JAM

PL/1 Programmer's Guide for Stratus

This is the PL/1 Programmer's manual for JAM Release 5. It is as accurate as possible at this time; however, both this manual and JAM itself are subject to revision.

Please send suggestions and comments regarding this document to:

Technical Publications Manager
JYACC, Inc.
116 John Street
New York, NY 10038

(212) 267-7722

A Note To Language Interface Users

JYACC makes every effort possible to design language interfaces that duplicate the original C Programmers Library. However, due to differences among various programming languages, an exact one to one correspondence is not always possible. In some cases, routines contained in the C version have been replaced with other routines designed to take advantage of a particular programming language's features.

Please note that your interface contains intentionally undocumented routines. Some of these routines are no longer part of JAM, having been replaced by more efficient routines, and are included only for backward compatibility with applications created with earlier versions of JAM. The rest are internal routines and are not intended to be directly accessed by developers.

A Note To Non–UNIX Users

Throughout the manual, a forward slash (/) has been used to indicate a subdirectory. For example,

```
/usr/local/file
```

means that `file` is a file in the directory `local` which is in turn a sub–directory of `usr`, which is not the root directory.

TABLE OF CONTENTS

Chapter 1. Introduction

This document is intended for JAM Programmers. We discuss the development and creation of executable JAM programs incorporating the Screen Manager, developer–written hook functions, and the JAM Executive. We will briefly touch on how custom executives may be written. Finally, there is a comprehensive reference of JAM library functions.

Discussions on the creation of JAM screens, data dictionaries, and keysets are found in the Author's Guide. JPL is fully documented in the JPL Programmer's Guide.

This document assumes that the reader has previously read the JAM Development Overview and the Author's Guide. The Development Overview is particularly important as the major architectural components of JAM are explained there in detail.

JAM is written in C, and the C programming interface and libraries are distributed with every license. This PL/1 language interface document is an adaptation of the JAM C Programmer's Guide.

You will need to program in PL/1 (or some other supported third–generation language) to accomplish the following tasks:

- To customize JAM to your environment or application by modifying the main program provided in source form with the product.
- To write hook functions that do application–specific and back–end processing during the execution of the application.
- To take full control of the application by writing an application–specific executive[1].
- To create executable JAM Programs.

As discussed in detail in the Development Overview, JAM Applications consist of screens, a data dictionary, hook functions, and an executable program. The creation of

1. It is strongly recommended that the JAM Executive be used in all but the most unusual of circumstances. A comparison of the JAM Executive with your own executive is presented in the Development Overview.

screens and data dictionaries is discussed in the Author's Guide. JPL programming is discussed in the JPL Programmer's Guide. In this chapter, we discuss how to create a JAM program. Compilation and linking are specific to platforms and operating systems and are discussed in the Installation Guide.

Two different versions of an application can be created with JAM. The Application Executable is the program delivered to the end-user to control the run time application. The JAM Authoring Executable is used to create application components and test the application during development. Only the JAM Authoring Executable will grant user access to the Screen Editor, the Data Dictionary Editor, and the Keyset Editor. The JAM Authoring Executable can only be used for the testing of applications that use the JAM Executive.

1.1. APPLICATION EXECUTABLE

Application Executable programs fall into two categories: those that use the JAM Executive to manage the flow of control from screen to screen, and those that use an application-specific executive. We discuss both of these approaches in the sections that follow.

1.1.1. Applications Using the JAM Executive

In applications that use the JAM Executive, most of the control flow is encapsulated in the screens. The majority of the PL/1 programming task is to write hook functions (section 2. page 7) that are called by the Screen Manager or by the JAM Executive when certain events occur.

Applications that use the JAM Executive will need to be linked with the PL/1 interface library `xif`, the Screen Manager library `sm`, the JAM Executive library `jm`, and, in general, the standard math library on your system.

NOTE: Refer to the Stratus Software Release Bulletin for specifics of the VOS library setup.

JYACC provides the main routine source code for applications that use the JAM Executive in a file called `jmain.pl1`. This routine performs various necessary initializations before calling the function that starts up the JAM Executive. You may want to modify this code to change JAM's default behavior.

1 1 2. Applications Using a Custom Executive

In rare cases, a developer may choose to write a custom executive, one that is specific to a particular application. In custom executives, no library functions specific to the JAM Executive should be used. The JAM Executive functions may only be used in applications using the JAM Executive — they are listed in section 11.15. on page 85. .

Applications that do not use the JAM Executive should be linked with the PL/1 interface library `xif`, the Screen Manager library `sm`, and, in general, the standard math library. If the LDB is needed, the JAM Executive library `jm` should also be linked in, but it is important the application not call any JAM Executive routines.

The "sample" application provided with JAM is a simple example of an application using a custom executive.[2] This application brings up a screen on which the end–user can enter some account data, and then save the data and call it up again. There is a help screen, tied to one of the function keys, which is implemented as a memory–resident screen, and a hook written function that verifies the area code. The discussion below outlines the basic steps that a custom executive should perform, using `sample.pl1` as an example.

To follow this discussion, you should either print this file out, or call it up in an editor. Refer to the Stratus Software Release Bulletin for the location of `sample.pl1`. The hook function AREACODE can be found in the same file.

Header Files

JAM user defines are included as necessary, depending on the library routines utilized in the program. The documentation for each library routine indicates which, if any, header files are required.

Declarations

A memory–resident screen is declared at the top of the program along with whatever variables are necessary.

Screen Manager Initialization

After all the header files and declarations at the top of the source module, the Screen Manager and the terminal are first initialized with a call to `xsm_initcrt`. Since an empty string is passed as the argument, the search path for screens is expected to be found in the environment.

2 Note that JPL is available to applications that do not use the JAM Executive Note also that hook functions may be installed and used in applications that do not use the JAM Executive. These applications, however, will not be able to use control strings.

Install Hook Functions

Most Screen Manager hook functions are installed via the `-retain_all` argument to the `bind` command. This is the case for the hook function `areacode`, which is called as a field validation function. For certain types of hook functions, explicit installation is necessary and should occur here—after initialization, but before displaying the first screen. The various types of hook functions and their installation are described in detail in Chapter 2.

Display the Main Form

After initialization is complete, the screen `sample1.frm` is opened as a form with a call to `xsm_r_form`. If an error occurs, the program will terminate.

Activate Screen

`sample1.frm` is activated within a loop. The loop terminates if the user strikes the EXIT key, which causes the routine `xsm_input` to return with the return code EXIT defined in `smkeys.incl.pl1`. The actual data entry, cursor movement, help processing, character edit masking, and validation are handled within `xsm_input`, so the programmer need not be concerned with them. Whenever the user strikes TRANSMIT, EXIT, or some other function key, `xsm_input` returns control to the calling program. In this case, the PF2, PF3 and EXIT keys cause specific actions. All other function keys cause a beep and the while loop to continue, calling `xsm_input` again.

Open a Window

The PF3 key brings up the memory-resident screen that was declared earlier, and then waits for the user to press a key.

Close a Window

During the run of any application, there is always a form displayed. When a new form is displayed, all existing screens are implicitly closed. Windows, however, need to be explicitly closed if the application is to retreat to an underlying screen. After the PF3 window is displayed, when the user strikes a key the program calls `xsm_close_window` to close this window.

Handle Errors

The executive should have a facility to handle errors. The PF2 key triggers a procedure, PROCESS, which opens a window allowing the user to save or read data. While the specifics of this data manipulation are beyond the scope of this introductory discussion, use of the error handling routine `xsm_err_reset`, which displays an error message on the status line, is illustrated about halfway through the procedure listing. `xsm_err_reset` takes a single string argument, and places that string on the status line. The user is forced to acknowledge the error by striking the space bar[3].

Reset the Terminal

Before the application terminates, it calls `xsm_resetcrt` to reset terminal characteristics to a state expected by the operating system. Here this occurs when the user presses the EXIT key.

1.2. AUTHORING EXECUTABLE

The Authoring Executable must use the JAM Executive, and may have developer-written hook functions linked in. The main routine for the Authoring Executable is provided in source form in a file called `jxmain.pl1`. You may want to modify that file to change the default behavior of the authoring tool `jxform`. It is strongly suggested that JAM developers read and understand this code, as it is instructive and may help with an understanding of the product.

The compiled Authoring Executable may be called with the optional command-line switch `-e`. This will cause the authoring tool to start up directly within the Screen Editor (as opposed to starting up in application mode).

Authoring executables must be linked with the PL/1 interface library `xif`, the JAM Authoring Library `jx`, the JAM Executive library `jm`, the Screen Manager library `sm`, and, in general, the standard math library. Since these executables are linked with the JAM Authoring Library `jx`, they may not be re-sold or distributed on machines for which there is no software license from JYACC. This restriction applies *only* to Authoring Executables, which are intended for application *development* only.

NOTE: Refer to the Stratus Software Release Bulletin for specifics of the VOS library setup.

3 The developer may change the way messages are acknowledged with the library routine `XSM_OPTION`.

Chapter 2. Hook Functions

The primary coding task facing JAM programmers is writing hook functions. These functions, which are called by the JAM Executive and by the Screen Manager when certain well-defined events occur, are written in PL/1[5].

In this chapter, we discuss how hook functions are written and installed. They must also be compiled and linked into the JAM Application (or Authoring) Executable: see the Installation Guide for details of that. We also discuss what JAM events have hooks accessible to developers and what arguments are passed to hook functions from any given hook. Finally, we discuss in detail the various types of hook functions, showing examples of some of them, and explaining how they are installed and used.

2.1. PREPARATION AND INSTALLATION

Hook functions, once properly installed, are called at certain well- defined JAM events. These events are outlined below in section 2.1.1. and discussed in detail later in the chapter.

There are many events that have hooks accessible to developers. JAM passes different arguments to the various hook functions, and interprets the return codes differently for each one. It is important that hook functions process the arguments that are passed correctly, and that they return meaningful codes based on the events to which they are attached.

Hook functions are installed individually, and are called at runtime by JAM when a certain event type occurs. Most hook functions are called by the Screen Manager. However,

5. Hook functions may also be written in C and other third-generation programming languages for which JYACC supports a language interface In particular, Fortran, Cobol and PL/1 are available for JAM on some platforms

the hook functions invoked with control strings are called by the JAM Executive, and will only be accessible to applications using a custom executive through JPL.

2.1.1. Types of Hook Functions

There are twenty–two installable hook function types, six of which are installed when the application is bound and sixteen of which are installed as individual functions. They are briefly outlined below, and discussed in detail later in the document:

■`FIELD_FUNC`
: These functions are installed using the `-retain_all` argument of the `bind` command. The functions on this list may be designated in the Screen Editor to be called by the Screen Manager as field entry, exit or validation functions for specific fields. The JPL `atch` verb may also be used to access these functions.

■`GROUP_FUNC`
: These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be designated in the Screen Editor to be called by the Screen Manager as group entry, exit or validation functions for specific groups (Radio Buttons and Checklists).

■`SCREEN_FUNC`
: These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be designated in the Screen Editor to be called by the Screen Manager as screen entry or exit functions on particular screens.

■`CONTROL_FUNC`
: These functions are installed using the `-retain_all` argument of the `bind` command. These functions may be entered and invoked from control strings. They are often associated with function keys and menus in the Screen Editor or with the `xsm_putjctrl` library call. The JPL `call` verb can invoke control functions.

■`DFLT_FIELD_FUNC`
: This is an individual function. It is installed using the library routine `xsm_n_uinstall`. Once installed, it is called on entry, exit and validation for all fields.

■`DFLT_GROUP_FUNC`
: Similar to the `DFLT_FIELD_FUNC`, this individual function is called on entry, exit, and validation for all groups.

■`DFLT_SCREEN_FUNC`
Individual function called on entry and exit for all screens.

■`KEYCHG_FUNC`
Individual function called whenever JAM reads a key from the keyboard. This allows for the application to intercept and process (and possibly translate) keystrokes at the logical key level.

■`INSCRSR_FUNC`
Individual function called by JAM whenever the keyboard entry mode toggles between insert and overstrike mode. This allows an application to update the display, if desired, to provide an indication of the new mode. Often used if there is no ability to change cursor styles between insert and overstrike modes.

