's current display.

The named screen is sought first in the memory-resident form list, and if found there is displayed using sm_d_window. It is next sought in all the open screen libraries, and if found is displayed using sm_l_window. Next it is sought on disk in the current directory; then under the path supplied to sm_initcrt; then in all the paths in the setup variable SMPATH. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the top most, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

0 if no error occurred during display of the window; -1 if the window cannot be successfully displayed because the format is incorrect; -2 if the form cannot be found; -3 if the system ran out of memory but the screen was restored to its former state.

VARIANTS AND RELATED FUNCTIONS

```
sm_r_form  name);
sm_r_window (name, line, column);
sm_close_window ();
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* In a validation routine, if the field contains a
 * special value, open up a window to prompt for a
 * second value and save it in another field. */

int validate (field, data, occur, bits)
char *data;
{
     char buf[256];

     if (bits & VALIDED)
          return 0;

     if (strcmp(data, "other") == 0)
     {
          sm_r_at_cur ("getsecval");
          if (sm_openkeybd () != EXIT)
               sm_getfield (buf, 1);
          else buf[0] = 0;
          sm_close_window ();
          sm_n_putfield ("secval", buf);
     }

     return 0;
}
```

NAME

sm_r_form - display a screen as a form

SYNOPSIS

```
int sm_r_form (screen_name)
char *screen_name;
```

DESCRIPTION

This function displays the named screen as a base form.

Bringing up a screen as a form causes the previously displayed form and windows to be discarded, and their memory freed. The new screen is displayed with its upper left-hand corner at the extreme upper left of the display (position (0, 0)). Any error in this function leaves the display and JAM internals in an undefined state.

If the form contains display data that are too big for the physical display, they are truncated without warning. However, if there are fields that won't fit within the physical display, this function returns an error without displaying the form.

The named screen is sought first in the memory-resident form list, and if found there is displayed using sm_d_window. It is next sought in all the open screen libraries, and if found is displayed using sm_l_window. Next it is sought on disk in the current directory; then under the path supplied to sm_initcrt; then in all the paths in the setup variable SMPATH. If any path exceeds 80 characters, it is skipped. If the entire search fails, this function displays an error message and returns.

In the case of a return of -1 or -2, the previously displayed form is still displayed and may be used. Other negative return code indicate that the display is undefined; the caller should display another form before using screen manager functions. The return code -2 typically means that the named screen does not exist; however, it may occur because the maximum allowable number of files is already open.

This function should be called by JAM applications only under unusual circumstances, as it does not update the control stack. You should execute a control string to display the form instead.

RETURNS

0 if no error occurred; -1 if the screen file's format is incorrect; -2 if the form cannot be found; -4 if, after the screen has been cleared, the form cannot be successfully displayed because of a read error; -5 if, after the screen was cleared, the system ran out of memory; -7 if the screen was larger than the physical display, and there were fields that would have fallen outside the display.

VARIANTS AND RELATED FUNCTIONS

```
sm_r_at_cur (name);
sm_r_window (name, line, column);
```

EXAMPLE

```
#include "smdefs.h"
#include <setjmp.h>

/* If an abort condition exists, read in a special
 * form to handle that condition, discarding all open
 * windows. */

extern jmp_buf re_init;

if (sm_isabort (ABT_OFF) > 0)
{
     sm_r_form ("badstuff");
     if (sm_query_msg ("Do you want to continue?") == 'y')
          longjmp (re_init);
     else sm_cancel ();
}
```

NAME

    sm_r_window - display a window at a given position

SYNOPSIS

```
int sm_r_window (screen_name, start_line,
    start_column)
char *screen_name;
int start_line, start_column;
```

DESCRIPTION

Displays screen_name with its upper left-hand corner at the specified line and column. The line and column are counted from zero: if start_line is 1, the form is displayed starting at the second line of the screen.)

Whatever part of the display the new window does not occupy will remain visible. However, only the top most window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the top most, or current, window. (See sm_wselect for a way to shuffle windows.)

If the window will not fit on the display at the location you request, JAM will adjust its starting position. If the window would hang below the screen and you have placed its upper left-hand corner in the top half of the display, the window is simply moved up; but if your starting position is in the bottom half of the screen, the lower left hand corner of the window is placed there. Similar adjustments are made in the horizontal direction.

If, after adjustment, the window contains display data that won't fit on the display, it is brought up anyway, without the extra. But if any field won't fit, display of the window is aborted and an error code returned.

JAM provides the control string mechanism for opening, closing, and keeping track of windows. If your code calls this function instead of executing a window control string, certain features of the JAM run-time system will not work as expected, particularly the EXIT key.

RETURNS

    0 if no error occurred during display of the screen; -1 if the screen file's format is incorrect; -2 if the form cannot be found; -3 if the system ran out of memory but the previous screen was restored; -7 if the screen was larger than the physical display, and there were fields that would have fallen outside the display.

VARIANTS AND RELATED FUNCTIONS

```
sm_r_at_cur (name);
sm_r_form (name);
sm_close_window ();
```

EXAMPLE

```
#include "smdefs.h"

/* The following is a horribly inefficient (but simple) hack
 * to bring a window up centered on the display. It reads
 * the thing in once to find out how big it is (with display
 * turned off), then reads it in again in the right place.
 */

int center_win (window)
char *window;
{
     int start_line, start_column;

     sm_do_not_display = 1;
     if (sm_r_at_cur (window))
          return -1;
     /* Compute offsets. */
     start_line = (sm_nlines - (sm_eline - sm_stline)) / 2;
     start_column = (sm_ncolms - (sm_ecolm - sm_stcolm)) / 2;
     sm_close_window ();

     sm_do_not_display = 0;
     return sm_r_window (window, start_line, start_column);
}
```

NAME

sm_rd_part - read part of a data structure to the screen

SYNOPSIS

```
#include "smdefs.h"

void sm_rd_part (screen_struct, first_field, last_field, language)
char *screen_struct;
int first_field, last_field, language;
```

DESCRIPTION

This function copies a data structure in memory to the screen, for all fields between first_field and last_field. An array and its scrolling items will be copied only if the first element falls between first_field and last_field.

The address of the structure is in screen_struct; note that this is a structure for the whole screen, not just the part of interest. There is a utility, JYACC f2struct, that will automatically generate such a structure from the screen file.

The argument language stands for the programming language in which the structure is defined; it controls the conversion of string and numeric data. Zero stands for C with null-terminated strings, one for C with blank-filled strings.

The structure being read may have been filled in previously by a call to sm_wrtstruct or sm_wrt_part, using the same values of screen_structure and language; or your application can fill it in. Using an uninitialized structure, using an inconsistent value for language, or not terminating strings properly can cause sm_rdstruct to lie or crash.

If your screen is so designed that (for instance) the input and output fields are grouped together, this function can be much faster than sm_rdstruct, which copies every field.

VARIANTS AND RELATED FUNCTIONS

```
sm_wrt_part (screen_structure, first_field, last_field, language);
sm_rdstruct (screen_structure, byte_count, language);
```

EXAMPLE

Refer to sm_wrt_part for a rather lengthy example.

NAME

sm_rdstruct - copy data from a structure to the screen

SYNOPSIS

```
#include "smdefs.h"

void sm_rdstruct (screen_struct, byte_count,
     language)
char *screen_struct;
int *byte_count;
int language;
```

DESCRIPTION

This function copies a data structure in memory to the screen, converting individual items as appropriate.

The address of the structure is in screen_struct. There is a utility, JYACC f2struct, that will automatically generate such a structure from the screen file.

The argument byte_count is the address of an integer variable. sm_rdstruct will store there the number of bytes copied from the structure.

The argument language stands for the programming language in which the structure is defined; it controls the conversion of string and numeric data. Zero stands for C with null-terminated strings, one for C with blank-filled strings.

The structure being read may have been filled in previously by a call to sm_wrtstruct, using the same values of screen_structure and language; or your application can fill it in. Using a partially uninitialized structure, using an inconsistent value for language, or not terminating strings properly can cause sm_rdstruct to lie or crash.

VARIANTS AND RELATED FUNCTIONS

```
sm_wrtstruct (screen_structure, byte_count, language);
sm_rd_part (screen_structure, first_field, last_field, language);
```

EXAMPLE

Please refer to sm_wrtstruct for an extended example.

NAME

sm_rescreen - refresh the data displayed on the screen

SYNOPSIS

```
void sm_rescreen ();
```

DESCRIPTION

Repaints the entire display from JAM's internal screen and attribute buffers. Anything written to the screen by means other than JAM library functions wil be erased. This function is normally bound to the RESCREEN key and executed automatically within sm_getkey.

You may need to use this function after doing screen I/O with the flag sm_do_not_display turned on, or after escaping from an JAM application to another program (see sm_leave). If all you want is to force writes to the display, use sm_flush.

VARIANTS AND RELATED FUNCTIONS

```
sm_flush ();
sm_return ();
```

EXAMPLE

```
/* Mess the screen up good and proper, then restore it
 * with a call to sm_rescreen. */

for (i=1; i<30; i++)
{
     printf("******************************************");
     printf("******************************************\n");
}

sm_rescreen();
sm_err_reset("Verify that the screen has been restored.");
```

NAME

sm_resetcrt - reset the terminal to operating system default state

SYNOPSIS

```
     void sm_resetcrt ();
```

DESCRIPTION

This function resets terminal characteristics to the operating system's normal state. This function should be called when leaving the screen manager environment, as before program exit.

It frees all the memory associated with the display and open screens. However, the buffers holding the message file, key translation file, etc. are not released; a subsequent call to sm_initcrt will find them in place. It then clears the screen and turns on the cursor, transmits the RESET sequence defined in the video file, and resets the operating system channel.

This function is called automatically during JAM exit, and should not be called by application programs except in case of abnormal termination.

VARIANTS AND RELATED FUNCTIONS

```
     sm_leave ();
     sm_cancel ();
```

EXAMPLE

```
/* If an effort to read the first form results in
 * failure, clean up the screen and leave. */

if (sm_r_form ("first") < 0)
{
     sm_resetcrt ();
     exit (1);
}
```

NAME

sm_resize - dynamically change the size of the display

SYNOPSIS

```
int sm_resize (rows, columns)
int rows;
int columns;
```

DESCRIPTION

This function enables you to change the size of the display used by JAM from the default defined by the LINES and COLMS entries in the video file. It makes it possible to use a single video file in a windowing environment, with FORMAKER applications being run in different sized windows; each application can set its display size at run time. It can also be used for switching between normal and compressed modes (e.g. 80 and 132 columns on VT100-compatible terminals).

All screens brought up following a call to sm_resize must fit within the display rectangle it defines; if that rectangle is larger than the physical display, the results will be unpredictable. You may specify at most 255 rows or columns.

This function clears the physical and logical screens; any displayed forms or windows, together with data entered on them, are lost.

RETURNS

-1 if a parameter was less than 0 or greater than 255; 0 if successful. Program exit on memory allocation failure. -2 if the current form or window exceeds the size specified for it.

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"
#include "smvideo.h"
#define WIDTH_TOGGLE PF9

/* Somewhat irregular code to switch a VT-100 between
 * 80- and 132-column mode by pressing PF9. */

switch (sm_openkeybd ())
{
...
case WIDTH_TOGGLE:
     if (sm_video[V_COLMS] == 80)
     {
          printf ("\033[?3h");
          sm_resize (sm_video[V_LINES], 132);
     }
     else
     {
          printf ("\033[?3l");
          sm_resize (sm_video[V_LINES], 80);
     }
     break;
...
}
```

NAME

sm_restore_data - restore previously saved data to the screen

SYNOPSIS

```
int sm_restore_data (buffer)
char *buffer;
```

DESCRIPTION

Restores all data items, both onscreen and offscreen, to the current screen from an area initialized by sm_save_data. Buffer is the address of the area. Passing an address not returned by sm_save_data, or attempting to restore to a screen other than the one saved, can produce unpredictable results.

Data items are stored in the save-data buffer as null-terminated character strings. The contents of a scrollable field or array is preceded by 2 bytes giving the total number of items saved (high order byte first); each item is preceded by two bytes of display attribute, and followed by a null. There is an additional null following all the scrolling data. The whole area is preceded by an unsigned integer giving its size. If you are programming in C, you can access it by

```
length = ((unsigned int *)buffer)[-1];
```

RETURNS

-1 if an error occurred, usually memory allocation failure; 0 otherwise.

NAME

     sm_return - prepare for return to JAM application

SYNOPSIS

```
     void sm_return ();
```

DESCRIPTION

This routine should be called upon returning to a JAM application after a temporary exit.

It sets up the operating system channel and initializes the display using the SETUP string from the video file. It does not restore the screen to the state it was in before sm_leave was called; use sm_rescreen to accomplish that, if desired.

VARIANTS AND RELATED FUNCTIONS

```
     #include "smdefs.h"

     sm_leave ();
     sm_resetcrt ();
```

EXAMPLE

```
#include "smdefs.h"

/* Escape to the UNIX shell for a directory listing */

sm_leave ();
system ("ls -l");
sm_return ();
sm_c_off ();
sm_d_msg_line ("Hit any key to continue", BLINK | WHITE);
sm_getkey ();
sm_d_msg_line ("", WHITE);
sm_rescreen ();
```

NAME

sm_rmformlist - empty out the memory-resident form list

SYNOPSIS

```
     void sm_rmformlist ();
```

DESCRIPTION

This function erases the memory-resident form list established by sm_formlist, and releases the memory used to hold it. It does not release any of the memory-resident screens themselves. Calling this function will prevent sm_r_window and related functions from finding memory-resident screens.

VARIANTS AND RELATED FUNCTIONS