■`CKDIGIT_FUNC`
Individual function called by JAM for check digit validation of numeric fields. Only necessary if the default check–digit algorithm provided with JAM is not sufficient.

■`UINIT_FUNC`
Individual function called just before the Screen Manager and the physical display are initialized at the start of the application.

■`URESET_FUNC`
Individual function called just after the Screen Manager and the physical display are closed and reset at the end of the application, even if the application aborts ungracefully.

■`RECORD_FUNC`
Individual function used to record keystrokes so they can be played back for tutorials or for regression testing.

■`PLAY_FUNC`
Individual function used to playback recorded keys.

■`AVAIL_FUNC`
Individual function used in advanced record/playback algorithms.

■`STAT_FUNC`
Individual function used to intercept JAM status line processing and alter or replace it.

■`VPROC_FUNC`
Individual function used to intercept JAM video processing and to alter or replace it.

■`BLKDRVR_FUNC`
This is an individual function that acts as a block mode terminal driver. This is discussed in section 10.1.3.

■`ASYNC_FUNC`
Individual function called asynchronously when JAM is waiting for keyboard input. This is installed via the library routine `xsm_async`. Often used to poll external systems for mail delivery or the availability of data over a communications line.

2.1.2.
Installing Functions

As mentioned above, certain hook functions must be installed explicitly with the library routines `xsm_n_uinstall` or `xsm_async`, others are installed using the `-retain_all` argument of the `bind` command.

`xsm_n_uinstall` is called with three arguments. The first argument identifies the type of function being installed, and may be one of the following values:

```
UINIT_FUNC          CKDIGIT_FUNC        STAT_FUNC
URESET_FUNC         BLKDRVR_FUNC        DFLT_FIELD_FUNC
VPROC_FUNC          PLAY_FUNC           DFLT_SCREEN_FUNC
KEYCHG_FUNC         RECORD_FUNC         DFLT_GROUP_FUNC
INSCRSR_FUNC        AVAIL_FUNC
```

The second argument is the name of the function. The third argument identifies the language. This argument should be 1 for all programming languages except C.

`xsm_async` is used exclusively for installing asynchronous functions. It takes as arguments the address of the function and a timeout period.

The other function types, which are installed via the `-retain_all` argument to the `bind` command, are the following:

```
FIELD_FUNC
SCREEN_FUNC
CONTROL_FUNC
GROUP_FUNC
```

2.2.
WRITING HOOK FUNCTIONS

Arguments passed to hook functions and return values received from hook functions vary from hook to hook. In this section, we discuss the various JAM hooks in detail.

2.2.1. Field Functions

The Screen Manager will call field functions, if specified, on field entry, field exit, and field validation. Calls to field entry and field exit functions are guaranteed to be paired for any given field.

A single default field function may also be installed. It will be invoked on entry, exit, and validation for every field. The default field function must be installed explicitly as via `xsm_n_uinstall`.

JPL procedures may be directly specified as field functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl fieldfunc`.

Field Function Invocation

Field functions are called for field entry whenever the cursor enters a field, including when the field containing the cursor is activated by virtue of an overlying window being closed. Field functions are called for field exit whenever the cursor leaves a field, including when the field is exited because a window is popped up over the existing screen. Field functions are called for validation whenever the field is validated. This occurs at the following times:

- As part of field validation, when you exit the field or scroll to the next occurrence by filling it or by hitting TAB or RETURN key. The BACK-TAB and arrow keys do not normally cause validation. Field functions are called for validation only after the field's contents pass all other validations for the field.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for field validation.

Field functions may also be invoked from JPL with the `atch` verb.

For fields that are members of menus, radio buttons, or checklists, the validation function is not called as part of validation. The validation function for such fields is called instead when that field is selected. For checklist fields, the field validation function is also called when the field is deselected.

Field functions specified for field entry via the Screen Editor are invoked after any installed default field function. Field functions specified for field exit or validation via the Screen Editor are called before any installed default field function.

Field Function Arguments

All field functions receive four arguments:

1. The field number as an integer.
2. A buffer containing a copy of the field's contents.
3. The occurrence number of the data as an integer.
4. An integer bitmask containing contextual information about the validation state of the field and the circumstances under which the function was called.

The contextual information in the last parameter includes the following bit masks[7]:

■VALIDED

If this is set (i.e. if the 'bitwise and' of `param4` and `VALIDED` is not zero), the field has passed all its validations and has not been modified since.

■MDT

If this is set (i.e. if the 'bitwise and' of `param4` and `MDT` is not zero), the field data has been changed either from the keyboard or from the application code since the current screen was opened[8]. JAM never clears this bit. The application code may clear it directly with the `xsm_bitop` library routine.

■K_ENTRY

If set (i.e. if the 'bitwise and' of `param4` and `K_ENTRY` is not zero), the field function was called on field entry.

■K_EXIT

If set (i.e. if the 'bitwise and' of `param4` and `K_EXIT` is not zero), the field function was called on field exit[9].

■K_EXPOSE

If set (i.e. if the 'bitwise and' of `param4` and `K_EXPOSE` is not zero), the field function was called because a window overlying the screen on which the field resides was opened or closed[10].

■K_KEYS

Mask for the bits indicating which keystroke or event caused the field to be entered, exited, or validated. The intersection of this mask and the fourth pa-

NO TAG.
The example field function below contains a procedure called `bitmask` that is useful for checking whether a particular flag (bit location in a binary value) is set. Source code for this procedure can also be found in the `sample` application provided with JAM.

8. Note that when the screen is being opened, when the screen entry function modifies data in a field the `MDT` bit is not set. However, when the screen is exposed by virtue of an overlaid window being closed, modification of field data in the screen entry function will cause the `MDT` bit to be set.

9. Note that if neither `K_ENTRY` nor `K_EXIT` are set, the field is being validated.

10 This means that if both `K_ENTRY` and `K_EXPOSE` are set, the field is being exposed. If `K_EXIT` and `K_EXPOSE` are set, the field is being hidden.

rameter to the field function should be tested for equality against one of the six remaining values below:

■K_NORMAL
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_NORMAL`), a "normal" key caused the cursor to enter or exit the field in question. For field entry, "normal" keys are NL, TAB, HOME, and EMOH. For field exit, only TAB and NL are considered "normal".

■K_BACKTAB
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_BACKTAB`), the BACKTAB key caused the cursor to enter or exit the field in question.

■K_ARROW
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_ARROW`), an arrow key caused the cursor to enter or exit the field in question.

■K_SVAL
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_SVAL`), the field is being validated as part of screen validation.

■K_USER
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_USER`), the field is being validated directly from the application with the `xsm_fval` library routine.

■K_OTHER
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_OTHER`), some key other than backtab, arrow or those mentioned as "normal" caused the cursor to enter or exit the field in question.

Field functions are called for validation regardless of whether the field was previously validated. They may test the VALIDED and MDT bits to avoid redundant processing.

Field Function Return Codes

Field functions called on entry or exit should return 0. Field functions called for validation should return 0 if the field contents pass the validation criteria. Any non-zero return code should indicate that the field does not pass validation. If the returned value is 1, the cursor will not be repositioned to the offending field. Any other non-zero return value will cause the cursor to be repositioned to the field that failed the validation. This is useful because when the entire screen is undergoing validation, the field that fails validation may not be the field where the cursor is.[11]

Example Field Function

The following code illustrates how to interpret the fourth argument passed to a field function, and how to handle errors.

```
%include 'smdefs.incl.pl1';      /* basic JAM user defines */

apfunc1:
    procedure(field_number, field_data, occurrence, misc_bits)
    returns(fixed_binary(31));

    declare field_number                fixed_binary(31);
    declare field_data                  char(*) varying;
    declare occurrence                  fixed_binary(31);
    declare misc_bits                   fixed_binary(31);
    declare error                       fixed_binary(31);

    if bitmask(misc_bits, VALIDED)
        then return 0

    /* and later...     */
    /* check for error */

    if(error ^= 0)
        then do;
        xsm_gofield(1);
        xsm_quiet_err('Re-enter all data.');
        return(1);
        end;
    return(0);
end apfunc1;

    /* The following procedure checks if a particular flag is set.    */
    /* NOTE:  "unspec" only works on variables. Constants are passed */
    /* into bitmask as parameters, so bitmask will work with them.   */

bitmask:
    procedure(xbits,ybits)
    returns(bit(1));
    declare(xbits,ybits) fixed_binary(31);
    return((unspec(xbits) & unspec(ybits)) ^= '0'b);
end bitmask;
```

11 In many cases, it is better for the field validation function itself to reposition the cursor before displaying an error message, otherwise the error message might be misleading

2.2.2. Screen Functions

The Screen Manager will call screen functions, if specified, on entry and exit of screens. Calls to screen entry and screen exit functions are guaranteed to be paired for each screen.

A single default screen function may be installed. It will be invoked on entry and exit for every screen. The default screen function is installed as via `xsm_n_uinstall`. Screen functions specified as entry or exit functions for a screen via the Screen Editor are installed via the `-retain_all` argument to the `bind` command. JPL procedures may also be directly specified as screen functions in the Screen Editor by preceding their name with the string "`jpl `", for example `jpl screenfunc`.

Because of the way LDB processing and form stack handling is done, it is neither recommended nor supported to call any form or window display library routines from screen entry or exit functions. If it is necessary to display windows at screen entry, the library routine `xsm_ungetkey` can be invoked, passing as the argument a function key with a control string that brings up a window.

Screen Function Invocation

Screen functions are called for screen entry whenever a screen is opened. Screen functions are called for screen exit whenever a screen is closed. Optionally, screen functions may also be called for entry when a screen is exposed by virtue of a window overlaying it being closed or deselected, and called for exit when a window is popped up or selected over the screen in question. This is not the default behavior because it would introduce incompatibilities with earlier releases of JAM.

If you are not concerned with compatibility with earlier releases, it is strongly suggested that you make the following library function call near the beginning of your application, enabling the calling of screen functions when screens are exposed or hidden:

```
  xsm_option(EXPHIDE_OPTION, ON_EXPHIDE)
```

Screen functions specified for screen entry via the Screen Editor are invoked after any installed default screen function. Screen functions specified for screen exit via the Screen Editor are called before any installed default screen function.

Screen Function Arguments

All screen functions receive two arguments:

1. The screen name.
2. An integer bitmask containing contextual information about the circumstances under which the function was called.

The contextual information in the second parameter includes the following bit masks:

■`K_ENTRY`
: If this is set (i.e. if the 'bitwise and' of `param4` and `K_ENTRY` is not zero), the function was called on screen entry.

■`K_EXIT`
: If this is set (i.e. if the 'bitwise and' of `param4` and `K_EXIT` is not zero), the function was called on screen exit.

■`K_EXPOSE`
: If this is set (i.e. if the 'bitwise and' of `param4` and `K_EXPOSE` is not zero), the function was called because the screen was selected or deselected, or because a window was popped over the screen or a window that used to be overlaid on the screen was closed[12].

■`K_KEYS`
: Mask for the bits indicating which event caused the screen to be exited. The intersection of this mask and the second parameter to the screen function should be tested for equality against one of the two remaining values below:

■`K_NORMAL`
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_NORMAL`), a "normal" call to `xsm_close_window` caused the screen to close.

■`K_OTHER`
: If set (i.e. if the 'bitwise and' of `param4` and `K_KEYS` equals `K_OTHER`), the screen is being closed because another form is being displayed or because `xsm_resetcrt` is called.

Screen Function Return Codes

All screen functions should return 0.

2.2.3. Control Functions

Control functions are called by the JAM Executive in the processing of control strings and by JPL routines that call PL/1 functions. The JAM Executive will call control functions, if specified and installed, when control strings that start with a caret (^) are executed. JPL procedures may also execute control functions by using the `call` verb.

12. If both `K_ENTRY` and `K_EXPOSE` are set, the screen is being uncovered and activated by virtue of an overlaid window being closed. If both `K_EXIT` and `K_EXPOSE` are set, the screen is being covered and deactivated by virtue of a window being popped up over it.

There is no default control function. Control functions are installed via the `-retain_all` argument to the `bind` command. JPL procedures may be directly specified as control functions by preceding the name of the procedure in a control string with the string "`jpl `".

A number of control functions of general use are built in to JAM. These built-ins can be used by any JAM application. They are listed in Chapter 4.

Control Function Invocation

Control functions are called by the JAM Executive when a control string starting with a caret is processed. Such control strings are often attached, via the Screen Editor, to function keys or to menu selections in control fields. In addition, the JPL verb `call` can be used to invoke control functions.[13]

Control Function Arguments

Control functions receive a single argument, namely a buffer containing a copy of the control string that invoked the function, without the leading caret. It is only the first word on the control string that identifies the function, the rest of the string may contain arbitrary data that can be parsed and used as arguments.

Control Function Return Codes

Control functions may return any integer. The return value from a control function may be used for conditional control branching in target lists (see the Authoring Guide). If there is no target list, and the control string returns a function key which has an associated control string in it's own right, then that control string is executed.

2.2.4. Key Change Functions

The key change function is called by the Screen Manager as keys are read from the keyboard from within the library routine `xsm_getkey`, which is called in the input processing for all keys by JAM. Only one individual keychange function may be installed at a time.

Keys placed on the queue with the library routine `xsm_ungetkey` or with the built-in control function `^jm-keys` are not processed by the installed key change function.

13 The JPL call verb does not execute control strings. It looks for functions to call.

The key change function is installed as `KEYCHG_FUNC` via `xsm_n_uinstall`.

Key Change Function Invocation

The key change function is called exactly once for every key read in from the keyboard or supplied by the playback hook function described in section 2.2.10..

Key Change Function Arguments

The key change function is passed a single integer argument, namely the JAM logical key that was read from the keyboard or received from the playback hook function.

Key Change Function Return Codes

The key change function returns the key to be substituted for the one passed as an argument. Any key returned to `xsm_getkey` will be returned by `xsm_getkey` to its caller. However, if the key change function returns 0, `xsm_getkey` will get the next key from the keyboard[14].

2.2.5. Group Functions

The Screen Manager will call group functions, if specified, on entry, exit, and validation of radio buttons and checklists. Calls to group entry and group exit functions are guaranteed to be paired for each group.

A single default group function may be installed. It will be invoked on entry, exit, and validation for every group. The default group function is installed as via `xsm_n_uinstall`. Group functions specified as entry, exit, or validation functions for a given group in the Screen Editor are installed via the `-retain_all` argument to the `bind` command. JPL procedures may also be directly specified as group functions in the Screen Editor by preceding their name with the string "`jpl` ", for example `jpl groupfunc`.

Please note that field validation functions for fields that are members of groups or menus are called at selection and, in the case of checklists, deselection as discussed above in section 2.2.1. on page 11.

Group Function Invocation

Group functions are called for group entry whenever the cursor enters a group, including the times when the group containing the cursor is activated by virtue of an overlying win-

14. See the library routine `XSM_KEYOPTION` for a different method of changing the function of a logical key.

dow being closed. Group functions are called for group exit whenever the cursor leaves a group, including the times when the group is left because a window is popped up over the existing screen. Group functions are called for validation whenever the group is validated. This occurs at any of the following times:

- As part of group validation, when you exit the group by hitting TAB or making a selection from an autotab group. The BACKTAB and arrow keys do not normally cause validation.
- As part of screen validation when the XMIT key is struck.
- When the application code calls library routines for group validation.

Group functions specified for group entry via the Screen Editor are invoked after any installed default group function. Group functions specified for group exit or validation via the Screen Editor are called before any installed default group function.

Group Function Arguments

All group functions receive two arguments:

1. The group name.
2. An integer containing contextual information about the validation state of the group and the circumstances under which the function was called.

The information contained in the third argument to group functions is identical to that passed in the fourth argument to field functions. See section 2.2.1. on page 11 for an explanation.

Group functions are called for validation regardless of whether the group was previously validated. They may test the `VALIDED` and `MDT` bits to avoid redundant processing.

Group Function Return Codes

Group functions called on entry or exit should return 0. Group functions called for validation should return 0 if the group selections pass the validation criteria. Any non-zero return code should indicate that the group does not pass validation. If the returned value is 1, the cursor will not be repositioned to the offending group. Any other non-zero return value will cause the cursor to be repositioned to the group that failed the validation.

2.2.6. Asynchronous Functions

The installed asynchronous function is called periodically by the Screen Manager while the keyboard input routine waits for user input. It can be used to poll or otherwise manipulate communications resources, or to update the display on the screen.

The asynchronous function is installed individually as `ASYNC_FUNC` via the library routine `xsm_async`.

Asynchronous Function Invocation

The asynchronous function is called from the very lowest level of JAM keyboard input. When the asynchronous function is installed, the device driver clock on the terminal input device is set to time out on its character read operation, and if a character is not read in that time interval the asynchronous function is invoked before another character read operation is attempted. The time out interval is specified when the function is installed. The time out is measured in tenths of seconds. The maximum interval is 255 (25.5 seconds).

Asynchronous Function Arguments

The asynchronous function is passed no arguments.

Asynchronous Function Return Codes

The asynchronous function should generally return 0. If it returns –1, it will not be called again until at least one additional character has been read from the keyboard. The asynchronous function may return a key, which will be returned to `xsm_getkey` and on to the application. If that key is a JAM logical key, no further translation will be done. If the asynchronous function returns a data character, JAM will interpret it as a physical keyboard stroke.

2.2.7. Insert Toggle Functions

The Screen Manager will call the Insert Toggle Function when switching between input and overstrike mode for data entry. Generally this hook function will be used to update some aspect of the display informing the user of the current mode.

The insert toggle function is installed individually as `INSCRSR_FUNC` via `xsm_n_uinstall`. JAM automatically installs an insert toggle function that changes the cursor style when the mode is changed. If an application installs its own insert toggle function, the JAM function will be de–installed, and the new insert toggle function may want to call the function directly.

Insert Toggle Function Invocation

The function will be invoked by JAM whenever the data entry mode shifts from insert to overstrike mode or from overstrike to insert mode. Most often, this occurs when the end–user strikes the INSERT key.

Insert Toggle Function Arguments

One integer argument is passed to the insert toggle function. It specifies the mode. If its value is 1, JAM is entering insert mode. If it is 0, JAM is entering overstrike mode.

Insert Toggle Function Return Codes

The insert toggle function should return 0.

2.2.8. Check Digit Functions

The Screen Manager will call the check digit function for any field that is marked for check digit in the Screen Editor. It may be used to implement any desired check–digit algorithm. If there is no check digit function installed in the application, JAM will use the default library function `xsm_ckdigit`. A new check digit function is installed as `CKDIGIT_FUNC` via the library routine `xsm _n_uinstall`.

Check Digit Function Invocation

The check digit function is called by JAM during validation of fields marked for check digit.

Check Digit Function Arguments

The check digit function is passed the following arguments:

1. The integer number of the field undergoing validation.
2. The field contents.
3. The integer occurrence number for the data undergoing validation.
4. The integer modulus as specified in the Screen Editor.
5. The integer minimum number of digits as specified in the Screen Editor.

Check Digit Function Return Codes

The check digit function should return 0 if the field passes the check digit validation. If a non–zero value is returned, the cursor is positioned to the offending field and the field is not marked as validated. It is assumed that the check digit function display its own error messages.

2.2 9.

Initialization and Reset Functions

The initialization and reset functions are called by the Screen Manager on display setup and display reset respectively. The initialization function can be used to set the terminal type and the reset function can be used to handle any cleanup that the application needs to do whether it is terminated gracefully or not.

Initialization and reset functions are installed individually as `UINIT_FUNC` and `URESET_FUNC` respectively via calls to `xsm_n_uinstall`.

Initialization and Reset Function Invocation

The initialization function is called from the library routine `xsm_initcrt`. When it is called, JAM has not yet allocated its required memory structures, and the physical display characteristics are still untouched by JAM. In general, it is suggested that hook functions be installed after initialization with `xsm_initcrt`, but clearly this is an exception. The initialization function must be installed before `xsm_initcrt` is called. This function is installed as `UINIT_FUNC` via the library routine `xsm_n_uinstall`.

The reset function is called from the library routine `xsm_resetcrt` after JAM has released its memory and reset the physical display characteristics. Since the JAM abort routine `xsm_cancel` calls `xsm_resetcrt` before the application terminates, the reset function is generally called at application exit whether the exit is graceful or not[15]. This function is installed as `URESET_FUNC` via the library routine `xsm_n_uinstall`.

Initialization and Reset Function Arguments

The initialization function is passed a single argument, namely a 30 byte character buffer into which it may place the null-terminated string mnemonic identifying the terminal type in use. This is primarily of use on operating systems without an environment. This function can be used to obtain the terminal type in some system-specific way.

The reset function is passed no arguments.

Initialization and Reset Function Return Codes

Both the initialization and reset hook functions should return 0.

15 Interrupt handlers may need to be set by the developer to insure that `XSM_CANCEL` is called at all the necessary hardware and software interrupt signals. It is suggested that this setup be done in the function installed as an initialization function.

2.2.10. Recording and Playing Back Keystrokes

The Screen Manager provides hooks for recording and playing back keystrokes. This facility could be used to implement simple macro capabilities, or to perform regression testing on a JAM application. The developer should ensure that the record and playback functions are not in use simultaneously.

Record and playback functions are installed individually as `RECORD_FUNC` and `PLAY_FUNC` respectively via `xsm_n_uinstall`.

Record/Playback Function Invocation

The record function is called from `xsm_getkey` when it has a translated key value in hand that it is about to return to the application. The playback function is called from `xsm_getkey`, when installed, in place of a read from the keyboard[16]. For accurate regression testing, the playback function may need to pause and flush the output to simulate a realistic rate of typing, and may need to call the asynchronous function, if there is one.

Record/Playback Function Arguments

The record function is passed a single integer, which is the JAM logical key to record. Generally that key is recorded in some fashion for a possible playback at a later date. The playback function receives no arguments.

Record/Playback Function Return Codes

The record function should return 0. The playback function should return the logical key that was recorded at an earlier time.

2.2.11. Status Line Functions

The status line function is called by the Screen Manager whenever the status line is about to be flushed, or physically written to the terminal device. It is intended for use on terminals that require unusual status line processing, beyond the scope of the generic code, but other uses are possible.

16. Since characters are recorded after processing by the key change function but played back before key change translation, some key change functions may interfere with the accurate playback of recorded keystrokes. See the description of `XSM_GETKEY` in the Programmer's Reference Manual for more information.

The status line function is installed individually as `STAT_FUNC` via `xsm_n_uinstall`.

Status Line Function Invocation

The status line function is called when the status line is about to be physically written to the terminal display. Because of delayed write, this may or may not be at the time when the functions that specify message line text are actually called.

Status Line Function Arguments

The status line function receives no arguments. It can access copies of the text and attributes about to be flushed to the status line using the following library routine calls:

```
  stat_text = xsm_pinquire(SP_STATLINE);
  stat_attr = xsm_pinquire(SP_STATATTR);
```

Status Line Function Return Codes

If the status line function returns 0, JAM continues its usual processing and actually writes out the status line. If the function returns any other value, JAM assumes that the physical write of the status line was handled in the hook function.

2.2.12. Video Processing Functions

The Screen Manager calls the developer–installed video processing function to allow for special handling of various video sequences by the application. This is a specialized hook required only when the JAM video file is unable to provide support for a particular type of terminal.

The video processing function is installed individually as `VPROC_FUNC` via `xsm_n_uinstall`.

Video Processing Function Invocation

The video processing function is called by JAM's output routine just before a video output operation is about to begin.

Video Processing Function Arguments

The video processing function receives two arguments. The first is an integer video processing code defined in the header file `smvideo.incl.pl1` and outlined in the table below. The second is an array of integers with parameters for the video processing code. The number of parameters passed depends on the operation as shown in the table below. For video processing codes that require no arguments, a NULL is passed.