```
     sm_formlist (ptr_to_form_list);
```

EXAMPLE

```
/* Hide all the memory-resident forms, perhaps
 * because the disk versions are being modified. */

sm_rmformlist ();
```

NAME

```
     sm_rrecord - read data from a structure to the screen or local data
                  block
```

SYNOPSIS

```
     #include "smdefs.h"

     void sm_rrecord (structure_ptr, record_name, byte_count, lang);
     char *structure_ptr;
     char * record_name;
     int *byte_count;
     int lang;
```

DESCRIPTION

When a data dictionary containing records is run through the dd2struct utility, structure definitions based on the fields of each record in the data dictionary are saved in a file with the record name plus a language-specific extension. Including this file in an application allows declarations of objects of these structure types. Such objects must be declared for sm_rrecord and sm_wrecord to be used.

The argument structure_ptr is the address of one such declared structure. The argument record_name is the name of the data dictionary record, needed for looking up its attributes.

The argument byte_count is a pointer to an integer. Upon return from sm_rrecord, the value contained in the integer will be the number of bytes or characters read from or written to the structure. It will be 0 if an error occured.

The argument lang is the language number, as defined in smdefs.h . Zero stands for C with null-terminated strings, one for C with blank-filled strings.

sm_rrecord reads a structure into fields on the screen, or, if the field is not on the screen, of the local data block. The fields involved are those contained in the corresponding data dictionary record. If a structure element is of a numeric type, such as integer or floating point, it is first converted into a character array. Because the field names are available in the data dictionary record definition, they may be selected arbitrarily; with sm_rdstruct, on the other hand, the structure must contain the entire screen.

The structure being read should have been filled in previously by a call to sm_wrecord, using the same values of structure_ptr, record_ptr, and lang. Or, the application can access the elements of the structure directly, and fill in data prior to a call of sm_rrecord. Failure to use the same values of structure_ptr and lang, or not terminating strings properly when accessing the structure directly, may cause sm_rrecord to lie or crash.

VARIANTS AND RELATED FUNCTIONS

```
     sm_wrecord (structure_ptr. record_name, byte_count, lang);
```

NAME

sm_rs_data - restore saved data to some of the screen

SYNOPSIS

```
void sm_rs_data (first_field, last_field, buffer);
int first_field;
int last_field;
char *buffer;
```

DESCRIPTION

Restores all data items, both onscreen and offscreen, to the fields between first_field and last_field from an area initialized by sm_sv_data. The address of the area is in buffer.

See sm_sv_data to create a buffer for subsequent retrieval by this function. If the range of fields passed to this function does not match that passed to sm_sv_data, or buffer is not a value returned by that function, grievous errors will probably occur.

The format of the data area is explained briefly under sm_restore_data.

RETURNS

-1 if an error occurred, usually memory allocation failure; 0 otherwise.

EXAMPLE

NAME

sm_rscroll - scroll an array or parallel arrays

SYNOPSIS

```
int sm_rscroll (field_number, req_scroll)
int field_number;
int req_scroll;
```

DESCRIPTION

This function scrolls an array or set of parallel arrays by req_scroll items. If req_scroll is positive, the array scrolls down (towards the bottom of the data); otherwise, it scrolls up. It supersedes the Release 3 function sm_scroll.

The function returns the actual amount scrolled. This could be the amount requested; a smaller value, if the requested amount would bring the array past its beginning or end; or zero, if the array was at its beginning or end, or an error occurred. It is never negative.

RETURNS

See above.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_rscroll (field_name, req_scroll);
sm_t_scroll (field_number);
sm_ascroll (field_number, occurrence);
```

EXAMPLE

```
#include "smdefs.h"

/* Find the number of the scrolling item under the cursor,
 * and scroll down to the next higher multiple of 5. */

int thisn;

thisn = sm_occur_no ();
sm_rscroll (sm_getcurno (), 5 - (thisn % 5));
```

NAME

sm_s_val - validate the current screen

SYNOPSIS

```
int sm_s_val ();
```

DESCRIPTION

This function loops through the screen validating each field and data item, whether on or offscreen, that is not protected from validation (VPROTECT). It is called automatically from sm_openkeybd when you press the TRANSMIT key.

Validations are performed as follows. If there are no scrolling arrays or fields, the order is left to right, then top to bottom. If a scrolling array or field is encountered, and it contains the first onscreen item for that array (or field), earlier offscreen items are validated first. If it contains the last onscreen item, any later offscreen items are validated immediately after that field.

If parallel scrolling arrays (or fields) exist, there are more complications. When an offscreen item is validated, the corresponding items from parallel arrays are validated as well, from left to right. The offscreen items preceding the parallel arrays are validated before the first onscreen item of the leftmost of the parallel arrays; similarly, the offscreen items following the arrays are validated immediately after the last onscreen item of the rightmost.

Individual field validations are performed in the following order. The table also notes conditions under which validations are skipped.

Validation	Skip if valid	Skip if empty
required	y	n
must fill	y	y
regular expression	y	y
range	y	y
check-digit	y	y
date or time	y	y
table lookup	y	y
currency format	y	n*
math expresssion	n	n
exit function	n	n
jpl function	n	n

*
The currency format edit contains a skip-if-empty flag; see the Author's Guide.

If you need to force a skip-if-empty validation, make the field required. A digits-only field must contain at least one digit in order to be considered nonempty; for other character edits, any nonblank character makes the field nonempty. The currency format edit contains a skip-if-empty flag; see the Author's Guide.

If an item fails validation, the cursor is positioned to it and an error message displayed. (If the item was offscreen, its array or field is first scrolled to bring it onscreen.) This routine returns at the first error; fields past that will not be validated.

RETURNS

-1 if any field fails validation, 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_fval (field_number);
```

EXAMPLE

```
#include "smdefs.h"
#include "smkeys.h"

/* Treat the SPF1 key as transmit, for a change. */

int key;

sm_d_msg_line ("Press %KSPF1 when done.", WHITE | REVERSE);

while ((key = sm_openkeybd ()) != EXIT)
{
     if (key == SPF1)
     {
          if (sm_s_val ())
               sm_err_reset ("Please correct the mistake(s).");
          else break;
     }
}
...
```

NAME

sm_save_data - save screen contents

SYNOPSIS

```
char *sm_save_data ();
```

DESCRIPTION

The current screen's data is saved for external access or subsequent retrieval, and the address of the save area returned.

To restore the saved data, use sm_restore_data; refer to that function for a brief explanation of the save format. Use sm_sv_free to discard a save area.

RETURNS

0 if insufficient memory was available, otherwise the address of a memory area containing the screen's data.

VARIANTS AND RELATED FUNCTIONS

```
sm_restore_data (buffer);
sm_sv_data (first_field, last_field);
sm_sv_free (buffer);
```

EXAMPLE

NAME

sm_sc_max - alter the maximum number of items allowed in a scrollable field or array

SYNOPSIS

```
int sm_sc_max (field_number, new_max)
int field_number, new_max;
```

DESCRIPTION

Changes the maximum number of items allowed in field_number, and in all fields (or arrays) parallel to it. The original maximum is set when the form is created. If the desired new maximum is less than the number of items already entered into the field or array, the old maximum will remain. The maximum can decrease only if new_max is between the number of items already entered and the previous maximum.

RETURNS

The actual new maximum (see above); or 0 if the desired maximum is invalid, or if the field is not scrollable.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_sc_max (field_name, new_max);
sm_max_item (field_number);
sm_num_items (field_number);
```

EXAMPLE

```
#include "smdefs.h"
#define SCROLLNUM 7

/* When the number of items entered in a scroll exceeds
 * ten less than the maximum, increase the maximum by 100. */

int maxnow;

maxnow = sm_max_item (SCROLLNUM);
if (maxnow - sm_num_items (SCROLLNUM) < 10)
     sm_sc_max (SCROLLNUM, maxnow + 100);
```

NAME

sm_sdate - get formatted system date

SYNOPSIS

```
     #include "smdefs.h"

     char *sm_sdate (format)
     char *format;
```

DESCRIPTION

Obtains the current date from the operating system, and formats it according to format, and returns the resulting string.

You may retrieve the date format from a field, using sm_edit_ptr, or construct it by other means. Refer to the Author's Guide for a description of date formats. Refer to sm_calc for a way of getting an arbitrary date into a formatted date field.

This function shares a single static buffer with other date and time formatting functions. The formatted date returned by this function should therefore be processed quickly, or copied to a local string.

RETURNS

The current date in the specified format. Empty if the format is invalid.

EXAMPLE

```
#define FORMAT1 "MM/DD/YY"

/* Display the date and the string used to format it. */

     sm_n_putfield ("format", FORMAT1);
     sm_n_putfield ("date", sm_sdate (FORMAT1));
```

NAME

sm_setbkstat - set background text for status line

SYNOPSIS

```
     #include "smdefs.h"

     void sm_setbkstat (message, attr)
     char *message;
     int attr;
```

DESCRIPTION

The message is saved, to be shown on the status line whenever there is no higher priority message to be displayed. The highest priority messages are those passed to sm_d_msg_line, sm_err_reset, sm_quiet_err, or sm_query_msg; the next highest are those attached to a field by means of the status text option (see the JAM Author's Guide). Background status text has lowest priority.

Attr is the initial display attribute for the message, and is a combination of the following values.

Colors		Highlights	
BLACK	BLUE	BLANK	REVERSE
GREEN	CYAN	UNDERLN	BLINK
RED	MAGENTA	HILIGHT	
YELLOW	WHITE	DIM	

The background colors defined in smdefs.h (B_BLACK and so forth) are also available.

sm_setstatus sets the background status to an alternating ready/wait flag; you should turn that feature off before using this function.

Refer to sm_d_msg_line for an explanation of how to embed attribute changes and function key names into your message.

VARIANTS AND RELATED FUNCTIONS

```
     sm_setstatus (flag);
     sm_d_msg_line (message, attribute);
```

EXAMPLE

```
#include "smdefs.h"
#define PAUSE sleep (3)

/* The hierarchy of status messages. Assume the field
 * "mama" has status text reading "Mama bear", and that
 * the home field has none. */

sm_d_msg_line ("", WHITE);
sm_setstatus (0);
sm_setbkstat ("Baby bear", MAGENTA);
PAUSE;
sm_n_gofield ("mama");
PAUSE;
sm_d_msg_line ("Papa bear", BLUE | HILIGHT);
PAUSE;
sm_home ();
PAUSE;
```

NAME

sm_setstatus - turn alternating background status message on or off

SYNOPSIS

```
     void sm_setstatus (mode)
     int mode;
```

DESCRIPTION

If mode is non-zero, alternating status flags are turned on. After this call, one message (normally Ready) is displayed on the status line while the keyboard is open for input, and another (normally Wait) when it is not. If mode is zero, the messages are turned off.

The status flags will be replaced temporarily by messages passed to sm_err_reset or a related function. They will overwrite messages posted with sm_d_msg_line or sm_setbkstat.

The alternating messages are stored in the message file as SM_READY and SM_WAIT, and can be changed there. Attribute changes and function key names can be embedded in the messages; refer to sm_d_msg_line for instructions.

VARIANTS AND RELATED FUNCTIONS

```
     sm_setbkstat (message, attr);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"
#define PAUSE sleep (3)

char buf[100];

/* Tell people what you're gonna tell 'em. */
sprintf (buf, "You will soon see %s alternating with %s below.",
     sm_msg_get (SM_READY), sm_msg_get (SM_WAIT));
sm_do_region (3, 0, 80, WHITE, buf);

/* Now tell 'em. */
sm_setstatus (1);
PAUSE;               /* Shows WAIT */
sm_openkeybd ();     /* Shows READY */

/* Finally, tell 'em what you told 'em. */
sprintf (buf, "That was %s alternating with %s on the status line.",
     sm_msg_get (SM_READY), sm_msg_get (SM_WAIT));
sm_err_reset (buf);
```

NAME

sm_sh_off - determine the cursor location relative to the start of a shifting field

SYNOPSIS

```
     int sm_sh_off ();
```

DESCRIPTION

Returns the difference between the start of data in a shiftable field and the current cursor location. If the current field is not shiftable, it returns the difference between the leftmost column of the field and the current cursor location, like sm_disp_off.

RETURNS

The difference between the current cursor position and the start of shiftable data in the current field. -1 if the cursor is not in a field.

VARIANTS AND RELATED FUNCTIONS

```
     sm_disp_off ();
```

EXAMPLE

```
#include "smdefs.h"

/* Fancy test to see whether a field is shifted to the left. */

if (sm_sh_off () != sm_disp_off ())
     sm_err_reset ("Ha! You shifted!");
```

NAME

sm_size_of_array - get the number of elements in an array

SYNOPSIS

```
int sm_size_of_array (field_number)
int field_number;
```

DESCRIPTION

Returns the number of elements in the array containing field_number. A non-array field is considered to have one element.