Code	*Operation Description*	*# of params*
`V_ARGR`	remove area attribute	
`V_ASGR`	set area graphics rendition	11
`V_BELL`	visible alarm sequence	
`V_CMSG`	close message line	
`V_COF`	turn cursor off	
`V_CON`	turn cursor on	
`V_CUB`	cursor back (left)	1
`V_CUD`	cursor down	1
`V_CUF`	cursor forward (right)	1
`V_CUP`	set cursor position (absolute)	2
`V_CUU`	cursor up	1
`V_ED`	erase entire display	
`V_EL`	erase to end of line	
`V_EW`	erase window to background	5
`V_INIT`	initialization string	
`V_INSON`	set insert cursor style	
`V_INSOFF`	set overstrike cursor style	
`V_KSET`	write to soft key label	2
`V_MODE4`	single character graphics mode (also `V_MODE5, 6`)	
`V_MODE0`	set graphics mode (also `V_MODE1, 2, 3`)	
`V_OMSG`	open message line	

Code	*Operation Description*	*# of params*
`V_RESET`	reset string	
`V_RCP`	restore cursor position	
`V_REPT`	repeat character sequence	2
`V_SCP`	save cursor position	
`V_SGR`	set latch graphics rendition	11

Video Processing Function Return Codes

When the video processing function returns 0, JAM will continue with normal processing. If it returns any other value, JAM will assume that the operation has been handled in the hook function. This allows the developer to implement only necessary operations.

Other Hook Functions

The Screen Manager provides an additional hook to handle block mode terminals. This function is best viewed as a driver. Block mode is described in Chapter 10.

2.3. CODING STRATEGY, RULES AND PITFALLS

2.3.1. Displaying Screens

There are a number of library functions provided for the display of screens as forms or windows. In general, the following rules and guidelines should be followed in choosing between them and deciding when they can be used:

- The display of screens as forms or windows from within screen functions at screen entry or screen exit is neither recommended nor supported.

- The routines `xsm_jform`, `xsm_jwindow`, and `xsm_jclose` are provided specifically for the display and destruction of screens in applications that use the JAM Executive. Applications not using the JAM Executive should not use these routines. They are recommended over the other screen display routines in applications that do use the JAM Executive.
- The form display routine `xsm_jform` manipulates the form stack appropriately. The use of any other form display routines in applications that use the JAM Executive will exhibit unexpected behavior, as the form stack will not be synchronized with the application flow.

2.3.2. Recursion

The developer should be careful, when using hook functions, to avoid the recursion that will come from nested hook function calls. Such recursion will not be easy to detect in the source code itself: some understanding of the product mechanism is required.

For example, care should be taken when writing record, playback, or key change functions that read from the keyboard, or status line functions that themselves cause the status line to be flushed. A default screen entry function that in and of itself opens new screens could be a problem.

Chapter 3.
Local Data Block

The Local Data Block, or LDB, is a region of memory for the storage of JAM field data that is generally shared between screens. It is discussed in the JAM Development Overview and in the Author's Guide.

3.1. LDB CREATION

The LDB is created with the library routine call `xsm_ldb_init`. This routine searches for a data dictionary file created from the authoring tool with the Data Dictionary Editor. For more information about the data dictionary and the Data Dictionary Editor, see the Author's Guide.

If the data dictionary file is found, it is read and a single LDB entry is created in memory for every data dictionary entry that has a non–zero scope. Note that only the name of the LDB entry is placed in memory, storage for the field data that is stored with the entry is not allocated until the entry is used.

After it is created, the LDB is initialized from ASCII text files. These files, described in the Author's Guide, contain pairs of LDB names and values. The LDB entries named are filled with the values that follow them in the files.

3.2. HOW JAM USES THE LDB

JAM uses the LDB for the storage and propagation of field data from screen to screen in the application. Every time a screen is opened, or exposed by the closing of a window that covers it, every field on the screen named identically to an LDB entry is filled with the value of the LDB entry. This occurs after the screen entry function is called.

Correspondingly, every time a screen is closed, or hidden when a window pops up over it, every LDB entry that is named identically to a field on the screen is filled with the value of the screen field. This occurs before the screen exit function is called.

When a screen is populated from the LDB at screen entry time, there is a subtle difference between a new screen being opened and a screen being exposed when a covering window is closed. When a screen is newly opened, only empty fields with corresponding LDB entries will be populated from the LDB. When a screen is exposed, all fields that have corresponding LDB entries will be populated.

3.3. LDB ACCESS

Data in the LDB can be accessed with the library routines `xsm_n_getfield`, `xsm_n_putfield`, `xsm_i_getfield`, `xsm_i_putfield`, and related functions that access data by field name. These routines access the data on the current screen if the field that is named exists on the current screen. If the field does not exist on the current screen, these routines access the LDB.

During screen entry and exit processing only, the search order is reversed. During the screen entry and exit functions, these access routines first search the LDB and then search the screen. This is because the LDB is merged to the screen after the screen entry function, and the screen is stored to the LDB before the screen exit function. If the search order were not reversed the data accessed would be invalid[18].

18 This could, in a very small number of cases, introduce some incompatibilities with applications that were written with earlier releases of JAM If such compatibility problems arise, use the library function `XSM_OPTION` setting the option `ENTEXT_OPTION` to `FORM-FIRST`.

Chapter 4. Built–in Control Functions

This section describes control functions supplied with JAM. Note that the synopsis is for a JAM control string, not a programming language source statement. The return value of a control function can be used in a target list; see the Author's Guide for information on control strings and target lists.

You may use these functions in control strings and in JPL `call` statements.

jm_exit

end processing and leave the current screen

SYNOPSIS

```
^jm_exit
```

DESCRIPTION

Clears the current form or window and returns to the previous one. If the current form is the application's top-level form, JAM will 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 replaced by another form or a window:

```
^(0=&w2)jm_exit
```

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 on the form stack and windows on the window stack 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`.

```
+---------------------------------------------------+
:                                                   :
:  Query customer database__    custquery.jam__     :
:  Update customer database_    custupdate.jam_     :
:  Free-form query__________    'sql___________     :
:  Return to previous menu__    ^jm_exit_______     :
:  Return to main menu______    ^jm_gotop______     :
:                                                   :
+---------------------------------------------------+
```

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 all open windows 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
```

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 `xsm_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.incl.pl1* . Data characters should be enclosed between apostrophes ' ', back-quotes ` `, or double quotes " ". This function passes its arguments to `xsm_ungetkey` in reverse order, so you supply them in the natural order.

`jm_keys` will process a maximum of 20 keys. This limit includes function keys plus characters contained in strings.

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
```

jm_mnutogl

switch between menu and data entry mode on a dual–purpose screen

SYNOPSIS

```
^jm_mnutogl {screen-mode}
```

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. This function performs the same function as the MTGL logical key.

An optional argument may be specified which will force the screen into a particular mode, regardless of its current state. To specify menu mode, use the argument 'M' (or 'm'). To specify open–keyboard (data entry) mode, use the argument 'O' (or 'o').

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
```

jm_winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
^jm_winsize
```

DESCRIPTION

Calling `jm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows may be specified as siblings by specifying `&&` in a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the Author's Guide for further information. This function parallels the library routine `xsm_winsize`.

jpl
invoke a JPL procedure

SYNOPSIS

```
^jpl procedure [ argument ... ]
```

DESCRIPTION

This function invokes a procedure written in the JYACC Procedural Language. procedure should be the name of a JPL procedure or module; anything following that will be passed to the procedure as arguments. See the JPL Programmer's Guide for the rules used by the JPL interpreter to determine which JPL procedure is executed. The value returned by your procedure will be returned by `jpl` for use in a target list.

This function is similar to the JPL `jpl` command. Colon expansion is done on the arguments.

EXAMPLE

The control string below invokes a JPL function to concatenate two strings and store the result in `target`.

```
^jpl concat target "king" "kong"
```

Chapter 5. ***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 the chapter "Writing and Installing Hook Functions".

5.1. LOGICAL 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 physical keys or sequences of physical 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

5.2. 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 the configuration guide for details). The table itself is derived from an ASCII file which can be modified by any editor; a screen-oriented utility, `modkey`, is also supplied for creating and modifying key translation tables (see the Utilities Guide).

JAM assumes that the first character of any multi-character key sequence to be translated to a single logical key 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.

Upon receipt of a control character, the keyboard input function `xsm_getkey` searches the translation table. If no match is found on the first character, the key is accepted without translation. If a full match is found on the first character, an exact match has been found, and `xsm_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 `xsm_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 `xsm_getkey` to distinguish, for example, between the sequences `ESC O x`, `ESC [ A`, and `ESC [ 1 0 ~`.

5.3. KEY ROUTING

The main routine for keyboard processing is `xsm_input`. This routine calls `xsm_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, `xsm_input` returns to the caller with this value as the return code.

If the value is between 0x01 and 0x1ff, the key is first translated via the key translation table. This table is changed with the library routine `xsm_keyoption`. Then processing is determined by a routing table. Use `xsm_keyoption` to get and set the routing information for a particular key. The routing value consists of two bits, examined independently, so four different actions are possible:

1. If neither bit is set, the key is ignored.
2. 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.
3. If the RETURN bit is set, `xsm_input` returns the logical value to the caller; otherwise, `xsm_getkey` is called for another value.
4. 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 default setting for displayable characters is EXECUTE. All other ASCII and exteneded ASCII characters are ignored. 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. All other function keys (EXIT, SPGU etc...) are initially set to EXECUTE.

Applications can change key actions on the fly by using `xsm_keyoption`. For example, to disable the backtab key the application program would execute

```
call xsm_keyoption(BACK, KEY_ROUTING, KEY_IGNORE)
```

To make the field erase key return to the application program, use

```
call xsm_keyoption(FERA, KEY_ROUTING, RETURN)
```

Key mnemonics can be found in the file `smkeys.incl.pl1`.

Chapter 6. Terminal Output Processing

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 `xsm_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 `xsm_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 detailed in the Video Manual; the following sections describe additional ways in which applications may use the information encoded in the video file.

6.1. 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.

6.2. 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 `xsm_err_reset` or a related function
2. Ready/wait status
3. Messages installed with `xsm_d_msg_line` or `xsm_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: `xsm_query_msg`, `xsm_err_reset`, `xsm_emsg`, `xsm_quiet_err`, and `xsm_qui_msg`. `xsm_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.

`xsm_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 `xsm_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 `xsm_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 `xsm_c_vis`.

During debugging, calls to `xsm_err_reset` or `xsm_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 `xsm_initcrt`. `xsm_err_reset` and `xsm_quiet_err` can be called with a message text that is defined locally, as in:

```
  call xsm_err_reset("ZIP CODE INVALID FOR THIS STATE.");
```

However, the JAM library functions use a set of messages defined in an internal message table. This table is accessed by the function `xsm_msg_get`, using a set of defines in the header file `smerror.incl.pl1`. The return value from `xsm_msg_get` can be used as input for one of the status line functions.

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.

Chapter 7. Writing International (8 bit) Applications

7.1. INTRODUCTION

This chapter describes how to use the 8 bit internationalization capabilities that have been incorporated into JAM Release 5.

From the point of view of someone who has used JAM without these features, a few differences will be apparent immediately. Other, more subtle, differences will emerge as the package is used in building language–independent applications. Finally, many of the changes were made so that the development utilities could be localized for use in other countries. These will largely go unnoticed by people using the package in English.

7.1.1. General Overview

The purpose of the 8 bit NLS is to allow the JAM product and applications created with with it to be "localized" for use in non–English–speaking countries. This means that the product can be made to look like it originated in the country in which it is being used. All prompts and messages can appear in the appropriate language and customs for formatting dates, currency fields and the like can be observed. Notwithstanding this, many of the features that are only visible to programmers will continue to be in English since many programmers are used to working in English.

The capabilities described are limited to languages in which characters can be represented in 8 bits of information and those that use a left–to–right entry order. This eliminates the complexities associated with many far– and middle–eastern languages.

7.2. LOCALIZATION

JAM and JAM applications can be localized by taking the following steps:

- Use the Screen Editor to translate all screens in the application.
- Modify and recompile the message file.
- Translate the documentation.

7.2.1. Background

The JAM product was originally developed with some internationalization issues in mind. It has always used 8 bit character data, without appropriating a bit for internal use. So one of the major demands of the international market was already satisfied.

Date and time formats have always been completely specified by the screen creator. The wide variety of formats available in Release 4 could satisfy most requirements. In Release 5, additional capabilities were added to make it easier to convert screens from one language to another. Currency formats were the least international of the features in the Release 4 product. Release 5 makes these completely language independent.