RETURNS

0 if the field designation is invalid; 1 if the field is not an array; The number of elements in the array otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_n_size_of_array (field_name);
sm_num_items (field_number);
sm_max_occur (field_number);
```

EXAMPLE

```
#define THEFIELD 6

/* Compute the number of pages of data in a scrolling
 * array, where a page is one onscreen-array-full. */

int pages, elements;

elements = sm_size_of_array (THEFIELD);
pages = (sm_num_items (THEFIELD) + elements - 1) / elements;
```

NAME

     sm_smsetup - initalize table of setup variables, and execute some

SYNOPSIS

```
     int sm_smsetup (memfile)
     char memfile[];
```

DESCRIPTION

This function loads a file or files of setup variables into memory, and uses some of them to set various screen manager options. It is called automatically at screen manager start-up, by sm_initcrt. The file can be either disk- or memory-resident. A complete list of setup variables can be found in the section on the setup file, in the Configuration Guide. You may find using a setup file to be more flexible and convenient than calling many of the option-setting routines in the library.

If the argument memfile is nonzero, this function uses it alone, in place of the system SMVARS file. If there is an SMSETUP in the memory-resident file or the system environment, it will be used too. If memfile is zero, the two files pointed to by SMVARS and SMSETUP are read in.

There is another function, sm_unsetup, that restores all the options affected by this routine to their default values.

RETURNS

VARIANTS AND RELATED FUNCTIONS

```
     sm_unsetup ();
```

EXAMPLE

```
/* Install a memory-resident setup file. */
extern char memsetup[];

if (sm_smsetup (memsetup) < 0)
     sm_cancel ();
```

NAME

sm_stime - get formatted system time

SYNOPSIS

```
     #include "smdefs.h"

     char *sm_stime (format)
     char *format;
```

DESCRIPTION

Obtains the current time from the operating system, formats it according to format, and returns the resulting string. The format can be obtained from a time field by calling sm_edit_ptr, or you can construct it by other means.

See the Author's Guide for a description of recognized time formats.

This function shares a single static buffer with other date and time formatting functions. The formatted time returned by this function should therefore be processed quickly, or copied to a local string.

RETURNS

The time of day in the specified format.

EXAMPLE

```
#define FORMAT1 " HH:MM:SS"

/* Print the time, and why it looks the way it does. */

     sm_n_putfield ("format" ,FORMAT1);
     sm_n_putfield ("time", sm_stime (FORMAT1));
```

NAME

sm_strip_amt_ptr - strip amount editing characters from a string

SYNOPSIS

```
#include "smdefs.h"

char *sm_strip_amt_ptr (field_number, inbuf)
int field_number;
char *inbuf;
```

DESCRIPTION

Strips all non-digit characters from the string, except for an optional leading minus sign and decimal point. If inbuf is nonzero, field_number is ignored and the passed string is processed in place.

If inbuf is zero, the contents of field_number are used. This function shares with several others a pool of buffers where it stores returned data. The value returned by any of them should therefore be processed quickly or copied.

RETURNS

A pointer to a buffer containing the stripped text, or 0 if inbuf is 0 and the field number is invalid.

VARIANTS AND RELATED FUNCTIONS

```
sm_dblval (field_number);
sm_amt_format (field_number, string);
```

EXAMPLE

```
#include "smdefs.h"

char *strip_text;

strip_text = sm_strip_amt_ptr (0, "$1,123,456");
if (strcmp (strip_text, "1123456") != 0)
     punt ("Bug in strip_amt_ptr");
```

NAME

 #COMMENT(NOT PYRCOB) sm_sv_data - save partial screen contents

SYNOPSIS

```
#include "smdefs.h"

char *sm_sv_data (first_field, last_field)
int first_field;
int last_field;
```

DESCRIPTION

The current form's data, from all fields numbered from first_field to last_field, is saved for external access or subsequent retrieval, and the address of the save area returned. Use sm_rs_data to restore it.

See sm_restore_data for the save format.

RETURNS

The address of an area containing the saved data. 0 if the current screen has no fields, or sufficient free memory is not available.

VARIANTS AND RELATED FUNCTIONS

```
sm_rs_data (first_field, last_field, buffer);
sm_save_data ();
sm_sv_free (buffer);
```

NAME

sm_sv_free - free a save-data buffer

SYNOPSIS

```
void sm_sv_free (buffer)
char *buffer;
```

DESCRIPTION

The save area at buffer, which must have been created by sm_save_data or sm_sv_data, is released and is no longer accessible.

sm_save_data and related functions record the addresses of save areas, and after ten have been accumulated the oldest are released automatically; calls to this function are therefore not strictly necessary.

NAME

```
     sm_t_scroll - test whether field can scroll
```

SYNOPSIS

```
     int sm_t_scroll (field_number)
     int field_number;
```

DESCRIPTION

Returns 1 if the field in question is scrollable, and 0 if not or if there is no such field.

RETURNS

     1 if field exists and scrolls; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_t_shift (field_number);
```

EXAMPLE

```
/* If the current field is scrolling, set it back to
 * the beginning. */

if (sm_t_scroll (sm_getcurno ())
     sm_ascroll (sm_base_fldno (sm_getcurno ()), 1);
```

NAME

sm_t_shift - test whether field can shift

SYNOPSIS

```
int sm_t_shift (field_number)
int field_number;
```

DESCRIPTION

Returns 1 if the field in question is shiftable, and 0 if not or if there is no such field.

RETURNS

1 if field is shifting. 0 if not shifting or no such field

VARIANTS AND RELATED FUNCTIONS

```
sm_t_scroll (field_number);
```

EXAMPLE

```
#include "smdefs.h"

/* Turn on shifting indicators if the screen contains any
 * shifting fields. */

int f;

for (f = 1; f <= sm_numflds; ++f)
{
     if (sm_t_shift (f))
     {
          sm_ind_set (IND_SHIFT);
          sm_rescreen ();
          break;
     }
}
```

NAME

sm_tab - move the cursor to the next unprotected field

SYNOPSIS

```
void sm_tab ();
```

DESCRIPTION

If the cursor is in a field with a next-field edit and one of the fields specified by the edit is unprotected, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is advanced to the first enterable position of the next unprotected field on the form.

The first enterable position is the leftmost in a left-justified field and the rightmost in a right-justified field; furthermore, in a digits-only field, punctuation characters are skipped.

Unlike the TAB key, this function does not cause field exit processing to be performed. To simulate a TAB keystroke, see below.

EXAMPLE

```
#include "smkeys.h"

/* This moves the cursor to the next field. */
sm_tab ();

/* This moves the cursor to the next field, validating
 * the current one first. */
sm_ungetkey (TAB);
```

NAME

     sm_tst_all_mdts - find first modified item

SYNOPSIS

```
     int sm_tst_all_mdts (occurrence)
     int *occurrence;
```

DESCRIPTION

This routine tests the MDT bits of all occurrences of all fields on the current screen, and returns the field and occurrence numbers of the first item with its MDT set, if there is one. The MDT bit indicates that an item has been modified, either from the keyboard or by the application program, since the screen was displayed (or since its MDT was last cleared by sm_op_mdt).

This function returns zero if no items have been modified. If one has been modified, it returns the field number, and stores the occurrence number in the variable addressed by occurrence.

RETURNS

     0 if no MDT bit is set anywhere on the screen; The number of the first field on the current screen for which some occurrence has its MDT bit set. In this case, the number of the first occurrence with MDT set is returned in the reference parameter occurrence.

VARIANTS AND RELATED FUNCTIONS

```
     sm_op_mdt (field_number, operation);
```

EXAMPLE

```
#include "smdefs.h"

/* Clear MDT for all fields on the form; then write
 * data to the last field, and check that its MDT is
 * the first one set. */

int occurrence;

sm_cl_all_mdts();
sm_putfield (sm_numflds, "Hello");
if (sm_tst_all_mdts (&occurrence) != sm_numflds)
     sm_err_reset ("Something is rotten in the state of Denmark.");
```

NAME

sm_u_async - asynchronous keyboard input hook

SYNOPSIS

```
#include "smdefs.h"

extern struct fnc_data *sm_u_async;

int my_async (interval)
int interval;
```

DESCRIPTION

The function is called only when the keyboard is being read, and only if a keystroke does not arrive within the time limit given at installation. That timeout is in tenths of a second, although its real value depends on the granularity of your system's clock; it is placed in the intrn_use member of the fnc_data structure. See sm_install for more about installation.

The asynchronous function is called from sm_gtchar, one level below sm_getkey. If it returns zero, everything proceeds as before. If it returns -1, sm_gtchar goes directly to the keyboard for a character, and does not call the asynchronous function again until it gets one (and is asked for another). Any other value is passed back to sm_getkey.

The authoring utility uses an asynchronous function to update its cursor position display. Another typical use might be to implement a real-time clock display.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

See above.

VARIANTS AND RELATED FUNCTIONS

```
sm_install (which_hook, what_funcs, howmany);
```

EXAMPLE

```
/* A typical use of the window selection routines is to
 * update information to a window that may (or may not) be
 * covered.  For example, suppose that the current time
 * should be maintained on the underlying form.  Assume
 * that a field named "curtime" exists on that form.
 * The following code fragments can be used
 * to maintain that field independent of the number of windows
 * currently open above the form.
 */

#include "smdefs.h"

updatetime()
{
     sm_wselect (0); /* quietly select the bottom form */
     sm_n_putfield ("curtime", ""); /* update system time display */
     sm_wdeselect ();/* restore visible window */
     sm_flush ();
     return (0);
}

/* In initialization code: called every second. */

     static struct fnc_data afunc = { 0, updatetime, 0, 10, 0, 0 };
     sm_install (ASYNC_FUNC, &afunc, (int *)0);
```

NAME

sm_u_avail - playback character availability hook

SYNOPSIS

```
#include "smdefs.h"

extern struct fnc_data *sm_u_avail;

int my_avail (interval)
int interval;
```

DESCRIPTION

This hook is called from sm_keyhit, q.v. That function's mission is to find if there's a key waiting to be read; before looking at the physical keyboard, it checks for avavilable playback characters by calling this function. The latter should return a positive value if there are characters available for playback (by sm_u_play), and zero if there is nothing to play back. If the playback system is inactive, it should return -1, and sm_keyhit will go on to poll the keyboard. This hook is also called from sm_getkey; if it returns a positive value, the latter then calls sm_u_play to retrieve a key.

The argument interval is the length of time, in tenths of a second, that sm_keyhit is supposed to wait for a key. You can use this to simulate a realistic rate of typing, by pausing in this function.

Along with sm_u_play and sm_u_record, this function forms part of a keystroke recording and playback package. Such a package can be quite useful in regression testing and performance analysis of JAM applications, because it enables you to reproduce a series of inputs exactly and with little effort.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

1 (or other positive value) if there are characters to be played back; 0 if the playback system is active but there are no characters available; -1 if the playback system is inactive.

VARIANTS AND RELATED FUNCTIONS

```
sm_keyfilter (flag);
sm_u_play ();
sm_u_record (key);
```

EXAMPLE

See sm_u_play for a detailed example of a keystroke recording package.

NAME

```
     sm_u_ckdigit - check digit validation hook
```

SYNOPSIS

```
     #include "smdefs.h"

     extern struct fnc_data *sm_u_ckdigit;

     int myckdigit (field, data, item, modulus, digits);
     int field, item, modulus, digits;
     char *data;
```

DESCRIPTION

This hook is called by field validation. It verifies that data contains the required minimum number of digits, terminated by the proper check digit. If not, it positions the cursor to the indicated occurrence and posts an error message before returning. It also be used to check any character string, or any field. If data is null, the string to check is obtained from the field and occurrence number, and the error message is displayed if that string is bad. If field_number is zero, no message will be posted, but the function's return code will indicate whether the string passed its check.

The source code to sm_ckdigit is included with JAM; refer to it for descriptions of our check digit algorithms and how to implement your own. Note that the parameter digits specifies the minimum number of digits, not the check digit. If you decide to modify that module without renaming it, you do not need to call sm_install; JAM already does that.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

```
     0 if the number contains the minimum number of digits and the proper check
          digit; -1 for a present but invalid string; -2 if no string is
          supplied and the field or occurrence number is out of range.
```

VARIANTS AND RELATED FUNCTIONS

```
     sm_ckdigit (field, data, item, modulus, digits);
     sm_install (which_hook, what_func, howmany);
```

EXAMPLE

```
#include "smdefs.h"

/* Validate a check digit in field 1 directly. The
 * routine itself will display an error message if the
 * validation fails. */

if (sm_ckdigit(1, (char *)0, 0, 10, 2))
     sm_gofield (1);
```

NAME

```
     sm_u_inscrsr - insert/overstrike mode switch hook
```

SYNOPSIS

```
     #include "smdefs.h"

     extern struct fnc_data *sm_u_inscrsr;

     int myinscrsr (insert_mode)
     int insert_mode;
```

DESCRIPTION

The JAM library calls a user-defined function while switching from insert to overstrike mode (or vice versa). The intention is to allow for a visible indication of the mode. Insert_mode is 1 if JAM is entering insert mode, and 0 if it is entering overstrike mode.

Your function must be installed via a call to sm_install, q.v. It will be called by sm_getkey when the latter reads the INSERT key from the keyboard. If the INSON and INSOFF entries are present in the video file, JAM will send them in addition to calling this function; if all you want is a change of cursor style, you can get away with that.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

VARIANTS AND RELATED FUNCTIONS

```
     sm_install (which_hook, what_function, howmany);
```

EXAMPLE

```
#include "smdefs.h"

/* The following function is installed on the "change
 * insert mode" hook. It modifies an area at the lower right-
 * hand corner of the screen to show whether the current
 * mode is insert or overstrike. */

int insf (insert_on)
int insert_on;
{
     if (insert_on)
          sm_do_region (sm_nlines - 1, sm_ncolms - 4, 3,
               REVERSE | WHITE, "INS");
     else
          sm_do_region (sm_nlines - 1, sm_ncolms - 4, 3,
               REVERSE | WHITE, "OVR");
     return 0;
}

/* Installation code for the above. */

struct fnc_data insf = {
     "mycurs", mycurs, 0, 0, 0, 0
};

sm_install (INSCRSR_FUNC, &insf, (int *)0);
```

NAME

sm_u_keychg - logical key translation hook

SYNOPSIS

```
#include "smdefs.h"

extern struct fnc_data *sm_u_keychg;

int mykeychg (input_key)
int input_key;
```

DESCRIPTION

This is a user-installable function called from sm_getkey. You can install your own by calling sm_install with KEYCHG_FUNC.

The argument to this function is a logical key read by sm_getkey; you may put special key processing here. If your function returns zero, the key is removed from the input stream, and sm_getkey reads in the next one without returning to its caller. If your function returns any other value, sm_getkey returns that directly to its caller.

See sm_u_play for an additional example of a keychange function.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

as described above

VARIANTS AND RELATED FUNCTIONS

```
sm_install (which_hook, what_func, howmany);
sm_getkey ();
```

EXAMPLE

```
/* What follows is an extremely simple-minded keyboard
 * macro implementation, using a keychange function.
 * Any character following a backslash \ is looked up in a
 * hard-coded table, and the string corresponding to
 * that character (if any) substituted. If the character isn't
 * there, it beeps. The expansion string is pushed back
 * onto the input using sm_ungetkey.
 */

#include "smdefs.h"

int macro ();
void try_expand ();

/* Macro expansion table. Will expand \j to "JYACC, Inc."
 * and so forth. The zero tag marks the end of the table. */

static struct macrotab {
     char tag;
     char *expansion;
} macrotab[] = {
     'j', "JYACC, Inc.",
     'v', "Version 4.0",
     'a', "116 John St., New York, NY, 10038",
     0,   ""
};

/* This is the keychange function. It looks out for a
 * backslash, and when it gets one sets a flag to try
 * expanding the next character it gets. Both the backslash
 * and the following character are deleted from the input
 * stream. The actual expansion is done in a subroutine.
 */

int macro (key)
int key;
{
#define ESCAPE '\\'
     static int saw_escape_last_time;

     if (saw_escape_last_time)
     {
          try_expand (key);
          saw_escape_last_time = 0;
          return 0;
     }
     else if (key == ESCAPE)
     {
          saw_escape_last_time = 1;
          return 0;
     }
     else return key;
}

/* This function looks up 'key' in the table, and if it
 * finds it there, pushes the expansion onto the input.
 * Note that the expansion must be pushed backwards.
 */

void try_expand (key)
int key;
{
     struct macrotab *m;
     char *p;

     for (m = macrotab; m->tag != 0; ++m)
     {
          if (m->tag == key)
          {
               p = m->expansion + strlen (m->expansion) - 1;
               while (p >= m->expansion)
                    sm_ungetkey (*p--);
               return;
          }
     }

     sm_bel ();     /* Could not expand */
}

/* Finally, here is code to install 'macro' as a
 * keychange function, using sm_install. */

static struct fnc_data keychg = {
     "macro", macro, 0, 0, 0, 0
};

sm_install (KEYCHG_FUNC, &keychg, (int *)0);
```

NAME

sm_u_play - keystroke playback hook

SYNOPSIS

```
#include "smdefs.h"

extern struct fnc_data *sm_u_play;

int myplay ();
```

DESCRIPTION

This hook is called from sm_getkey. If it returns a nonzero value, sm_getkey treats it as an already translated logical key, which is returned to its caller. The key may be a standard displayable ASCII character, or one of the values defined in smkeys.h . Normally, the returned key will have been stored in some fashion by a previous call to sm_u_record; the recording algorithm is entirely up to you. Refer to sm_getkey for details about just when this and other input processing hooks are invoked.

Along with sm_u_avail and sm_u_record, this function forms part of a keystroke recording and playback package. Such a package can be quite useful in regression testing and performance analysis of JAM applications, because it enables you to reproduce a series of inputs exactly and with little effort.

When sm_getkey obtains a key from this function, it does not update the display first, as it does when reading a key from the keyboard. You may want to call sm_flush from this function or from sm_u_avail in order to keep the display in sync with the input, e.g. to have data characters echoed.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

Any ASCII character or JAM logical key code; zero if there is nothing to play back.

VARIANTS AND RELATED FUNCTIONS

```
sm_keyfilter (flag);
sm_u_record (key);
sm_u_avail (interval);
```

EXAMPLE

```
#include "smdefs.h"
#include "smerror.h"
#include "smkeys.h"

/* The code below forms a simple keystroke recording and playback
 * package, using the three hooks provided for that purpose.
 * Pressing PF2 causes following keystrokes to be saved in memory;
 * pressing PF3 stops recording keystrokes; and pressing PF4 causes
 * the list of saved keystrokes to be played back, with a one-second
 * pause between them.  These transitions are done in a keychange
 * function, which is another sm_getkey hook.
 * While either recording or playback is in progress, a small
 * display area in the upper left-hand corner tells what is going on
 * and how many keys have been processed so far.
 */

/* Assign names for the control keys. */
#define START_RECORDING   PF2
#define STOP_RECORDING    PF3
#define START_PLAYING     PF4

int play(), record(), avail(), playchange();
void status();

int  *keylist,    /* Buffer for saved keys */
     keycount,    /* Number of keys saved up */
     keysize,     /* Size of save buffer */
     playkey,     /* Index of next key to play back */
     recording,   /* Nonzero while recording */
     playing;     /* Nonzero while playing back */

/* This function is installed on the playback hook.
 * It just steps through the list of recorded keys, and
 * turns itself off when it reaches the end. */

int play ()
{
     if (! playing)
          return 0;
     else if (playkey == keycount)
     {
          playing = 0;
          status (0, 0);
          return 0;
     }
     else
     {
          status ('P', playkey + 1);
          return keylist[playkey++];
     }
}

/* This function is installed on the record hook.
 * It save the key it is passed in a dynamic array,
 * expanding the array when it's full. */

int record (key)
{
     if (recording)
     {
          if (keycount == keysize)
               keylist = (int *)realloc(keylist,
          sizeof(int) * (keysize *= 2));
          keylist[keycount++] = key;
          status ('R', keycount);
     }
     return 0;
}

/* This function tells sm_getkey whether there's anything left
 * to play back. It also flushes the display and pauses for
 * a second, so you can see the replayed characters going by. */

int avail (interval)
{
     if (playing)
     {
          sm_flush ();
          sleep (1);
          return 1;
     }
     return 0;
}

/* This function controls all of the above, triggering on
 * the PF2, PF3, and PF4 keys. */

int playchange (key)
{
     switch (key)
     {
     case START_PLAYING:
          if (recording)
               sm_quiet_err ("You're recording.");
          else if (keycount <= 0)
               sm_quiet_err ("Nothing to play.");
          else
          {
               playing = 1;
               playkey = 0;
               status ('P', 0);
          }
          return 0;

     case START_RECORDING:
          if (playing)
               sm_quiet_err ("You're playing.");
          else if (recording)
               sm_quiet_err ("You're already recording.");
          else
          {
               keylist = (int *)malloc(sizeof(int) * (keysize = 50));
               keycount = 0;
               recording = 1;
               status ('R', 0);
          }
          return 0;

     case STOP_RECORDING:
          recording = 0;
          status (0, 0);
          return 0;

     default:
          return key;
     }
}

/* This routine places the current operation and count in the
 * upper left-hand corner of the screen. The calculation is
 * to avoid borders. */

void status (flag, count)
{
#define SWIDTH    4
     int  corner;
     char buf[10];

     corner = sm_cform->form.bord_char ? 1 : 0;
     if (flag == 0)
     {
          buf[0] = 0;
          sm_do_region (corner, corner, SWIDTH, WHITE, buf);
     }
     else
     {
          sprintf (buf, "%c%.3d", flag, count);
          sm_do_region (corner, corner, SWIDTH, REVERSE | WHITE, buf);
     }
}

/* Finally, here is code to initialize the necessary hooks. */

static struct fnc_data playf = {
     "play", play, C_FUNC, 0, 0, 0
};
static struct fnc_data recordf = {
     "record", record, C_FUNC, 0, 0, 0
};
static struct fnc_data availf = {
     "avail", avail, C_FUNC, 0, 0, 0
};
static struct fnc_data changef = {
     "playchange", playchange, C_FUNC, 0, 0, 0
};

sm_install (PLAY_FUNC, &playf, (int *)0);
sm_install (RECORD_FUNC, &recordf, (int *)0);
sm_install (AVAIL_FUNC, &availf, (int *)0);
sm_install (KEYCHG_FUNC, &changef, (int *)0);
```

NAME

```
     sm_u_record - keystroke recording hook
```

SYNOPSIS

```
     #include "smdefs.h"

     extern struct fnc_data *sm_u_record;

     void myrecord (key);
     int key;
```

DESCRIPTION

This hook is called from sm_getkey. When it has a translated key value in hand, whether read from the keyboard or obtained from another source, sm_getkey passes it to this function so that it may be recorded, and perhaps played back by a future call to sm_u_play. Values stored by sm_ungetkey are not passed to this hook, but are returned directly to the caller of sm_getkey; refer to that function's description for details about just when this and other input processing hooks are invoked.

Along with sm_u_play and sm_u_avail, this function forms part of a keystroke recording and playback package. Such a package can be quite useful in regression testing and performance analysis of JAM applications, because it enables you to reproduce a series of inputs exactly and with little effort.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

VARIANTS AND RELATED FUNCTIONS

```
     sm_keyfilter (flag);
     sm_u_play ();
     sm_u_avail (interval);
```

EXAMPLE

See sm_u_play for a detailed example of a keystroke recording package.

NAME

```
     sm_u_statfnc - status line display hook
```

SYNOPSIS

```
     #include "smdefs.h"

     extern struct fnc_data *sm_u_statfnc;

     int mystatfnc ();
```

DESCRIPTION

This function is called before the generic code JAM uses to display stuff on the status line. It is intended for use on terminals with really weird status lines indecipherable to the generic code (see the video manual in the Configuration Guide), but more imaginative uses are possible.

This function takes no arguments. The text to be displayed on the status line is always stored in the last line of the global screen data buffer, sm_screen. You can examine that line as:

```
     sm_screen[sm_nlines]
```

and have your way with it. If your function returns 0, sm_d_msg_line continues with its processing. Otherwise, sm_d_msg_line returns immediately, assuming that you have updated the status line.

jxform uses this hook to redisplay the cursor position on the screen whenever the status line is changed; see sm_c_vis. Your function must be installed by a call to sm_install, using STAT_FUNC.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

     See above.

VARIANTS AND RELATED FUNCTIONS

```
     sm_d_msg_line (message, attribtue);
     sm_c_vis (flag);
     sm_install (which_hook, what_func, howmany);
```

EXAMPLE

NAME

```
     sm_u_uinit - do application-specific things at screen manager
                  initialization time
```

SYNOPSIS

```
     #include "smdefs.h"

     extern struct fnc_data *sm_u_uinit;

     int myuinit (terminal_type)
     char terminal_type[];
```

DESCRIPTION

This function is a hook for application code at screen manager initialization time. It is called immediately on entry to sm_initcrt, which treats it as if its job were to determine the terminal type; however, it need not do that, and it may do anything else. It receives, in terminal_type, the address of a buffer where JAM stores its terminal type string; if it writes a null-terminated string there, sm_initcrt will use that instead of checking the environment.

On operating systems without an environment, this function can be used to obtain the terminal type in some system-specific way. It can be used to do initializations, but those are better done before the call to sm_initcrt.

This function must be installed via a call to sm_install, q.v., using UINIT_FUNC.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

VARIANTS AND RELATED FUNCTIONS

```
     sm_initcrt (path);
     sm_install (which_hook, what_func, howmany);
```

NAME

    sm_u_ureset - screen manager cleanup hook

SYNOPSIS

```
#include "smdefs.h"

extern struct fnc_data *sm_u_ureset;

void myureset ();
```

DESCRIPTION

sm_resetcrt calls this function just before returning; it provides a hook for application-specific cleanup code. It receives no parameters, and any return value is ignored. You should use sm_install to install it.

sm_resetcrt is normally called when the program catches an interrupt signal, or exits normally; this is therefore a good place to attach processing that needs to be done in the case of (abnormal) exit.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

VARIANTS AND RELATED FUNCTIONS

```
sm_resetcrt ();
sm_install (which_hook, what_func, howmany);
sm_cancel ();
```

NAME

sm_u_vproc - video processing hook

SYNOPSIS

```
#Include "smdefs.h"

extern struct fnc_data *sm_u_vproc;

int myvproc (operation, parameters)
int operation;
int *parameters;
```

DESCRIPTION

This function is a hook for video output. JAM defines generic video operations, and uses a configuration file to support them on a given display. It calls this function to display output, passing it a video operation code, and a vector of parameters containing zero or more integers. The operation codes are defined in smvideo.h , and are listed below.

Code	Operation	# Parameters
V_INIT	initialization string	
V_RESET	reset string	
V_ED	erase entire display & home cursor	
V_EL	erase from cursor to end of line	
V_EW	erase window to given background 5	
V_REPT	repeat character sequence	2
V_CON	turn cursor on	
V_COF	turn cursor off	
V_INSON	set insert cursor style	
V_INSOFF	set overstrike cursor style	
V_SCP	save cursor position	
V_RCP	restore cursor position	
V_CUP	absolute cursor position	2
V_CUU	cursor up	1
V_CUD	cursor down	1
V_CUF	cursor forward	1
V_CUB	cursor back	1
V_SGR	set latch graphics rendition	11
V_ASGR	set area graphics rendition	11
V_ARGR	remove area attribute	
V_OMSG	open message line	
V_CMSG	close message line	
V_KSET	write to softkey label	2
V_MODE0	set graphics mode 0 (likewise V_MODE1, 2, 3)	
V_MODE4	single-character graphics mode (likewise V_MODE5, 6)	
BELL	visible alarm sequence	

This hook enables you to add special processing for standard video operations, should you wish to replace or extend them. If you have severe memory constraints, you can use sm_u_vproc to hard-code all video processing and eliminate the device-independent video package entirely, thus saving substantial space. Not all codes defined there will be passed to this routine, just those that actually produce output, as are shown in the table.

Your video function must be installed with a call to sm_install, using VPROC_FUNC. If it returns a nonzero value, JAM's run-time system will process the operation normally, so you need implement only those operations you are really interested in.