Each of the sections below discusses some aspect of internationalization.

7.2.2. 8 Bit Character Data

As pointed out in the introduction, JAM supports 8 bit character data. Video files specific to the terminal can give special instructions, if necessary, as to how to display international characters. This is needed if the terminal requires shifting to a different character set to display non–ASCII characters. Most terminals used in the international market will not need to shift character sets.

The video file can also be used to translate between two different standards for international characters. Thus the screens could be created with one standard and displayed using a different one.

The use of 8 bit characters for international symbols does not necessarily preclude the use of graphics for borders, etc. Any unused entries in character set (e.g. 0x01 – 0x1f, or 0x80 – 0x9f) can be mapped to line graphics symbols.

JAM rarely, if ever, interprets characters present in screens or entered from the keyboard. Internally it merely manipulates numbers. Any meaning as an alphabetic character, graphics symbol, or whatever, is generally irrelevant to JAM. The cursor control keys (arrows, tab, etc.), function keys, and soft keys are all assigned logical values that are outside the range 0x00 to 0xff, and thus cannot conflict with international characters.

Keyboards that support international character sets will usually produce a single (8 bit) byte (perhaps with the high bit set) for each character. However there are some terminals that generate a sequence to represent an international character. If so, `modkey` (or a text editor) would be used to map the byte sequences into a logical value, just as the video file would be used to map the logical value to the sequence required by the display terminal.

If you have questions about how to display non–English characters or to receive them from the keyboard, consult the chapters on keyboard and video processing.

7.2.3. Date And Time Fields

Date and Time fields have been completely revamped in Release 5. They have been combined to enable one field to have both date and time information. This, and the fact that more flexibility was added to date and time formatting, required changes to the date and time mnemonics. For example, in Release 4, the mnemonic `mm` was used for a 2–digit month in Date fields as well as the specifier for minutes in Time fields. Clearly, this cannot serve both purposes when the fields are combined.

In Release 5, the mnemonics for specifying date and time formats are stored in the message file so they may be changed. In addition, they are stored in a "tokenized" form internally which provides two major benefits. First, the need to parse the formats at runtime is eliminated, thus speeding up processing and reducing memory requirements. Second, screen designers in different countries editing the same screen will all see date and time specifications in formats they are used to. For example, if an English screen designer created a date field with the format `mon/day/year`, it might show up on a French system as `mois/jour/annee`.

The problem of interchanging the month and day is dealt with later.

The table below shows the default message file entries for date and time mnemonics:

Msg # Mnemonic	*Date/Time Mnemonic*	*Tokenized Format*	*Description*
FM_YR4	YR4	%4y	4 digit year
FM_YR2	YR2	%2y	2 digit year
FM_MON	MON	%m	month number
FM_MON2	MON2	%0m	month number, zero fill
FM_DATE	DATE	%d	date (day of month)
FM_DATE	DATE2	%0d	date, zero fill
FM_HOUR	HR	%h	hour
FM_HOUR	HR2	%0h	hour, zero fill
FM_MIN	MIN	%M	minute
FM_MIN2	MIN2	%0M	minute, zero fill
FM_SEC	SEC	%s	seconds
FM_SEC2	SEC2	%0s	seconds, zero fill
FM_YRDA	YDAY	%+d	day of the year
FM_AMPM	AMPM	%p	am/pm
FM_DAYA	DAYA	%3d	abbreviated day name
FM_DAYL	DAYL	%*d	long day name
FM_MONA	MONA	%3m	abbrev. month name
FM_MONL	MONL	%*m	long month name

Thus, a date field specified as mm/dd/yyyy in Release 4 would be MON2/DATE2/YR4 in Release 5. The f4to5 conversion program will convert the format to %m/%d/%4y internally so it will automatically show up correctly when the screen is edited. The mnemonics were chosen to correspond to ANSI standards. You can change them to suit your own needs by simply changing the message file and running msg2bin. To change the mnemonic for a 4 digit year from YR4 to YYYY, for example, change the message file line

```
FM_YR4 = YR4
```

to

```
FM_YR4 = YYYY
```

and run `msg2bin`.

If all development is done in one language, the fact that different mnemonics for date and time formats can be used for different languages is unimportant. What is important, however, is to be able to modify an application to operate in a different language. The goal is that only the text of the screens and the message file should need to be changed.

Consider a screen with a date field of the form `DAYA MONA DATE, YR4`. If executed on a system with an English message file it might appear as

```
Mon Apr 4, 1989
```

whereas on a French system it would be

```
Lun Avr 4, 1989
```

This happens without changing the date format. All that has changed are the names and abbreviations of the months and days which are also stored in the message file so it is a simple matter to convert them.

Now consider a date field which in English should show up in `mm/dd/yyyy` form but should appear in French as `dd-mm-yyyy`. In this case, the date format itself would have to be modified. For this reason, 10 additional formats are supplied for the designer's use. For instance, in the message file the designer can specify a new date mnemonic called `REGULAR DATE`. In the English message file this can be equated to `mm/dd/yyyy` and in the French message file to `dd-mm-yyyy`. Thus, if the date format is specified as `REGULAR DATE`, only the message file, not the screen, needs to be changed to convert the date field to French.

For this capability, both the mnemonics *and* what they represent are specified in the message file. The actual formats are stored in the message file in tokenized form so that there is no need for a parser.

The following table shows the default message file entries for these extra date mnemonics:

Msg Number Mnemonic	Date/Time Mnemonic	Tokenized Form	Corresponding Msgfile Entry	Default
FM_0MN_DEF_DT	DEFAULT	%0f	SM_0DEF_DTIME	%m/%d/%2y %h:%0M
FM_1MN_DEF_DT	DEFAULT DATE	%1f	SM_1DEF_DTIME	%m/%d/%2y

Msg Number Mnemonic	Date/Time Mnemonic	Toke-nized Form	Corresponding Msgfile Entry	Default
FM_2MN_DEF_DT	DEFAULT TIME	%2f	SM_2DEF_DTIME	%h:%0M
FM_3MN_DEF_DT	DE-FAULT3	%3f	SM_3DEF_DTIME	%m/%d/%2y %h:%0M
FM_4MN_DEF_DT	DE-FAULT4	%4f	SM_4DEF_DTIME	%m/%d/%2y %h:%0M
FM_5MN_DEF_DT	DE-FAULT5	%5f	SM_5DEF_DTIME	%m/%d/%2y %h:%0M
FM_6MN_DEF_DT	DE-FAULT6	%6f	SM_6DEF_DTIME	%m/%d/%2y %h.%0M
FM_7MN_DEF_DT	DE-FAULT7	%7f	SM_7DEF_DTIME	%m/%d/%2y %h:%0M
FM_8MN_DEF_DT	DE-FAULT8	%8f	SM_8DEF_DTIME	%m/%d/%2y %h.%0M
FM_9MN_DEF_DT	DE-FAULT9	%9f	SM_9DEF_DTIME	%m/%d/%2y %h:%0M

Thus, if the screen designer specifies a date field with the format `DEFAULT DATE`, it would show up in `mm/dd/yy` form. If the line

```
SM_1DEF_DTIME = %m/%d/%2y
```

in the message file were changed to

```
SM_1DEF_DTIME = %d-%m-%2y
```

the date would show up in `dd-mm-yy` form. To change the mnemonic for this date format to `REGULAR DATE`, the message `FM_1MN_DEF_DT` should be modified.

7.2.4. Currency Fields

Like Date and Time fields, Currency fields have been modified in Release 5. Since it is not uncommon in Europe to be dealing with several currencies simultaneously, release 5 does not force any one system on the screen creator. Thus, the formatting capabilities were enhanced to support any convention the screen creator might desire. As with date and time formats, a "default" format is supplied that causes the actual format to be taken from the message file. For Currency fields however, this option is supplied only for the parts of the format that may vary from one currency to another.

The new release allows the following items to be specified for Currency fields:

- the decimal symbol (usually dot or comma)
- minimum number of decimal places
- maximum number of decimal places
- thousands separator (usually dot or comma; b = blank)
- the currency symbol to be used (up to 5 characters)
- the placement of that symbol (left, right or at decimal pt)
- default currency from the message file (to replace the above entries)
- rounding (round–up, round–down, round–adjust)
- fill character
- justification
- clear if zero
- apply if empty

There is a slight problem in specifying currency symbols when using the Screen Editor. Since the currency symbol is entered into a regular field, it is not possible to enter trailing spaces (they are always stripped off). Thus, to specify a leading currency symbol separated from the data by a space (`FF 123.456,78`) you must use the message file. For this reason, the dot (`.`) may be used to signify a space when entered into the currency field. A dot in the message file for this purpose will appear as a dot.

The default currency formats are strings of the form ***rmxtpccccc*** where:

■***r***	= decimal symbol (usually comma or dot)
■***m***	= minimum number of decimal places
■***x***	= maximum number of decimal places
■***t***	= thousands separator (usually comma or dot; b = blank)
■***p***	= placement of currency symbol (l, r or m)
■***ccccc***	= up to 5 characters for the currency symbol

Thus, if the screen designer specifies a currency field with the format CURRENCY, it would show up in `$999,999.99` form. If the line

```
SM_0DEF_CURR = ".22,1$"
```

in the message file were changed to

```
SM_0DEF_CURR = ",22.1FF"
```

the field would show up as `FF 999.99,99`. To change the mnemonic for this currency field, the message `FM_0MN_CURRDEF` should be modified. The following table shows the default message file entries for the currency mnemonics:

Msg Number Mnemonic	*Currency Mnemonic*	*Corresponding Msgfile Entry*	*Default*
FM_0MN_CURRDEF	CURRENCY	SM_0DEF_CURR	.22,1$
FM_1MN_CURRDEF	NUMERIC	SM_1DEF_CURR	.09,
FM_2MN_CURRDEF	PLAIN	SM_2DEF_CURR	.09
FM_3MN_CURRDEF	DEFAULT3	SM_3DEF_CURR	.09
FM_4MN_CURRDEF	DEFAULT4	SM_4DEF_CURR	.09
FM_5MN_CURRDEF	DEFAULT5	SM_5DEF_CURR	.09
FM_6MN_CURRDEF	DEFAULT6	SM_6DEF_CURR	.09
FM_7MN_CURRDEF	DEFAULT7	SM_7DEF_CURR	.09
FM_8MN_CURRDEF	DEFAULT8	SM_8DEF_CURR	.09
FM_9MN_CURRDEF	DEFAULT9	SM_9DEF_CURR	.09

7.2.5. Decimal Symbols

JAM 5 will accomodate 3 decimal symbols which are used in different circumstances:

- System Decimal Symbol
- Local Decimal Symbol
- Field Decimal Symbol

The System Decimal Symbol is the one that library routines like `atof` and `sprintf` use. The Local Decimal Symbol is the one that is used when local customs are followed (dot in English; comma in French). The Field Decimal Symbol is the one specified for a given field if that field is not observing local conventions.

The System and Local Decimal Symbols are obtained from the operating system if the operating system supports such things (see the installation notes for JAM for your operating system). The Local Decimal Symbol may be specified in the message file (message `SM_DECIMAL`), in which case it overrides the operating system decimal symbol. Dot is the system decimal if no symbol is specified in the message file and if the operating system does not supply one.

The sections below describe the circumstances under which each of the different symbols is used.

7.2.6. Character Filters

The one time that JAM requires some knowledge of the meaning of the data is while enforcing the character filters on a field. The filters currently supported are digits only, numeric, alphabetic, alphanumeric, and yes/no and regular expression.

To validate the data JAM uses the standard C macros: `isdigit`, `isalpha`, etc. JAM 5 assumes that the operating system supplies these macros in a form suitable for international use. In absence of such operating system support, care should be taken when using these capabilities.

Special code is used to process numeric fields since C does not provide an "isnumeric" macro. If the field has a currency edit, JAM uses the Field Decimal Symbol to validate the numeric entry. If the field has no currency edit or the currency edit has no decimal symbol specified, JAM uses the Local Decimal Symbol.

Yes/no fields have always been internationalized in that the yes and no characters (`y` and `n` in English) are specified in the message file. Although some vendors will supply information about these characters, the proposed ANSI standard does not address the issue. Therefore, for reasons of portability, JAM will continue to use the message file for this data.

Upper and lower case fields will also behave properly provided that `toupper` and related functions are language dependent. The present code assumes that the return from `toupper` is appropriate for an upper case field. Therefore a lower case letter can appear in such a field if there is no upper case equivalent for that letter. (The German "double s" has no upper case equivalent.)

In processing regular expressions, JAM 5 uses the ASCII collating sequence for ranges of characters. Therefore, the expression

```
[a-z]*
```

will match only the English lower case letters. The European character a, for example, would not be matched by this expression.

7.2.7. Status And Error Messages

All messages produced by JAM 5 are stored in the message file so they may be easily localized. Each message is a complete phrase or sentence. Message components are never pieced together because doing so would make it difficult to translate to a language that has a sentence structure different from English.

7.2.8. Screens In The Utilities

These screens were memory resident in Release 4. For international customers they must be modifiable.

A linkable `jxform` is be provided, and the library containing the source for the screens is made available. A developer may translate the screens and relink the utilities. Similarly `modkey` is developer–linkable, and the source for its screens is provided. In this way the screens remain memory resident and no compromise of speed need be made.

Unfortunately this solution is not ideal if several users on the same machine wish to use different languages. To support this, the screens may be kept on disk. The current mechanism of `SMPATH` allows run–time selection of the set of screens to be used.

7.2.9. Screens In Application Programs

The same approach as discussed in the above section can be used for screens in application programs. Thus different language screens can be kept in separate directories and the user can specify which is to be used at run–time.

7.2.10. Menu Processing

`xsm_input` returns the first character of the selected entry. This, of course, is not language independent. JAM utilities have been modified to use the current field number rather than the return value. Because it cannot be assumed that all entries will have unique first letters, the `string` option is specified.

Application programs intended for an international market should not rely on the initial character of the menu selection. The field number containing the cursor is a better way of determining which selection the operator has made. However the field numbers may change if the screen is redesigned. Note that this is not a problem when the JAM Executive is used, since the JAM Executive uses relative field numbers to determine the control string to execute when a menu field is selected.

A new additional edit was instituted in JAM 4 that specifies the return code from a `return entry` (or `menu`) field. The screen creator specifies the return code (an integer) when designing the screen. If this edit exists, `xsm_input` uses that value as the return code to the calling program. If this edit does not exist, the usual return code is used.

7.2.11. `lstform`, `lstdd`, and `jammap`

These utilities list data about the screen in English. Since they are often used for documentation it is important that the text be translatable to other languages. Thus the textual material, headings, etc., have been moved to the message file.

7.2.12. Range Checks

Range checks for numeric data are presently correctly handled since they use `atof` (assuming that the "strip" routine works properly).

Alphabet data presents special problems. One of the major issues for internationalization is the collating sequence of a language. For dictionary or telephone book processing the problem is particularly troublesome. For example, upper and lower case letters compare equal. Also, in a telephone book, `St.` and `Saint` compare equal, hyphens are ignored, etc. In some languages even less demanding applications pose severe problems. For example, ligatures compare equally to pairs of letters. The placement of vowels with diacritical marks varies widely even among countries using the same language.

The proposed ANSI standard specifies a routine, `strcoll`, that can be used to expand the word into a format suitable for comparison by `strcmp`. These routines assume that the data supplied is a word in the local language. They will given unexpected results on non–language data.

JAM is not designed to process languages in a way that requires such niceties. It does sort names of fields and other objects, but that is done only to speed look–up. As long as the sort routine and the search routine use the same algorithm, things will work.

In JAM, range checks are often given on non–language data. For example a menu selection might have a range of a to d. In certain languages an umlaut would fall into that range if a language specific comparison was made. This effect would complicate screen design. Different screens would be needed for different countries, even if they used the same language.

For these reasons no changes have been made to the Release 4 method of range checking. `strcmp` and `memcmp` continue to be used. These compare the internal values of the characters, without regard to their meanings in the local language.

7.2.13.
Calculations Using `@SUM` and `@DATE`

These keywords have been retained even though they are language specific. Computations with dates assume the Gregorian calendar. No provison is made for other calendars.

7 2.14.
`xsm_dblval` and `xsm_dtofield`

These routines use `atof` and `sprintf` therefore correctly interpret the System Decimal Symbol (radix character).

7.2.15.
`xsm_is_yes` and `xsm_query_msg`

These routines use the characters in the message file for `y` and `n` and thus are already internationalized. They use `toupper` to recognize the upper case variations.

7.2.16.
Batch Utilities

All the utilities messages, including usage messages have been moved to the message file.

The mnemonics for logical keys (XMIT, EXIT, etc.) are not translated to other languages, nor the mnemonics used in the video file, so the internal processing of the utilities need not be modified.

Chapter 8. Writing Portable Applications

The following section describes 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.

8.1. TERMINAL DEPENDENCIES

JAM can run on display terminals of any size. On terminals 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 (21 if two–line soft key labels will be used).

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. Colors will not show up on monochrome terminals, e.g. Use of graphics character sets is particularly terminal dependent.

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. For simple choices among alternatives, menus 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 the setup file can specify terminal–dependent message files.

Chapter 9. Writing Efficient Applications

9.1. 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. However, the screens must first be converted to source language modules with `bin2pl1` or a related utility (see the Utilities Guide), then compiled and linked with the application program.

`xsm_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 Screen Editor can only operate on disk files, so that altering memory–resident screens during program development requires a tedious cycle of test – edit – reinsert with `bin2pl1` – recompile. The JAM library maintains an internal list of memory–resident screens that `xsm_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.

Call `xsm_formlist` to add a screen to JAM's memory–resident screen 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.

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.

9.2. MEMORY–RESIDENT CONFIGURATION FILES

Any or all of the three configuration files required by JAM can be made memory resident. First a PL/1 source file must be created from the binary version of the file, using the bin2pl1 utility; see the Utilities Guide. The source files created are not readily decipherable. A call is then made to either xsm_msgread, xsm_vinit, or xsm_ keyinit, depending on the type of configuration file being installed.

If a file is made memory–resident, the corresponding environment variable or SMVARS entry can be dispensed with.

9.3. 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 xsm_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.

9.4. 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 variable `I_NODISP` may also be used to decrease screen output. While this variable is set to 0 (via `xsm_iset`), calls into the JAM library will cause the internal screen image to be updated, but nothing will be written to the actual display; the display can be brought up to date by resetting `I_NODISP` to 1 and calling `xsm_rescreen`. With the implementation of delayed write this sort of trick is rarely necessary.

9.5. JPL VS. COMPILED LANGUAGES

JPL code execution goes through an extra layer of intrepretation that compiled code, such as PL/1, does not. In most cases, the total run time is too small to matter, but if a JPL function is long or loops many times and a delay is noted, it may pay to rewrite it in PL/1.

Chapter 10. Block Mode

The purpose of this document is to describe the block mode capabilities of JAM from the perspective of someone using the system and from the perspective of a developer that needs to write a block mode driver.

10.1. USING BLOCK MODE

10.1.1. General Overview

The purpose of the block mode interface is to allow JAM to be used with terminals, like the HP2392A and IBM 3270's, that operate in block mode. Such terminals, which are hereinafter referred to as block mode terminals, operate differently than their interactive or character mode counterparts in that they do not interact with the computer on every keystroke. Instead, a formatted screen is sent to the terminal and processed by the terminal locally. When a function key is pressed, data are transmitted to the computer and are available to the program which sent the formatted screen.

Block mode terminals typically have capabilities for defining protected and unprotected fields and sometimes allow a minimal set of character validations such as restricting a field to only allow digits. They do not provide JAM–like capabilities such as shifting, scrolling and provisions for post–field validation. It should therefore seem obvious that an application will behave slightly differently on a block mode terminal than on an interactive one. The goal of the block mode interface, however, is to minimize these differences and, to the greatest extent possible, allow applications to be created that can operate in either mode without the need for the programmer to consider the differences. This is in keeping with the JAM philosophy of creating terminal–independent applications.

10.1.2. Authoring

Certain JAM utilities, like `modkey`, the Screen Editor, and the Data Dictionary Editor only work in interactive mode. Thus, they can only be used with interactive terminals or those that can be switched programmatically between block and interactive mode.

`jxform` is the JAM authoring utility. It allows the user to navigate through the screens in an application and to invoke the Screen and Data Dictionary Editors when appropriate. When used with block mode–only terminals, `jxform` does not permit entry into the aforementioned utilities. When used with hybrid terminals (i.e. those that can switch between block and interactive mode programmatically), `jxform` forces interactive mode before entering the utilities.

10.1.3. Selecting Block Mode

JAM operates with three types of terminals: interactive–only, block mode–only, and hybrid. Block mode can be used with either of the latter two.

By default, JAM operates in interactive mode regardless of the terminal type. To operate in block mode requires a block terminal driver to be linked with the system. (Block terminal drivers are described in detail later.) This alone, however, will not initiate block mode; two additional things must be done.

First there must be a call to `xsm_blkinit`. This is generally done in the "main" routine of the application, `jmain.pl1`. If this call is absent, the application will be run in interactive mode. Also the additional code to support block mode will not be linked with the program. Thus programs not desiring block mode support are not penalized.

Second the correct block mode driver must be selected. This can be done in one of two ways.

If the application program author knows the correct driver he/she can install it by calling `xsm_uinstall`. This should be done before calling `xsm_blkinit`. Typically the program will install a "hard–coded" driver, but it could instead key off of SMTERM, or some other environment variable, to find the correct one. In this case the application will run in block mode, independent of the end user's preference.

The second method for selecting the driver leaves the job to the end user. If `xsm_blkinit` is called without previously installing a driver, the entry BLKDRIVER in the video file is examined. If it is absent, `xsm_blkinit` fails and the application remains in interactive mode. If it is present the name given there is used to find the correct driver. This is done by a table lookup in a source routine (`blkdrvr.c`) that must be linked with the application. Naturally all possible choices of the driver must also be linked with the program. In this case the end user can override the application programers desire to use block mode.

The design allows for three scenarios: the programmer can prohibit block mode (no call to `xsm_blkinit`), the programmer can force block mode (`xsm_install` followed by `xsm_blkinit`), or the programmer can permit block mode but allow the end user final say (`xsm_blkinit` only).

Note that the application never calls `sm_blkdrvr`. The source code to that routine is given to customers to enable them to extend the capabilities of the second method.

10.1.4. Differences Between Block Mode And Interactive Mode

Although every attempt has been made to preserve the look and feel of applications operating in block mode, the following differences between block mode and interactive mode should be noted.

Windows

Windows work much as they do in interactive mode. The only noticable difference is that the cursor is not be restricted to the active window as this is not possible in block mode. In keeping with the concepts of interactive mode, however, only the fields on the active window are unprotected.

Menus

In interactive mode, menus utilize a "bounce bar" to track the cursor. The bounce bar moves when cursor-positioning keys are pressed and when ascii data are typed. Since block mode terminals do not return these keys, another approach must be taken. We supply two options:

In option 1, menu fields in block mode are unprotected, making it easy for an operator to tab to them. To make a selection, the operator positions to the appropriate field and presses XMIT. Thus, selection is similar to interactive mode except there is no bounce bar and there is no provision for selecting by typing the first N characters of the menu choice.

If the operator inadvertently types over a menu field there are no adverse consequences as JAM will "remember" the contents and restore it at an appropriate time.

This approach works well since the same screens can be used for block and interactive mode operation. However, for those who do not wish to allow the operator to type over menu choice fields, option 2 may be chosen. With option 2, JAM creates an unprotected field to the left of each menu choice so the menu fields themselves can remain protected. The operator can tab to these new fields to make a selection, or type the first character of a menu field and press XMIT. The new fields to the left of the menu choices are created as long as there is room on the screen even if it means they would be placed in a border or a separate window. If there is no room on the screen because the menu field starts in position 1 or 2, the system reverts to option 1.