Note: you may call the function you supply for this hook anything you like; it is included in the manual under this head only for definiteness. In fact, the name on this page is one you should not call your function, because it will conflict with the global variable used by JAM to store your function's address.

RETURNS

Zero if the parameters were successfully processed, nonzero otherwise.

VARIANTS AND RELATED FUNCTIONS

```
     sm_install (which_hook, what_func, howmany);
```

NAME

sm_ch_umsgatts - change the standard error windows attributes

SYNOPSIS

```
void sm_ch_umsgatt (border_style, border_attribute,
     protected_attribute, menu_attribute)
     int border_style;
     int border_attribute;
     int protected_attribute;
     int menu_attr;
```

DESCRIPTION

Changes the display characteristics of the error windows that are part of the library. Currently, there are two such windows: the error window, used to display a message too long to fit on the status line; and the hit space window, which pops up if you hit any key other than the space bar to acknowledge an error message.

This function is intended to be called once, at the beginning of an application, to set the display characteristics of the library windows to harmonize with the application's own forms.

This function is similar to sm_ch_form_atts, which changes the attributes of the system calls and "go to" windows as well as those of the error windows. Use of that function is therefore preferable.

If border_style is less than 0, the windows are made borderless. Otherwise, it is taken to be the border style number (0 through 9), and border_attribute, if nonzero, is made the border attribute.

If protect_attribute is nonzero, it is used for protected fields that contain messages in the error windows. menu_attribute is not currently used.

NAME

sm_ungetkey - push back a translated key on the input

SYNOPSIS

```
     #include "smkeys.h"

     int sm_ungetkey (key);
     int key;
```

DESCRIPTION

Saves the translated key given by key so that it will be retrieved by the next call to sm_getkey. Multiple calls are permitted; the key values are pushed onto a stack (LIFO).

When sm_getkey reads a key from the keyboard, it flushes the display first, so that the operator sees a fully updated display before typing anything. Such is not the case for keys pushed back by sm_ungetkey; since the input is coming from the program, it is responsible for updating the display itself.

RETURNS

The value of its argument, or -1 if memory for the stack is unavailable.

VARIANTS AND RELATED FUNCTIONS

```
     sm_getkey ();
```

EXAMPLE

```
#include "smkeys.h"

/* Force tab to next field */
sm_ungetkey (TAB);
```

NAME

sm_unprotect - completely unprotect a field

SYNOPSIS

```
int sm_unprotect (field_number)
int field_number;
```

DESCRIPTION

Removes all four kinds of protection (see table) from the field indicated by field_number.

Mnemonic	Meaning
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into (or from entering via any other key)
CPROTECT	protect from clearing
VPROTECT	protect from validation routines

To unprotect a field selectively, use sm_1unprotect; to unprotect an array as a unit, use sm_aunprotect.

RETURNS

-1 if the field is not found; 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_e_unprotect (field_name, element);
sm_n_unprotect (field_name);
sm_protect (field_number);
sm_1unprotect (field_number, mask);
sm_aunprotect (field_number, mask);
```

EXAMPLE

```
#include "smdefs.h"

/* If the executive has a PC, unprotect a field to
 * hold its make; otherwise, protect that field. */

if (sm_n_is_yes ("owns_pc"))
     sm_n_unprotect ("pc_make");
else sm_n_protect ("pc_make");
```

NAME

sm_unsetup - restore screen manager options to their default values

SYNOPSIS

```
void sm_unsetup ();
```

DESCRIPTION

This function calls all the option-setting library functions mentioned in the setup file with their default parameters, effectively restoring "factory defaults" to the whole library. There is a list of the functions involved in the section on setup files in the Configuration Guide.

The values read in by sm_smsetup are not erased from memory.

VARIANTS AND RELATED FUNCTIONS

```
sm_smsetup (memfile);
```

EXAMPLE

```
/* Back to defaults we go. */

sm_unsetup ();
```

NAME

sm_vinit - initialize video translation tables

SYNOPSIS

```
int sm_vinit (video_file)
char *video_file;
```

DESCRIPTION

This routine is called by sm_initcrt as part of the initialization process. It can also be called directly by an application program, with video_file the address of a memory resident video file. Such a file must be created by the JYACC vid2bin and bin2c utilities, then compiled into the application.

If video_file is zero, this function will read the binary file named by the environment variable SMVIDEO from disk.

RETURNS

0 if initialization is successful; -1 if video_file is zero and SMVIDEO is undefined; program exit if an error occurs in reading from disk.

EXAMPLE

```
/* Install a memory-resident video file */

extern char special_vid[];

sm_vinit (special_vid);
```

NAME

sm_wdeselect - restore the formerly active window

SYNOPSIS

```
int sm_wdeselect ();
```

DESCRIPTION

This function restores a window to its original position in the form stack, after it has been moved to the top by a call to sm_wselect. Information necessary to perform this feat is saved during each call to sm_wselect, but is not stacked; therefore a call to this routine must follow a call to sm_wselect, and select/deselect pairs cannot be nested.

RETURNS

-1 if there is no window to restore. 0 otherwise.

VARIANTS AND RELATED FUNCTIONS

```
sm_wselect (window);
sm_n_wselect (window_name);
```

EXAMPLE

```
/* A typical use of the window selection routines is to
 * update information to a window that may (or may not) be
 * covered.  For example, suppose that the current time
 * should be maintained on the underlying form.  Assume
 * that a field named "curtime" exists on that form.
 * The following code fragments can be used
 * to maintain that field independent of the number of windows
 * currently open above the form.
 */

#include "smdefs.h"

updatetime()
{
     sm_wselect (0); /* quietly select the bottom form */
     sm_n_putfield ("curtime", ""); /* update system time display */
     sm_wdeselect ();/* restore visible window */
     sm_flush ();
     return (0);
}

/* In initialization code: called every second. */

     static struct fnc_data afunc = { 0, updatetime, 0, 10, 0, 0 };
     sm_install (ASYNC_FUNC, &afunc, (int *)0);
```

NAME

    fnc(wrecord) - copy data from the screen or LDB to a structure

SYNOPSIS

```
     #include "smdefs.h"

     void sm_wrecord (structure_ptr, record_name, byte_count, lang);
     char *structure_ptr;
     char * record_name;
     int *byte_count;
     int lang;
```

DESCRIPTION

When a data dictionary containing records is run through the dd2struct utility, structure definitions based on the fields of each record in the data dictionary are saved in a file with the dictionary name plus a language-specific extension. Including this file (or specific structures of the file) in an application allows declarations of objects of these structure types. Such objects must be declared for sm_rrecord and sm_wrecord to be used.

The argument structure_ptr is the address of one such declared structure. The argument record_name is the name of the data dictionary record, needed for looking up its attributes.

The argument byte_count is a pointer to an integer. Upon return from sm_wrecord, the value contained in the integer will be the number of bytes or characters read from or written to the structure. It will be 0 if an error occured.

The argument lang is the language number, as defined in smsmdefs.h . Zero stands for C with null-terminated strings, one for C with blank-filled strings.

sm_wrecord reads field data from the screen if possible, or from the local data block, and fills in the appropriate elements of the structure. If a structure element is of a numeric type, the data is first converted into the appropriate representation for the machine.

VARIANTS AND RELATED FUNCTIONS

```
     sm_rrecord (structure_ptr, record_name, byte_count, lang);
```

NAME

     sm_wrt_part - write part of the screen to a structure

SYNOPSIS

```
     #include "smdefs.h"

     void sm_wrt_part (form_struct, first_field,
          last_field, language)
     char *form_struct;
     int first_field, last_field, language;
```

DESCRIPTION

This function copies the contents of all fields between first_field and last_field to a data structure in memory. An array and its scrolling items will be copied only if the first element falls between first_field and last_field.

The address of the structure is in screen_struct; it is a structure for the whole screen, not just the part of interest. There is a utility, JYACC f2struct, that will automatically generate such a structure from the screen file.

Language stands for the programming language in which the structure is defined; it controls the conversion of string and numeric data. Zero stands for C with null-terminated strings, one for C with blank-filled strings.

If your screen is so designed that (for instance) the input and output fields are grouped together, this function can be much faster than sm_wrtstruct, which copies every field.

VARIANTS AND RELATED FUNCTIONS

```
     sm_rd_part (screen_structure, first_field, last_field, language);
     sm_wrtstruct (screen_structure, byte_count, language);
```

EXAMPLE

The code example below uses the same screen as the sm_rdstruct example; refer to that example for the screen's picture and listing.

Here is a header file produced by f2struct from the screen:

```
struct strex
{
     long      date;
     char      name[26];
     char      address[3][76];
     char      telephone[14];
};
```

Finally, here is a program that processes the screen using sm_rd_part and sm_wrtpart.

```
#include "smdefs.h"
#include "smkeys.h"
#include "strex.h"

#define C_LANG    0

int main ();
void punt ();

char *program_name;
```

```
main (argc, argv)
char *argv[];
{
     struct strex
     example;
     int       key;
     char      ebuf[80];

     /* Initialize all structure members to nulls. This is
      * important because we are going to do an sm_rd_part
      * first. */
     example.date = 0L;
     example.name[0] = 0;
     example.address[0][0] = example.address[1][0] = 0;
     example.address[2][0] = 0;
     example.telephone[0] = 0;

     /* Copy command line arguments, if any, into the structure. */
     switch (argc)
     {
     case 6:
     default:
     /* Ignore extras */
          strcpy (example.telephone, argv[5]);
     case 5:
          strcpy (example.address[2], argv[4]);
     case 4:
          strcpy (example.address[1], argv[3]);
     case 3:
          strcpy (example.address[0], argv[2]);
     case 2:
          strcpy (example.name, argv[1]);
          program_name = argv[0];
          break;
     }

     /* Initialize the screen and copy the structure to it,
      * excluding the date field. */
     sm_initcrt ("");
     if (sm_r_form ("strex") < 0)
          punt ("Cannot read form.");
     sm_rd_part (&example, 2, sm_numflds, C_LANG);
     sm_n_putfield ("date", "");

     /* Open the keyboard to accept new data to the form, and
      * copy it to the structure when done. Break out when user
      * hits EXIT key. */
     sm_d_msg_line ("Enter data; press %KEXIT to quit.",
          WHITE | HILIGHT);
     do {
          key = sm_openkeybd ();
          sm_wrtstruct (&example, 2, sm_numflds, C_LANG);
          sprintf (ebuf, "Acknowledged: byte count = %d.",
               count);
          sm_err_reset (ebuf);
     } while (key != EXIT);

     /* Clear the screen and display the final structure contents. */
     sm_resetcrt ();
     printf ("%s\n", example.name);
     for (count = 0; count < 3; ++count)
          if (example.address[count][0])
               printf ("%s\n", example.address[count]);
     printf ("%s\n", example.telephone);
     exit (0);
}

void
punt (message)
char *message;
{
     sm_resetcrt ();
     fprintf (stderr, "%s: %s\n", program_name, message);
     exit (1);
}
```

NAME

     sm_wrtstruct - copy data from the screen to a memory structure

SYNOPSIS

```
     #include "smdefs.h"

     void sm_wrtstruct (form_struct, count, language)
     char *form_struct;
     int *count, language;
```

DESCRIPTION

This function copies a data structure in memory to the screen, converting individual items as appropriate. The address of the structure is in screen_struct. There is a utility, JYACC f2struct, that will automatically generate such a structure from the screen file.

The argument count is the address of an integer variable. sm_wrtstruct will store there the number of bytes copied to the structure.

The argument language stands for the programming language in which the structure is defined; it controls the conversion of string and numeric data. Zero stands for C with null-terminated strings, one for C with blank-filled strings.

VARIANTS AND RELATED FUNCTIONS

```
     sm_wrt_part (screen_structure, first_field, last_field, language);
     sm_rdstruct (screen_structure, byte_count, language);
```

EXAMPLE

The code example below uses this screen, whose picture and listing follow.

```
                  ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
                  º                                         º
                  º  Name:         _________________________   º
                  º  Address:      _________________________   º
                  º                _________________________   º
                  º                _________________________   º
                  º  Telephone: (___)___-____                 º
                  º                                         º
                  ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼


                            FORM 'strex'
                            ------------

FIELD DATA:
-----------

Field number        : 1   (line 1, column 32, length = 8)
Field name          : date
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
Field edits         : PROTECTED FROM:  ENTRY OF DATA;  TABBING INTO;  CLEARING;
                       VALIDATION;
Date field data     : SYSTEM DATE; FORMAT = MM.DD.YY
Data type           : LONG INT


Field number        : 2   (line 3, column 15, length = 25)
Field name          : name
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
```

```
Field number        : 3   (line 4, column 15, length = 25)
Field name          : address
Vertical array      : 3 elements; distance between
                      elements = 1
Array field numbers :  3  4  5
Shifting values     : maximum length = 75; increment =  1
Display attribute   : UNDERLINED HIGHLIGHTED WHITE


Field number        : 6   (line 7, column 15, length = 13)
Field name          : telephone
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
Character edits     : DIGITS-ONLY
Data type           : CHAR STRING
```

Here is a header file produced by f2struct from the screen above.

```
struct strex
{
     long      date;
     char      name[26];
     char      address[3][76];
     char      telephone[14];
};
```

Finally, here is a program that processes the screen using sm_rdstruct and sm_wrtstruct.

```
#include "smdefs.h"
#include "smkeys.h"
#include "strex.h"

#define C_LANG   0

int main ();
void punt ();

char *program_name;

main (argc, argv)
char *argv[];
{
     struct strex example;
     int       count,
               key;
     char      ebuf[80];

     /* Initialize all structure members to nulls. This is
      * important because we are going to do an sm_rdstruct
      * first. */
     example.date = 0L;
     example.name[0] = 0;
     example.address[0][0] = example.address[1][0] = 0;
     example.address[2][0] = 0;
     example.telephone[0] = 0;

     /* Copy command line arguments, if any, into the structure. */
     switch (argc)
     {
     case 6:
     default:
     /* Ignore extras */
          strcpy (example.telephone, argv[5]);
     case 5:
          strcpy (example.address[2], argv[4]);
     case 4:
          strcpy (example.address[1], argv[3]);
     case 3:
          strcpy (example.address[0], argv[2]);
     case 2:
          strcpy (example.name, argv[1]);
          program_name = argv[0];
          break;
     }

     /* Initialize the screen and copy the structure to it. */
     sm_initcrt ("");
     if (sm_r_form ("strex") < 0)
          punt ("Cannot read form.");
```

```
	sm_rdstruct (&example, &count, C_LANG);
	sm_n_putfield ("date", "");

	/* Open the keyboard to accept new data to the form, and
	 * copy it to the structure when done. Break out when user
	 * hits EXIT key. */
	sm_d_msg_line ("Enter data; press %KEXIT to quit.",
		WHITE | HILIGHT);
	do {
		key = sm_openkeybd ();
		sm_wrtstruct (&example, &count, C_LANG);
		sprintf (ebuf, "Acknowledged: byte count = %d.",
			count);
		sm_err_reset (ebuf);
	} while (key != EXIT);

	/* Clear the screen and display the final structure contents. */
	sm_resetcrt ();
	printf ("%ld\n", example.date);
	printf ("%s\n", example.name);
	for (count = 0; count < 3; ++count)
		if (example.address[count][0])
			printf ("%s\n", example.address[count]);
	printf ("%s\n", example.telephone);

	exit (0);
}

void
punt (message)
char *message;
{
	sm_resetcrt ();
	fprintf (stderr, "%s: %s\n", program_name, message);
	exit (1);
}
```

NAME

sm_wselect - shuffle windows

SYNOPSIS

```
int sm_wselect (window)
int window;
```

DESCRIPTION

This routine brings a hidden window (or form) to the active position, where it will be referenced by screen manager library calls such as sm_putfield and sm_getfield. You identify the window you want by its sequence number in the stack of windows: the form underlying all the windows is window 0; window = 1 gets the first window opened, window = 2 the second, and so forth.

Here are two different ways of using window selection. One is to select a hidden screen momentarily, to update something in it, and replace it by calling sm_wdeselect without opening the keyboard. This sequence will have no immediate visible effect, unless sm_flush is called. The other is to select a hidden screen and open the keyboard; in this case, the selected screen becomes visible, and may hide part or all of the screen that was previously active. In this way you can implement multipage forms, or switch among several windows that tile the screen (do not overlap).

After this routine is called the order of the windows on the stack, and thus the numbering of the windows, is changed. Subsequent calls to this function must be aware of the current ordering of windows. The JAM screen stack is unaffected by this function.

A related function sm_n_wselect, takes the name of a window as argument and looks for it in the window stack. It will not find windows displayed with sm_d_window or related functions, because they do not record the window name.

JAM applications should only use this routine in paired calls with sm_wdeselect, or the control stack will become incorrect and unexpected results may occur.

RETURNS

The number of the window that was made active (either the number passed, or the maximum if that was out of range). -1 if the system ran out of memory.

VARIANTS AND RELATED FUNCTIONS

```
sm_wdeselect ();
sm_n_wselect (window_name);
```

EXAMPLE

```
/* A typical use of the window selection routines is to
 * update information to a window that may (or may not) be
 * covered.  For example, suppose that the current time
 * should be maintained on the underlying form.  Assume
 * that a field named "curtime" exists on that form.
 * The following code fragments can be used
 * to maintain that field independent of the number of windows
 * currently open above the form.
 */

#include "smdefs.h"

updatetime()
{
     sm_wselect (0); /* quietly select the bottom form */
     sm_n_putfield ("curtime", ""); /* update system time display */
     sm_wdeselect ();/* restore visible window */
     sm_flush ();
     return (0);
}

/* In initialization code: called every second. */

     static struct fnc_data afunc = { 0, updatetime, 0, 10, 0, 0 };
     sm_install (ASYNC_FUNC, &afunc, (int *)0);
```

NAME

sm_zm_options - set zooming options

SYNOPSIS

```
#include "smdefs.h"

int sm_zm_options (flag)
int flag;
```

DESCRIPTION

Controls the behavior of the zooming function, normally bound to the ZOOM key. Flag is a mnemonic defined in smdefs.h , or a pair of mnemonics ored together, that select the operations desired.

Zooming makes more of scrolling and shifting fields visible than is ordinarily the case. It expands the current field either horizontally or vertically, as far as the physical display will allow, and places it in a window atop the currently displayed screen. If a field is both scrolling and shifting, zooming will ordinarily take place in two steps: scrolling expansion will be done first, and shifting expansion next. If a scrolling field has parallel scrolls, they will be displayed too.

The following mnemonics control scroll expansion:

ZM_NOSCROLL	Scrolling arrays will not be expanded; the process goes immediately to shift expansion.
ZM_SCROLL	Scrolling arrays will be expanded to display as many items as possible.
ZM_PARALLEL	Scrolling arrays will be expanded to display as many items as possible, along with their parallel arrays.
ZM_1STEP	Scrolling arrays will be expanded to display as many items as possible; shifting arrays will be expanded at the same time. This option overides the shift mode flag.

The following mnemonics control array/scrolling expansion when shifting fields are being expanded in step two:

ZM_NOSHIFT	Shifting fields will not be expanded; they will remain shifting, and no second step takes place.
ZM_SCREEN	Shifting arrays will have as many on-screen elements as the previous form had. This will be the original form ONLY if ZM_NOSCROLL is used; otherwise, ZM_SCREEN means the scroll-expansion screen, or show as many items as possible.
ZM_ELEMENT	Show 1 element on-screen, but allow scrolling (via arrow keys) to the rest of the array. If a scroll mode was selected, the shift window can be zoomed again, showing all elements.
ZM_ITEM	Show only 1 element. There is no way of seeing more items in the "ITEM" scroll window.

Here are some useful combinations whose semantics are perhaps not crystal clear from the above descriptions:

ZM_PARALLEL|ZM_SCREEN	This is the default, and zooms in two steps. The first step will expand the scrolling arrays; the second step will add expansion of shifting fields, but lose the parallel arrays.
ZM_PARALLEL|ZM_ELEMENT	This will zoom in three steps, adding a single-line shift expansion as step two. This

	will save screen update time when looking at a single record.
ZM_NOSCROLL|ZM_SCREEN	This will disable scroll expansion, but the shift expansion window will still show the items that were on-screen in the original screen.
ZM_NOSCROLL|ZM_ELEMENT	This will minimize screen output when all that is wanted is shift-expansion of an oversize description field.
ZM_NOSCROLL|ZM_ITEM	This will restrict zoom to shift expansion of only the current item.
ZM_1STEP	This will always give the maximum data in the first step. However, no parallel arrays are displayed.

If you define the SMZMOPTIONS variable in your setup file, it will cause this function to be called automatically during start-up with the parameters you specify there.

RETURNS

-1 if the parameter is invalid, otherwise the previous options.

EXAMPLE

```
#include "smdefs.h"

/* Restore zooming options to their defaults. */

sm_zm_options (ZM_PARALLEL | ZM_SCREEN);
```

4 Built-in Invoked Functions

This section describes invoked (caret) functions supplied with JAM. The format of each page is similar to that of the preceding section, with a few exceptions. The synopsis is for a JAM control string, not a programming language source statement; boldface indicates keywords to be entered as shown, while normal type indicates parameters to be replaced by a string you choose. The return value of a caret function can only be used in a target list; see Section 1.2.1.

You may use these functions only in caret control strings. If you call them directly from your own code a wide variety of results, ranging from perfect success through obscure run-time problems to linker errors, is possible.

NAME

jm_exit - end processing and leave the current screen

SYNOPSIS

```
^jm_exit
```

DESCRIPTION

Erases the current form or window and returns to the previous one. If the current form is the last one on the control path (the application's top-level form), will cause JAM to prompt and exit to the operating system.

The effect is like the default action of the run-time system's EXIT key.

EXAMPLE

The following control string invokes a function named process. If it returns 0, another function is invoked to reinitialize the screen; but if it returns -1, the screen is exited. See jm_gotop for another example.

```
^(-1=^jm_exit; 0=^reinit)process
```

The example below shows how a form or a window can be swapped with another form or a window:

```
^(0=&w2)jm_exit
```

NAME

     jm_gotop - return to application's top-level form

SYNOPSIS

     ^jm_gotop

DESCRIPTION

Returns to the application's top-level screen, ordinarily the first screen to appear when the application was run. All forms and windows on the control path are discarded.

The run-time system's SPF1 key performs the same action, unless you change it using SMINICTRL.

EXAMPLE

The following menu makes use of both jm_exit and jm_gotop.

```
              ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
              º                                                  º
              º  Query customer database__    custquery.jam__   º
              º  Update customer database_    custupdate.jam_   º
              º  Free-form query__________    !sql___________   º
              º  Return to previous menu__    ^jm_exit_______   º
              º  Return to main menu______    ^jm_gotop______   º
              º                                                  º
              ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼
```

NAME

jm_goform - prompt for and display an arbitrary form

SYNOPSIS

```
^jm_goform
```

DESCRIPTION

This function pops up a window in which you may enter the name of a form; it will then close the window and attempt to display the form, as if that form's name had appeared in a control string. It is useful for providing a shortcut around a menu system for experienced users.

The result is the same as the default action of the run-time system's SPF3 key.

EXAMPLE

The following line, if placed in your setup file, will make the PF10 key act like SPF3 normally does:

```
SMINICTRL= PF10=^jm_goform
```

NAME

jm_keys - simulate keyboard input

SYNOPSIS

```
^jm_keys keyname-or-string [keyname-or-string ...]
```

DESCRIPTION

Queues characters and function keys that appear after the function name for input to the run-time system, using sm_ungetkey. The run-time system then behaves as though you had typed the keys.

Function keys should be written using the logical key mnemonics listed in smkeys.h . Data characters should be enclosed between apostrophes '', backquotes ``, or double quotes "". This function passes its arguments to sm_ungetkey in reverse order, so you supply them in the natural order.

EXAMPLE

Enter the name of your favorite bar, followed by a tab and the name of its owner:

```
^jm_keys 'Steinway Brauhall' TAB "James O'Shaughnessy"
```

Return to the preceding menu and choose the second option:

```
^jm_keys EXIT HOME TAB XMIT
```

NAME

jm_mnutogl - switch between menu and data entry mode on a dual-purpose screen

SYNOPSIS

```
^jm_mnutogl
```

DESCRIPTION

JAM supports the use of a single screen for both menu selection and data entry; one popular example is a data entry screen with a "menu bar". The screen must, however, be either one or the other at any given moment. This function switches the run-time system's treatment of the screen to the other mode. When the screen is a menu, the run-time system uses sm_menu_proc for keyboard input; otherwise, it uses sm_openkeybd.

Screens with a jam_menu field will be treated initially as menus; others will be treated initially as data entry screens. A screen cannot be used as a menu unless it has fields with the MENU bit set.

EXAMPLE

Below is a screen with four menu options on the left and data entry fields on the right. Selecting a menu option first causes a function to be invoked, which will store the choice; when that function returns zero, jm_mnutogl is invoked to begin data entry. When data entry is completed by the TRANSMIT key, a function attached to that key can use the stored menu option to perform the desired transaction.

```
ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍËÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
º                       º  Name:    __________________________  º
º     LOOKUP            º  Address: __________________________  º
º ^(0=^jm_mnutogl)look  º           __________________________  º
º     ADD___            º           __________________________  º
º ^(0=^jm_mnutogl)add_  º  Phone:   (___)___-____               º
º     MODIFY            º  Notes:   __________________________  º
º ^(0=^jm_mnutogl)mod_  º           __________________________  º
º     DELETE            º           __________________________  º
º ^(0=^jm_mnutogl)del_  º           __________________________  º
ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÊÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼
```

NAME

jm_system - prompt for and execute an operating system command

SYNOPSIS

```
     ^jm_system
```

DESCRIPTION

This function pops up a small window, in which you may enter an operating system command. When you press TRANSMIT, it closes the window and executes the command. While the command is executing, your terminal is returned to the operating system's default I/O mode.

The run-time system's SPF2 key invokes this function by default.

EXAMPLE

The following line, when placed in your setup file, will cause the PF10 key to act as SPF2 normally does:

```
     SMINICTRL= PF10 = ^jm_system
```

NAME

jpl - invoke a JPL procedure stored in a file

SYNOPSIS

```
^jpl filename [ argument ... ]
```

DESCRIPTION

This function invokes a procedure written in the JYACC Procedural Language and stored in a file. Filename should be the name of the file containing your procedure; anything following that will be passed to the procedure as arguments. The value returned by your procedure will be passed along by jpl for use in a target list.

This function is similar to the JPL jpl command, but does not do colon expansion.

EXAMPLE

The control string below invokes a JPL function, passing it the name of a field in which its command is stored.