The above works well for traditional menus, but two–level (pull– down) menus pose a different problem in that the ONLY way to move horizontally in interactive mode is via the arrows (since TAB moves between the entries of the sub–menu). Thus, in block mode the following happens. When a pull–down menu is active, JAM unprotects all main menu fields except the one with which the pull–down is associated. Thus, the operator can either make a selection from the pull–down or tab to another main menu choice and press XMIT causing its sub–menu to be activated.

The two options for processing menus described above work equally well for pull–down menus.

Character Validation

The block mode interface takes advantage of the terminal's capabilities for character validation. However, for situations in which the specified validations go beyond what the terminal can handle, JAM will validate the character data during Screen Validation. The result will be something like this:

The operator enters alphabetic data in a digits–only field. When the XMIT key is pressed, all fields are validated in the normal fashion, left–to–right, top–to–bottom. Thus, the cursor will be positioned to the errant field and a message displayed.

Since programs do not rely on data being correct unless and until Screen Validation completes without error, this should pose no problem. The only consideration is that invalid character data can get into the screen buffer and LDB if the operator enters incorrect characters and then presses something like EXIT (this cannot happen in interactive mode because the invalid characters would not be allowed in the first place.

The only reason for mentioning this has to do with how punctuation characters in digits–only fields are handled. Let's say that a digits–only field got filled with slash ("/") characters and this, in turn, got transferred to the screen buffer and hence to the LDB. On a subsequent attempt to enter data into the field, an attempt to merge the slashes with the entered data would be made. But since the field has ALL slash characters, there would be no room for the digits.

Thus, to eliminate the possibility of "punctuation character creep", when reading data from a digits–only field, JAM first strips out all punctuation characters from the field and then merges in the punctuation characters from the screen buffer.

Field Validation

Clearly, fields are not validated when TAB and RETURN are pressed as in interactive mode. Thus, like character validations, field validations will be deferred until Screen Validation. This should not be a problem since, even in interactive mode, the operator can usually bypass field validation by using the arrow keys to move from field to field. Therefore, programs should not rely on the data until Screen Validation passes without error in either mode.

One type of field validation is worth noting. Consider a field with an attached function which does a database lookup and displays information in another field. In interactive mode, this would usually be executed when the field is completed, so the user would see the result. Since this is not really a validation, deferring it until Screen Validation would not help because the data would never be seen by the operator. Therefore, if this type of feature is contemplated in a block mode environment, the database lookup should be attached to a function key rather than as an attached function.

Screen Validation

Screen validation works the same in interactive and block mode. The cursor will be positioned to the first field in error and a message will be displayed to the operator.

Right Justified Fields

Unless the block mode terminal supports this feature directly, the cursor will always be positioned to the left side of right justified fields when the cursor enters them.

Field Entry Function, Automatic Help, Status Text, etc.

These are disabled in block mode since JAM does not know when fields are entered.

Currency Fields

Currency edits are usually applied to fields as they are exited. In block mode, since this is not possible, currency formatting is done during screen validation. Care should be taken with right justified currency formats since subsequent entry may be difficult for the reasons cited above in the section on right justified fields.

Shifting Fields

Normally fields shift when the left or right arrows are pressed with the cursor at the start or end of a shifting field or, in the case of unprotected fields, when the operator types off the edge of the field. Since arrows and data entry keys are not returned in block mode, this is not possible. To utilize shifting fields in block mode, use the logical keys: Shift Left and Shift Right. These shift the field by the shifting increment and work equally well in block and interactive mode.

An alternative is to use the Zoom feature if all shifting fields are limited to the width of the screen.

Scrolling Fields

This is similar to the situation with shifting fields. In block mode, one can define function keys as PAGE UP and PAGE DOWN, or use the Zoom feature.

Messages

Error messages are normally acknowledged by pressing the space bar, although the specific key used can vary depending on the setting of error message options. Also, options govern whether the key should be used as the next keystroke or discarded after the message is acknowledged. In block mode, ANY key that gets transmitted from the terminal will suffice to acknowledge messages, regardless of what key is defined for that purpose. Using or discarding the acknowledgement key apply equally to block mode and interactive mode.

With query messages, JAM normally expects a Y or N response. In block mode, JAM will create a field on the status line into which the Y or N response can be entered. This entry must be followed by the XMIT key for it to be accepted. On terminals that have a separate stauts line it is not possible to create such a field. In these cases, XMIT will be treated as a positive response; EXIT will be treated as a negative response.

Insert Mode

Insert mode will operate in whatever way the block mode terminal supports. However, since JAM never knows if insert mode is set or not in block mode, it will, for terminals in which this is a problem, reset insert mode before transmitting data to the terminal. This is so the new data will not be INSERTED into the terminal buffer, causing all other data to move around.

Non–Display Fields

If the block mode terminal supports this feature, it will be used.

System Calls

These operate as in interactive mode. However, before passing control to the OS, JAM sets the terminal to the mode (block or interactive) expected by the OS, and resets it upon return from the system call. The JAM routines `xsm_leave` and `xsm_return` do the same.

Zoom

With the exception of the limitations expressed in the sections on shifting and scrolling, Zoom works as in interactive mode.

Help and Item Selection

With the exception of the limitations expressed in the sections on shifting, scrolling, field entry and menu processing, these functions work as in interactive mode.

Groups

Radio buttons and check lists behave similar to menus as described above.

10.2. WRITING A BLOCK MODE DRIVER

10.2.1. Installation

There are two parts to the installation process. These were discussed in greater detail above.

First a block terminal driver must be installed. This driver performs the low level communication between JAM and the terminal. The PL/1 interface does not currently support writing your own block mode drivers.

Next the application program must initiate block mode by making the appropriate subroutine call. The application program can also switch to interactive mode by means of a call. The assumption is that the default is interactive mode, thus a call to set block mode is needed even if that is the normal mode of the operating system. The application program can also set some operating parameters by means of a subroutine call.

10.2.2. Application Program Support

JAM programs assume that the terminal is in interactive mode. Explicit calls are needed to switch from interactive to block and vice versa. To turn on block mode, the program should call `xsm_blkinit`. To turn off block mode (and turn on interactive mode) the program calls `xsm_blkreset`. The Screen Editor The key mapping utility (`modkey`) also requires interactive mode. The authoring utility (`jxform`) can be made to work in block mode, switching to interactive mode when the Screen Editor is invoked. This can be done by inserting the appropriate calls in `jxmain.pl1` (provided) and relinking `jxform`.

The routine `xsm_option` can be used to set some user-preference items.

Chapter 11. Library Function Overview

In this chapter, we summarize the JAM library functions and list them in categories. All JAM library function names begin with the prefix `xsm_`. However, in the Function Reference Chapter and in this chapter, the functions are listed without prefix for clarity.

In addition to stripping off the prefix in the listings that follow, groups of closely related variant functions are listed under a single root name. The functions `xsm_r_form`, `xsm_d_form`, and `xsm_l_form`, for example, are all grouped under the heading `form`. In a few cases, functions may be listed under a name that is not a portion of the the function name but is suggestive of the utility of the function. For example, the function `xsm_r_at_cur`, which displays a window at the cursor position, is listed under the root name `window`, along with `xsm_r_window` (which displays a window at a fixed location) and a number of other window display routines. The calling syntax of each function is found in the SYNOPSIS section of the function listing in the Function Reference Chapter.

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

- Initialization/Reset
- Screen and Viewport Control
- Keyboard and Display I/O
- Field/Array Data Access
- Field/Array Characteristic Access
- Group Access
- Local Data Block Access
- Cursor Control
- Message Display

- Scrolling and Shifting
- Mass Storage and Retrieval
- Validation
- Global Data and Changing JAM's Behavior
- Soft Keys and Keysets
- JAM Executive Control
- Block Mode Control
- Miscellaneous

The following sections summarize the functions that fall into these categories. Some listings are found in more than one category.

11.1. INITIALIZATION/RESET

The following library functions are called in order to initialize or reset certain aspects of the JAM runtime environment. Those that are necessary for the proper operation of JAM are called from within the supplied `main` routine source modules `jmain.pl1` and `jxmain.pl1`.

`cancel`	reset the display and exit
`dicname`	set data dictionary name
`ininames`	record names of initial data files for local data block
`initcrt`	initialize the display and JAM data structures
`keyinit`	initialize key translation table
`ldb_init`	initialize (or reinitialize) the local data block
`leave`	prepare to leave a JAM application temporarily
`msgread`	read message file into memory
`resetcrt`	reset the terminal to operating system default state
`return`	prepare for return to JAM application
`vinit`	initialize video translation tables

11.2. SCREEN AND VIEWPORT CONTROL

The following routines are used to control viewports, the display of screens, and the form and window stacks.

`close_window`	close current window
`form`	display a screen as a form
`hlp_by_name`	display help window
`issv`	determine if a screen in the saved list
`jclose`	close current window or form under JAM Executive control
`jform`	display a screen as a form under JAM control
`jwindow`	display a window at a given position under JAM control
`mwindow`	display a status message in a window
`shrink_to_fit`	remove trailing empty array elements and shrink screen
`sibling`	define the current window as being or not being a sibling window
`submenu_close`	close the current submenu
`svscreen`	register a list of screens on the save list
`unsvscreen`	remove screens from the save list
`viewport`	modify viewport size and offset
`wcount`	obtain number of currently open windows
`wdeselect`	restore the formerly active window
`window`	display a window at a given position
`winsize`	allow end–user to interactively move and resize a window
`wselect`	activate a window

11.3. DISPLAY TERMINAL I/O

The following routines provide the interface to JAM terminal I/O.

`bel`	beep!
`bkrect`	set background color of rectangle
`do_region`	rewrite part or all of a screen line
`flush`	flush delayed writes to the display
`getkey`	get logical value of the key hit
`input`	open the keyboard for data entry and menu selection
`keyfilter`	control keystroke record/playback filtering
`keyhit`	test whether a key has been typed ahead

`keylabel`	get the printable name of a logical key
`keyoption`	set cursor control key options
`m_flush`	flush the message line
`rescreen`	refresh the data displayed on the screen
`resize`	dynamically change the size of the display
`ungetkey`	push back a translated key on the input

11.4. FIELD/ARRAY DATA ACCESS

The following routines access the data in fields and arrays. Most routines in this section have a number of variants that perform the same task but reference the field to be accessed differently. In these cases, the calling syntax of the *major* variant is listed under the SYNOPSIS section of the listing in the Function Reference Chapter. All other variants are listed under the VARIANTS section.

Most field access routines have five variants, although some have fewer. The five possible variants are shown in the table below:

Variants of Functions That Access Fields

Prefix	*Example*	*Description*
`xsm_`	`xsm_intval(fieldnum);`	Access a field via field number.
`xsm_n_`	`xsm_n_intval(fieldname);`	Access a field (or an entire array) via field name. Access the LDB if there is no field on the screen.
`xsm_i_`	`xsm_i_intval(fieldname, occurrence);`	Access an occurrence via field name and occurrence number. Access the LDB if there is no field on the screen.
`xsm_o_`	`xsm_o_intval(fieldnum, occurrence);`	Access an occurrence via field number and occurrence number.
`xsm_e_`	`xsm_e_intval(fieldname, element);`	Access an element via field name and element number.

`amt_format`	write data to a field, applying currency editing
`calc`	execute a math edit style expression
`cl_unprot`	clear all unprotected fields
`clear_array`	clear all data in an array
`dblval`	get the value of a field as a real number
`dlength`	get the length of a field's contents
`doccur`	delete occurrences
`dtofield`	write a real number to a field
`fptr`	get the content of a field
`getfield`	copy the contents of a field
`gwrap`	get the contents of a wordwrap array
`intval`	get the integer value of a field
`ioccur`	insert blank occurrences into an array
`is_no`	test field for no
`is_yes`	test field for yes
`itofield`	write an integer value to a field
`lngval`	get the long integer value of a field
`ltofield`	place a long integer in a field
`null`	test if field is null
`putfield`	put a string into a field
`pwrap`	put text to a wordwrap field
`strip_amt_ptr`	strip amount editing characters from a string

11.5. FIELD/ARRAY ATTRIBUTE ACCESS

The following routines access information about fields and arrays. Like the routines in the previous section on field and array data access, each of these routines generally have five distinct variants. See the discussion in the introduction to the previous section for more information on variants of JAM library functions that access fields.