```
^jpl execute command
```

5 Environment and Configuration Files

This section describes only an essential few of the many configuration variables supported by JAM; refer to the section on setup files in the Configuration Guide for more. The following list summarizes JAM's most commonly used environment variables:

```
   SMMSGS      file name for message text SMVIDEO
               file name for video information SMKEY
               file name for keyboard translation SMVARS
               single variable for consolidation TERM
               terminal mnemonic SMTERM
               substitute for TERM
```

The first three are required: they name configuration files used by JAM to describe its operating environment. It finds them by looking either in the system environment, or in a binary file named by SMVARS. If it fails to find either the variables or the configuration files themselves, it will post a message and exit.

The system variable TERM and the JAM variable SMTERM are used in conjunction with SMVARS. You may replace the three environment variables with SMVARS. This variable gives the name of a binary file containing the other screen manager variables. A typical SMVARS source file might look like the following:

```
SMKEY = (TV:TV950)/sm/config/TVkeys.bin SMKEY =
(vt100:d0:d1:d2)/sm/config/vt100keys.bin SMVIDEO = /sm/config/vt100vid.bin
SMMSGS = /usr/local/msgfile.bin SMPATH = /appl/masks
```

The lists enclosed in parentheses are terminal types; JAM uses them to find the right files for your terminal. The SMVARS source file must be converted to binary using var2bin; the system environment then needs only the name of the binary file (and perhaps your terminal), as

```
SMVARS=/usr/local/smvars.bin
TERM=vt100
```

The terminal type, used to match against the lists in parentheses, is taken from the variable SMTERM, or from TERM if that is not present. (If you want JAM to recognize a terminal mnemonic different from TERM, put it in SMTERM. For example, the text editor might work fine with the terminal in vt100 emulation, but JAM could want the features of vt220 emulation; you could set TERM to vt100 and SMTERM to vt220.)

Application programs initialize JAM by calling sm_initcrt. This call must precede calls to just about any other library routine, except those that install memory resident message, key and/or video files, or set options; see Section 12.2. sm_initcrt first calls an optional user-supplied initialization routine, sm_u_uinit, which may (among other things) initialize the character string sm_term.

sm_initcrt then looks for SMVARS and SMSETUP in the system environment, and uses them to read in setup files. Subsequently, setup variables are sought first in the system environment and then in the setup files.

Next the terminal type is determined and placed in a character array called sm_term, which is declared in smdefs.h . An application program can force a terminal type by setting sm_term before sm_initcrt is called. If the array is empty, SMTERM is sought first, then TERM. If neither is found, initialization is attempted without a terminal type.

SMMSGS comes next. If this variable is not found, or the file it names is found by sm_msginit not to be useable, JAM will abort initialization. Initialization errors in file I/O are reported using the C library function perror; these messages are system-dependent. Other errors encountered before the message file is loaded provoke hard-coded messages; afterward, all error messages are taken from the message file.

Video and keyboard initialization are next attempted, in that order, using sm_keyinit and sm_vinit respectively. If after all this JAM still can't determine which configuration files to use, it will prompt you for a terminal type and retry the entire sequence. Finally, JAM initializes the terminal and its operating system channel.

After ensuring that the environment is set up, sm_initcrt initializes the operating system's terminal channel. It is set to "no echo" and non-buffered input. If other changes are desired (e.g. from 7 to 8 data bits), they can be made in the user initialization routine.

Next the initialization string found in the video file is transmitted to the terminal. The Video Manual gives details; here we simply note that system calls can be embedded in the string. Often this feature can be used in lieu of a user initialization routine.

6 Keyboard Input

Keystrokes are processed in three steps. First, the sequence of characters generated by one key is identified. Next the sequence is translated to an internal value, or logical character. Finally, the internal value is either acted upon or returned to the application ("key routing"). All three steps are table-driven. Hooks are provided at several points for application processing; they are described in Section 8.

6.1 Logical Characters or Keys

JAM processes characters internally as logical values, which frequently (but not always) correspond to the physical ASCII codes used by terminal keyboards and displays. Specific keys or sequences of keys are mapped to logical values by the key translation table, and logical characters are mapped to video output by the MODE and GRAPH commands in the video file. For most keys, such as the normal displayable characters, no explicit mapping is necessary. Certain ranges of logical characters are interpreted specially by JAM; they are

- 0x0100 to 0x01ff: operations such as tab, scrolling, cursor motion
- 0x6101 to 0x7801: function keys PF1 - PF24
- 0x4101 to 0x5801: shifted function keys SPF1 - SPF24
- 0x6102 to 0x7802: application keys APP1 - APP24

6.2 Key Translation

The first two steps together are controlled by the key translation table, which is loaded during initialization. The name of the table is found in the environment (see Section 5 for details). The table itself is derived from an ASCII file which can be modified by any editor; a screen-oriented program, modkey, is also supplied for creating and modifying key translation tables (see the Configuration Guide).

After the table is read into memory, it has the form of an array of structures:

```
     struct
     {
          char key[6];
          int value;
     };
```

The first field is an array of up to 6 characters; it holds the sequence of characters sent to the computer when the key is pressed. The second member is the logical value of the key.

JAM assumes that the first character of a multi-character key sequence is a control character in the ASCII chart (0x00 to 0x1f, 0x7f, 0x80 to 0x9f, or 0xff). All characters not in this range are assumed to be displayable characters and are not translated. The routine that performs the translation is called sm_getkey.

Upon receipt of a control character, sm_getkey searches the translation table. If no match is found on the first character, the key is accepted without translation. If a match is found on the first character and the next character in the table's sequence is 0, an exact match has been found, and sm_getkey returns the value indicated in the table. The search continues through subsequent characters until either

1. an exact match on n characters is found and the n+1'th character in the table is zero, or n is 6. In this case the value in the table is returned.
2. an exact match is found on n-1 characters but not on n. In this case sm_getkey attempts to flush the sequence of characters returned by the key.

This last step is of some importance: if the operator presses a function key that is not in the table, the screen manager must know "where the key ends." The algorithm used is as follows. The table is searched for all entries that match the first n-1 characters and are of the same type in the n'th character, where the types are digit, control character, letter, and punctuation. The smallest of the total lengths of these entries is assumed to be the length of the sequence produced by the key. (If no entry matches by type at the n'th character, the shortest sequence that matches on n-1 characters is used.) This method allows sm_getkey to distinguish, for example, between the sequences ESC O x, ESC [A, and ESC [1 0 ~.

6.3 Key Routing

The main routine for keyboard processing is sm_openkeybd. This routine calls sm_getkey to obtain the translated value of the key. It then decides what to do based on the following rules.

If the value is greater than 0x1ff, sm_openkeybd returns to the caller with this value as the return code.

If the value is between 0x01 and 0x1ff, the processing is determined by a routing table. This is an array whose address is stored in sm_route_table. The value returned by sm_getkey is used to index into the table, where two bits determine the action. The bits are examined independently, so four different actions are possible:

- If neither bit is set, the key is ignored.
- If the EXECUTE bit is set and the value is in the range 0x01 to 0xff, it is written to the screen (as interpreted by the GRAPH entry in the video file, if one exists). If the value is in the range 0x100 to 0x1ff, the appropriate action (tab, field erase, etc.) is taken.
- If the RETURN bit is set, sm_openkeybd returns the logical value to the caller; otherwise, sm_getkey is called for another value.
- If both bits are set, the key is executed and then returned.

The default settings are ignore for ASCII and extended ASCII control characters (0x01 - 0x1f, 0x7f, 0x80 - 0x9f, 0xff), and EXECUTE only for all others. The application function keys (PF1-24, SPF1-24, APP1-24, and ABORT) are not handled through the routing table. Their routing is always RETURN, and cannot be altered.

Applications can change key actions on the fly by modifying the routing table. For example, to disable the backtab key the application program would execute

```
     sm_route_table[BACK] = 0;
```

To make the field erase key return to the application program, use

```
     sm_route_table[FERA] = RETURN;
```

Key mnemonics can be found in the file smkeys.h .

7 Screen Output

JAM uses a sophisticated delayed-write output scheme, to minimize unnecessary and redundant output to the display. No output at all is done until the display must be updated, either because keyboard input is being solicited or the library function sm_flush has been called. Instead, the run-time system does screen updates in memory, and keeps track of the display positions thus "dirtied". Flushing begins when the keyboard is opened; but if you type a character while flushing is incomplete, the run-time system will process it before sending any more output to the display. This makes it possible to type ahead on slow lines. You may force the display to be updated by calling sm_flush.

JAM takes pains to avoid code specific to particular displays or terminals. To achieve this it defines a set of logical screen operations (such as "position the cursor"), and stores the character sequences for performing these operations on each type of display in a file specific to the display. Logical display operations and the coding of sequences are described in excruciating detail in the Video Manual; the following sections, along with Sections 11.2, and 12.4, describe additional ways in which applications may use the information encoded in the video file.

7.1 Graphics Characters and Alternate Character Sets

Many terminals support the display of graphics or special characters through alternate character sets. Control sequences switch the terminal among the various sets, and characters in the standard ASCII range are displayed differently in different sets. JAM supports alternate character sets via the MODEx and GRAPH commands in the video file.

The seven MODEx sequences (where x is 0 to 6) switch the terminal into a particular character set. MODE0 must be the normal character set. The GRAPH command maps logical characters to the mode and physical character necessary to display them. It consists of a number of entries whose form is

```
     logical value = mode physical-character
```

When JAM needs to output logical value it will first transmit the sequence that switches to mode, then transmit physical-character. It keeps track of the current mode, to avoid redundant mode switches when a string of characters in one mode (such as a graphics border) is being written. MODE4 through MODE6 switch the mode for a single character only.

7.2 The Status Line

JAM reserves one line on the display for error and other status messages. Many terminals have a special status line (not addressable with normal cursor positioning); if such is not the case, JAM will use the bottom line of the display for messages. There are several sorts of messages that use the status line; they appear below in priority order.

1. Transient messages issued by sm_err_reset or a related function
2. Ready/wait status
3. Messages installed with sm_d_msg_line or sm_msg
4. Field status text
5. Background status text

There are several routines that display a message on the status line, wait for acknowledgement from the operator, and then reset the status line to its previous state: sm_query_msg, sm_err_reset, sm_emsg, sm_quiet_err, and sm_qui_msg. sm_query_msg waits for a yes/no response, which it returns to the calling program; the others wait for you to acknowledge the message. These messages have highest precedence.

sm_setstatus provides an alternating pair of background messages, which have next highest precedence. Whenever the keyboard is open for input the status line displays Ready; it displays Wait when your program is processing and the keyboard is not open. The strings may be altered by changing the SM_READY and SM_WAIT entries in the message file.

If you call sm_d_msg_line, the display attribute and message text you pass remain on the status line until erased by another call or overridden by a message of higher precedence.

When the status line has no higher priority text, the screen manager checks the current field for text to be displayed on the status line. If the cursor is not in a field, or if it is in a field with no status text, JAM looks for background status text, the lowest priority. Background status text can be set by calling sm_setbkstat, passing it the message text and display attribute.

In addition to messages, the rightmost part of the status line can display the cursor's current screen position, as, for example, C 2,18. This display is controlled by calls to sm_c_vis.

During debugging, calls to sm_err_reset or sm_quiet_err can be used to provide status information to the programmer without disturbing the main screen display. Keep in mind that these calls will work properly only after screen handling has been initialized by a call to sm_initcrt. sm_err_reset and sm_quiet_err can be called with a message text that is defined locally, as in the following examples:

```
1.   sm_err_reset ("Zip code invalid for this state.");

2.   int i, j;
     char bugbuf[81];
     sprintf (bugbuf, "i = %d; j = %d.", i, j);
     sm_quiet_err (bugbuf);
```

However, the JAM library functions use a set of messages defined in an internal message table. This is accessed by the function sm_msg_get, using a set of defines in the header file smerror.h . For example:

```
     sm_quiet_err (sm_msg_get (SM_MALLOC));
```

The message table is initialized from the message file identified by the environment variable SMMSGS. Application messages can also be placed in the message file. See the section on message files in the Configuration Guide.

8 User-definable Functions

The JAM library contains numerous hooks where you can install routines to be called from someplace within the run-time system. This is occasionally necessary when you have exotic hardware, but more often it is simply convenient to do some application-specific function in the context of a run-time system operation. The keyboard input functions, for instance, contain several hooks.

A hook tells JAM everything it needs to know in order to call your routine. This generally includes the routine's name, its address, and the programming language it was written in. JAM knows how to interpret the routine's arguments and return value by which hook it is installed on.

The following list itemizes all the different hooks and tells where to go for information about each; the boldface references are to Section 3 of this chapter. Many hooks hold a single application routine, but the first three in the list can hold any number. In this case JAM uses the routine name, such as the one you specify in the screen for an attached function, to find the information it needs.

- Screen entry and exit functions, Section 1.4
- Attached functions, including field entry, field exit (or validation), and attached JPL procedure, Section 1.3
- Invoked functions, Section 1.2
- Memo edits, Section 8.2
- Keystroke processing hooks:
    - Key translation function, sm_u_keychange
    - Key recording hook, sm_u_record
    - Key playback hook, sm_u_play
    - Key lookahead hook, sm_u_avail
    - Asynchronous function, sm_u_async
- Display processing hooks:
    - Video processing hook, sm_u_vproc
    - Insert mode transition function, sm_u_inscrsr
    - Status line display function, sm_u_statfnc
- Screen manager initialization and reset hooks, sm_u_uinit and sm_u_ureset
- Check digit validation, sm_u_ckdigit

The hooks for which library function names are given are all documented under those names in the library section of this chapter, and should be installed using sm_install; the next section discusses installation. Attached, invoked (caret), and screen entry functions are discussed in Sections 1.3, 1.2, and 1.4, respectively. Memo edits are discussed below.

8.1 Installation

In general, there are two ways of getting JAM to call an application routine. One is to write a routine in C with the same name and calling sequence as one provided in the library, and link it with the application; the linker will then ignore the library routine and load the application routine in its place. You can use this technique for the sm_u_uinit, sm_u_ureset, video, and check digit functions. This mechanism will not work on many systems. It is supported for backward compatibility only, and may be phased out in the future.

The other technique is to call sm_install, q.v.; this is portable, supported, and strongly to be preferred. You pass this function a data structure containing the name, address, and source language of your function. Every function or type of function listed in the previous section (with the exception of memo edits) can be installed with sm_install.

8.2 Memo Text Edits

Memo text edits are not function hooks; rather, they are special edits you can use to attach arbitrary information to a field. The JAM run-time system ignores memo edits, but application routines may access them, using sm_edit_ptr. An example of memo edit use follows.

Suppose a screen contains fields whose contents are interdependent, such as state abbreviation and zip code. The zip code field might have an attached function that performs a validation based on the state abbreviation. However, if the zip code was validated and the operator subsequently changed the state abbreviation, the zip code might become invalid. A simple solution would be to attach the following function to the state abbreviation field:

```
int state_change (field_num, data, occurrence, val_mdt)
{
     if (val_mdt & MDT)
                       /* current field changed */
          sm_n_novalbit ("zip"); /* reset zip VALIDED bit*/
     return 0;
}
```

Now suppose a screen contains several groups of interdependent fields. One could use MEMO1 to hold a list of dependent fields (say by number, separated by commmas), and the following function to process them:

```
field_change (field_num, data, occurrence, val_mdt)
int field_num;
char *data;
int occurrence;
int val_mdt;
{
     char *ptr;

     if (val_mdt & MDT) /* current field changed */
     {
          ptr = sm_edit_ptr (field_number, MEMO1);
          if (ptr)      /* find MEMO1 edit, if any */
          {
               ptr += 2;
                  /* skip length, command code*/
               while (*ptr)
               {
                    sm_novalbit (atoi (ptr));
                    while (*ptr && *ptr != ',')
                         ++ptr;
                    if (*ptr) ++ptr;
               }
          }
     }
     return (0);
}
```

9 The Local Data Block

The LDB is a table of name-value pairs maintained by the run-time system. As you enter data in named fields, the values are copied to the LDB; when you bring up a new screen with named fields, values are copied in from the LDB. This procedure implements data links, and goes by the name of LDB write-through. The list of names is taken from the data dictionary; certain other field characteristics needed for the proper formatting of data, such as field size and currency formats, are also taken from the data dictionary.

During screen display, i.e. during execution of sm_r_window or a variant, sm_allget is called to load named fields with values from the LDB (without setting their MDT bits). Whenever a screen is removed from the display or is covered by a window, sm_lstore is first called to update the LDB with screen values. In between, i.e. while the screen is being processed, certain library functions requiring data values seek them first in the screen, and search the LDB if they are not there. This preserves the illusion that named fields in the LDB are always accessible and always have the latest values in them. Screens brought up by the run-time system share this feature with those brought up by your control fields and code, since the run-time system uses sm_r_window too.

The library functions that use the LDB are n_ and i_ variants, plus a few that reference the LDB explicitly, such as sm_lreset and sm_allget. Functions that refer to a field by number, namely the o_ variant and the basic function, obviously cannot access data outside the screen. The e_ functions do refer to fields by name; however, they are designed specifically for accessing elements of onscreen arrays, and so do not go to the LDB. Field calculation expressions and JPL procedures that contain named fields will reference the LDB in the same way.

If the scope of an LDB entry is 1, or constant, its value cannot be changed by write-through; only sm_lreset can change the values of constants.

LDB write-through is normally on; you can turn it off (and back on) by calling sm_dd_able. Care is necessary, though; even the explicit LDB access functions, such as sm_allget, do not work when write-through is off. As an example, you could make the LDB read-only in a single screen by putting the following code fragment in a screen entry function:

```
     sm_allget (1);
     sm_dd_able (0);
```

You would then have to call sm_dd_able (1) to re-enable LDB write-through in a screen exit function, which is called after the sm_lstore; calling it before then, as in an EXIT control string, would cause the LDB to be updated. If you want to disable LDB processing for an entire application, simply omit the call to sm_ldb_init before starting up the run-time system; see Section 1.6.

10 Writing Portable Applications

The following section is an attempt to identify features of hardware and operating system software that can cause JAM to behave in a non-uniform fashion. An application designer wishing to create programs that run across a variety of systems will need to be aware of these factors.

10.1 Terminal Dependencies

JAM can run on screens of any size. On screens without a separately addressable status line, JAM will steal the bottom line of the display (often the 24th) for a status line, and status messages will overlay whatever is on that line. A good lowest common denominator for screen sizes is 23 lines by 80 columns, including the border (if any).

Different terminals support different sets of attributes. JAM makes sensible compromises based on the attributes available; but programs that rely extensively on attribute manipulation to highlight data may be confusing to users of terminals with an insufficient number of attributes.

Attribute handling can also affect the spacing of fields and text. In particular, anyone designing screens to run on terminals with onscreen attributes must remember to leave space between fields, highlighted text, and reverse video borders for the attributes. Some terminals with area attributes also limit the number of attribute changes permitted per line (or per screen).

The key translation table mechanism supports the assignment of any key or key sequence to a particular logical character. However, the number and labelling of function keys on particular keyboards can constrain the application designer who makes heavy use of function keys for program control. The standard VT100, for instance, has only four function keys properly speaking. For simple choices among alternatives, the library routines sm_menu_proc and sm_choice are probably better than switching on function keys.

Using function key labels, or keytops, instead of hard-coded key names is also important to making an application run smoothly on a variety of terminals. Field status text and other status line messages can have keytops inserted automatically, using the %K escape. No such translation is done for strings written to fields; in such cases, you may want to place the strings in a message file, since your setup file can now specify terminal-dependent message files.

10.2 Items in smmach.h

The header file smmach.h , which is supplied with the JAM library, contains information that library routines need to deal with certain machine, operating system, and compiler dependencies. These include:

- The presence of certain C header files and library functions.
- Byte ordering in integers and support for the unsigned character type.

- Path name and command line argument separator characters.
- Pointer alignment and structure padding.

The header file is thoroughly commented, and application designers are encouraged to make use of the information there.

11 Writing International Applications

11.1 Messages

All messages displayed by the JAM library routines are in the ASCII message file; this file may be edited to translate the messages into any language. Further, the screens used by JAM utilities are supplied in libraries; they may be edited there and their prompts translated.

11.2 Characters Outside the U.S. ASCII Set

Terminals that are capable of displaying characters outside the 7-bit U.S. ASCII set may do it two ways: either they display 8-bit characters (those in the range 0x80 to 0xff), or they have control sequences for switching among several character sets. On input, again, the foreign keys may generate either 8-bit codes or a multi-character sequence.

If the terminal's keyboard generates 8-bit codes and its display can display them, there is little the application needs to do except set the GRTYPE entry in the video file. However, some caution is necessary, because JAM reserves the range 0x100 to 0x1ff for its logical operations (tab, field erase, scroll, etc.). Applications should avoid mapping keys into this range. Alternate character sets require more work. First a logical value for the character in question must be selected; the range 0xa0 to 0xfe is good. Next, the control sequences for switching character sets must be defined in the video file as MODEx. MODE0 is the normal character set; JAM supports up to 6 alternate character sets (MODE1 to MODE6). Finally, the GRAPH entry in the video file must be set up. It consists of any number of sub-entries whose format is

```
     logical-value = mode physical-character
```

where mode is 1 to 3 and logical-value is the value that gives the desired output in the appropriate mode. (See Section 6.1 for more details on alternate character sets, and the Video Manual for how to create and use video files.)

12 Writing Efficient Applications

12.1 Memory-resident Screens

Memory-resident screens are much quicker to display than disk-resident screens, since no disk access is necessary to obtain the screen data. There are two ways of using the JAM library functions with memory-resident screens; but in either case, the screens must first be converted to source language modules with bin2c or a related utility (see the Configuration Guide), then compiled and linked with the application program.

sm_d_form and related library functions can be used to display memory-resident screens; each takes as one of its parameters the address of the global array containing the screen data, which will generally have the same name as the file the original screen was originally stored in.

A more flexible way of achieving the same object is to use a memory-resident screen list. Bear in mind that the JAM utility can only operate on disk files, so that altering memory-resident screens during program development requires a tedious cycle of test - edit - reinsert with bin2c - recompile. The JAM library maintains an internal list of memory-resident screens that sm_r_window and related functions examine. Any screen found in the list will be displayed from memory, while screens not in the list will be sought on disk. This means that the application can be coded to use one set of calls, the r-version, and screens can be configured as disk- or memory-resident simply by altering the list.

The screen list is pointer to an array of structures:

```
struct form_list
{
     char *form_name;
     char *form_ptr;
} *sm_memforms;
```

To initialize it, an application would use code like the following:

```
#include "smdefs.h"
extern char mainform[], popup1[];
extern char popup2[], helpwin[];

struct form_list mrforms[] =
{
     "mainform.jam",mainform,
     "popup1.jam",  popup1,
     "popup2.jam",  popup2,
     "helpwin.jam", helpwin,
     "",            (char *)0

};
...
sm_formlist(mrforms);
```

Note the last entry in the screen list: an empty string for the name and a null pointer for the screen data. This marks the end of the list, and is required. The call to sm_formlist adds the scrərens in your list to JAM's internal list.

Using memory-resident screens (and configuration files, see the next section) is, of course, a space-time tradeoff: increased memory usage for better speed.

The naming of screens in the screen list presents a few problems. First of all, the name of the character array for a screen must not conflict with the name of any other global data item or function; for instance, a memory-resident screen with the name of a function could not be installed with a program. Secondly, the filename extension must be removed from the array name, which means that screens with the same name but different extensions will conflict. If your application code structure permits, you can minimize the foregoing problems by making the screens in question local in scope to a particular source file.

JAM will append the extension found in the setup variable SMFEXTENSION to screen names (e.g. in control fields) that do not already contain an extension; you must take this into account when creating the screen list. JAM may also convert the name to uppercase before searching the screen list; this is governed by the SMFCASE variable.

12.2 Memory-resident Configuration Files

Any or all of the three configuration files required by JAM can be made memory resident. First a C source file must be created from the text version of the file, using the bin2c utility; see the Configuration Guide. The source files created are not intended to be modifiable, or even understandable, any more than the binary files are; each defines some data objects which are of no concern to us. Each file contains one data object that is globally known. The following fragment makes all three files memory-resident:

```
     /* Memory-resident message, key, and video files */
     extern char msg_file[];
     extern char key_file[];
     extern char video_file[];

     /* ...more declarations... */

     sm_msginit (msg_file);
     sm_keyinit (key_file);
     sm_vinit (video_file);
     sm_initcrt ("");

     /* ...possibly initialize function and form lists  */

     /* ...application code */
```

If a file is made memory-resident, the corresponding environment variable or SMVARS entry can be dispensed with.

12.3 Message File Options

If you need to conserve memory and have a large number of messages in message files, you can make use of the MSG_DSK option to sm_msgread. This option avoids loading the message files into memory; instead, they are left open, and the messages are fetched from disk when needed. Bear in mind that this uses up additional file descriptors, and that buffering the open file consumes a certain amount of system memory; you will gain little unless your message files are quite large.

12.4 Avoiding Unnecessary Screen Output

Several of the entries in the JAM video file are not logically necessary, but are there solely to decrease the number of characters transmitted to paint a given screen. This can have a great impact on the response time of applications, especially on time-shared systems with low data rates; but it is noticeable even at 9600 baud. To take an example: JAM can do all its cursor positioning using the CUP (absolute cursor position) command. However, it will use the relative cursor position commands (CUU, CUD, CUF, CUB) if they are defined; they always require fewer characters to do the same job. Similarly, if the terminal is capable of saving and restoring the cursor position itself (SCP, RCP), JAM will use those sequences instead of the more verbose CUP.

The global flag sm_do_not_display may also be used to decrease screen output. While this flag is set, calls into the JAM library will cause the internal screen image to be updated, but nothing will be written to the actual display; the latter can be brought up to date by resetting the flag and calling sm_rescreen. With the implementation of delayed write in JAM Release 4, this sort of trick is necessary much less often than it was under Release 3.

12.5 Stub Functions

Certain screen manager facilities can be omitted from an application if they are not used, by defining certain literals in the application. This can result in substantial memory savings; however, it requires that the screen manager libraries not be pre-linked or pre-bound, i.e. is not supported on all systems. The following facilities may be stubbed out:

subsystem	#define
the math package	NOCALC scrolling functions
	NOSCROLL time and date functions
	NOTIMEDATE help screens
	NOHELP shifting fields
	NOSHIFT range checking functions
	NORANGE word wrap
	NOWRAP field zoom expansion
	NOZOOM regular expressions
	NOREGEXP form libraries
	NOFORMLIB JYACC procedural language
	NOJPL read/write data structure
	NOSTRUCT save/restore screen data
	NOSRD local print
	NOLPR area attributes
	NOAREA window selection
	NOWSEL keytop translation
	NOLKEYLAB setup parameter file
	NOSETUP shift/scroll indicators
	NOINDICATORS

To omit any one or combination of the above, first #define the appropriate literal in your application, then #include the stubs file. This need only be done once, for instance in the application's main routine. For example, if the application is not going to use scrolling fields, the scrolling functions could be omitted, and the application source might look like the following:

```
#define NOSCROLL
#include "smdefs.h"
#include "sm_stubs.c"

main ()
{
     /* ...the application code... */
}
```

The effect of defining the literal and including sm_stubs.c is to declare stub routines in the application; this causes the linker not to add the real routines from the screen manager library to the application. The bulk of the savings will be in code space. The stubbing technique does not work on systems where the library is itself a linked entity, such as a shareable library.

If range, math, and JPL support are all stubbed out, you can also omit linking the C math library (-lm flag on UNIX systems, math library on MS-DOS systems).

Index

In this Index, library functions are displayed in boldface, without the prefixes specific to the language interface. Video and setup file entries appear in ELITE CAPS, while utility programs and JPL commands are in elite lower-case. Function key names are in ROMAN CAPS.