`base_fldno`	get the field number of the first element of an array
`bitop`	manipulate validation and data editing bits
`chg_attr`	change the display attribute of a field

`cl_all_mdts`	clear all MDT bits
`dlength`	get the length of a field's contents
`edit_ptr`	get special edit string
`finquire`	obtain information about a field
`fldno`	get the field number of an array element or occurrence
`ftog`	convert field references to group references
`ftype`	get the data type and precision of a field
`gtof`	convert a group name and index into a field number and occurrence
`length`	get the maximum length of a field
`max_occur`	get the maximum number of occurrences
`name`	obtain field name given field number
`num_occurs`	find the highest numbered occurrence containing data
`protect`	protect an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`size_of_array`	get the number of elements
`tst_all_mdts`	find first modified occurrence in the screen

11.6. GROUP ACCESS

The following routines access groups, that is, radio buttons and check lists. Groups are made up of fields that have attributes and data in them, but groups in and of themselves are implemented as phantom fields which take up no screen real estate. The value of a group indicates the set of selected consituent fields, although it is not recommended that that value ever be accessed or modified directly with any of the field access routines discussed in the preceding sections.

The routines that follow are those that are recommended for accessing groups:

`deselect`	deselect a checklist occurrence
`ftog`	convert field references to group references
`gp_inquire`	obtain information about a group
`gtof`	convert a group name and index into a field number and occurrence

`isselected`	determine whether a radio button or checklist occurrence has been selected
`select`	select a checklist or radio button occurrence

11.7. LOCAL DATA BLOCK ACCESS

The following routines access the Local Data Block, or LDB. Note that any of the field data access routines that reference fields by name or name and occurrence number (eg `xsm_n` and `xsm_i_` variants) will access the LDB if the named field does not exist on the active screen.

`allget`	load screen from the LDB
`dicname`	set data dictionary name
`dd_able`	turn LDB write–through on or off
`ininames`	record names of initial data files for local data block
`lclear`	erase LDB entries of one scope
`ldb_init`	initialize (or reinitialize) the local data block
`lreset`	reinitialize LDB entries of one scope
`lstore`	copy everything from screen to LDB

11.8. CURSOR CONTROL

The following routines control the positioning and display of the cursor on the active screen.

`ascroll`	scroll to a given occurrence
`backtab`	backtab to the start of the last unprotected field
`c_off`	turn the cursor off
`c_on`	turn the cursor on
`c_vis`	turn cursor position display on or off
`disp_off`	get displacement of cursor from start of field
`getcurno`	get current field number
`gofield`	move the cursor into a field

`home`	home the cursor
`last`	position the cursor in the last field
`nl`	position cursor to the first unprotected field beyond the current line
`occur_no`	get the current occurrence number
`off_gofield`	move the cursor into a field, offset from the left
`rscroll`	scroll an array
`sh_off`	determine the cursor location relative to the start of a shifting field
`tab`	move the cursor to the next unprotected field

11.9. MESSAGE DISPLAY

The following routines are intended for the access and display of runtime application messages.

`d_msg_line`	display a message on the status line
`emsg`	display an error message and reset the message line, without turning on the cursor
`err_reset`	display an error message and reset the status line
`m_flush`	flush the message line
`msg`	display a message at a given column on the status line
`msg_get`	find a message given its number
`msgfind`	find a message given its number
`msgread`	read message file into memory
`mwindow`	display a status message in a window
`query_msg`	display a question, and return a yes or no answer
`qui_msg`	display a message preceded by a constant tag, and reset the message line
`quiet_err`	display error message preceded by a constant tag, and reset the status line
`setbkstat`	set background text for status line
`setstatus`	turn alternating background status message on or off

11.10. SCROLLING AND SHIFTING

The following routines provide access to shifting and scrolling fields and arrays.

`achg`	change the display attribute of an occurrence within a scrolling array
`ascroll`	scroll to a given occurrence
`doccur`	delete occurrences
`ioccur`	insert blank occurrences into an array
`max_occur`	get the maximum number of occurrences
`num_occurs`	find the highest numbered occurrence containing data
`oshift`	shift a field by a given amount
`rscroll`	scroll an array
`sc_max`	alter the maximum number of items allowed in a scrollable array
`sh_off`	determine the cursor location relative to the start of a shifting field
`t_scroll`	test whether an array can scroll
`t_shift`	test whether field can shift
`tst_all_mdts`	find first modified occurrence

11.11. MASS STORAGE AND RETRIEVAL

The following routines move data to or from sets of fields in the screen or LDB.

`rd_part`	read part of a data structure to the current screen
`rdstruct`	read data from a structure to the screen
`restore_data`	restore previously saved data to the screen
`rrecord`	read data from a structure to a data dictionary record
`wrecord`	write data from a data dictionary record to a structure
`wrt_part`	write part of the screen to a structure
`wrtstruct`	write data from the screen to a structure

11.12. VALIDATION

The following routines provide an application interface to the field and group validation processes.

`bitop`	manipulate validation and data editing bits
`ckdigit`	validate check digit
`fval`	force field validation
`gval`	force group validation
`novalbit`	forcibly invalidate a field
`s_val`	validate the current screen

11.13. GLOBAL DATA AND CHANGING JAM'S BEHAVIOR

The following routines grant access to global data and provide a way to manipulate certain aspects of JAM and Screen Manager behavior.

`async`	install an asynchronous function
`dd_able`	turn LDB write–through on or off
`finquire`	obtain information about a field
`gp_inquire`	obtain information about a group
`inquire`	obtain value of a global integer variable
`isabort`	test and set the abort control flag
`iset`	change value of integer global variable
`keyfilter`	control keystroke record/playback filtering
`keyoption`	set cursor control key options
`li_func`	install an application hook function
`msgread`	read message file into memory
`option`	set a Screen Manager option
`pinquire`	obtain value of a global strings
`pset`	Modify value of global strings

`resize`	dynamically change the size of the display
`uinstall`	install an application function

11.14. SOFT KEYS AND KEYSETS

The following routines provide an application interface to JAM's soft key support.

`c_keyset`	close a keyset
`keyset`	open a keyset
`kscscope`	query current keyset scope
`ksinq`	inquire about key set information
`ksoff`	turn off key labels
`kson`	turn on key labels
`skinq`	obtain soft key information by position
`skmark`	mark or unmark a softkey label by position
`skset`	set characteristics of a soft key by position
`skvinq`	obtain soft key information by value
`skvmark`	mark a soft key by value
`skvset`	set characteristics of a soft key by value

11.15. JAM EXECUTIVE CONTROL

The following routines, available only to applications using the JAM Executive, provide JAM Executive services.

`getjctrl`	get control string associated with a key
`jclose`	close current window or form under JAM Executive control
`jform`	display a screen as a form under JAM control
`jtop`	start the JAM Executive
`jwindow`	display a window at a given position under JAM control
`putjctrl`	associate a control string with a key

11.16. BLOCK MODE CONTROL

The following routines are used in applications requiring block mode support.

`blkdrvr`	install block mode driver
`blkinit`	initialize (and turn on) block mode terminal
`blkreset`	reset (and turn off) block mode terminal

11.17. MISCELLANEOUS

`fi_path`	return the full path name of a file
`formlist`	update list of memory-resident files
`jplcall`	execute a JPL procedure
`jplload`	execute the JPL load command
`jplpublic`	execute the JPL public command
`jplunload`	execute the JPL unload command
`l_close`	close a library
`l_open`	open a library
`rmformlist`	empty the memory-resident form list
`sdtime`	get formatted system date and time
`udtime`	format user-supplied date and time

Chapter 12. Function Reference

All JAM function names begin with the prefix `xsm_`. In the Function Reference Chapter functions are listed without the prefix and, in a few cases, under a name that is not a portion of the function name — but that is suggestive of the utility of the function. For example, the function `xsm_r_at_cur`, which displays a window at a specified position, is found under the listing name `window`, along with the function `xsm_r_window`. In these cases, the calling syntax of each function is listed under the SYNOPSIS section of the listing.

For each entry, you will find several sections:

- A synopsis similar to a PL/1 function declaration, giving the types of the arguments and return value.
- A description of the function's arguments, prerequisites, results, and side-effects.
- The function's return values, if any, and their meanings.
- A list of variants.
- A list of functions that perform related tasks.
- An example illustrating the function's use.

A routine that calls JAM functions should include the file `smdefs.incl.pl1`. If another file should be included, then it is referenced in the synopsis section.

To view functions by category, refer to the Library Function Overview (chapter 11.) To view a complete list of functions alphabetically by the actual function name (including the `xsm_` prefix), see the Library Function Index (chapter 13.).

achg

change the display attribute of an occurrence within a scrolling array

SYNOPSIS

```
declare field_number      fixed binary(31);
declare occurrence        fixed binary(31);
declare display_attribute fixed binary(31);
declare status            fixed binary(31);
status = xsm_o_achg(field_number, occurrence,
        display_attribute);
```

DESCRIPTION

NOTE: This function has only two variants, `xsm_o_achg` and `xsm_i_achg`. There is NO `xsm_achg`.

This function changes the display attribute of an occurrence within a scrollable array. If the occurrence is onscreen, the attribute with which the occurrence is currently displayed is changed as well. When the occurrence is scrolled to another position within the array the new attribute moves with the occurrence. Use `xsm_chg_attr` if you want all of the occurrences within the array to scroll through an attribute so that their appearance is determined by their onscreen positions.

Possible values for the argument `display_attribute` are defined in the header file `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	

Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN
CYAN	B_CYAN
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

If `display_attribute` is zero, the occurrence's display attribute is removed, leaving it with the field display attribute. Then, if that occurrence is onscreen, it is displayed with the attribute attached to its field.

This function will not work on an array that is not scrollable. Use `xsm_chg_attr` to change the display attribute of an individual field.

RETURNS

–1 if the field isn't found or isn't scrollable, or if `occurrence` is invalid. 0 otherwise.

VARIANTS

```
status = xsm_i_achg(field_name,. occurrence, display_attribute);
```

RELATED FUNCTIONS

```
·status = xsm_chg_attr(field_number, display_attribute);
```

allget
load screen from the LDB

SYNOPSIS

```
declare respect_flag      fixed binary(31);
call xsm_allget(respect_flag);
```

DESCRIPTION

This function copies data from the local data block to fields on the current screen with matching names.

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 called automatically by the JAM screen-display logic, unless LDB processing has been turned off using `xsm_dd_able`. Application code should not normally need to call it.

RELATED FUNCTIONS

```
call xsm_dd_able(flag);
status = xsm_lstore();
```

amt_format
write data to a field, applying currency editing

SYNOPSIS

```
declare field_number       fixed binary(31);
declare buffer             char(256) varying;
declare status             fixed binary(31);
status = xsm_amt_format(field_number, buffer);
```

DESCRIPTION

If the specified field has a currency 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, `xsm_putfield` is called to write the edited string to the specified field.

If the field has no currency edit, `xsm_putfield` is called with the unedited string.

RETURNS

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

VARIANTS

```
status =  xsm_e_amt_format(field_name, element, buffer);
status = xsm_i_amt_format(field_name, occurrence, buffer);
status = xsm_n_amt_format(field_name, buffer);
status = xsm_o_amt_format(field_number, occurrence, buffer);
```

RELATED FUNCTIONS

```
status = xsm_dtofield(field_number, value, format);
outbuf = xsm_strip_amt_ptr(field_number, inbuf, );
```

ascroll

scroll to a given occurrence

SYNOPSIS

```
declare field_number      fixed binary(31);
declare occurrence        fixed binary(31);
declare status            fixed binary(31);
status = xsm_ascroll(field_number, occurrence);
```

DESCRIPTION

This function scrolls the designated field so that the indicated `occurrence` appears there. Synchronized arrays will scroll along with the target array.

The field need not be the first element of a scrolling array. You can use this function, for instance, to place the nineteenth occurrence 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 and `occurrence` is 1 a call to `xsm_ascroll` will fail on a non-circular scrolling array but succeed if scrolling is circular.

RETURNS

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

VARIANTS

```
status = xsm_n_ascroll(field_name, occurrence);
```

RELATED FUNCTIONS

```
lines = xsm_rscroll(field_number, req_scroll);
status = xsm_t_scroll(field_number);
```

async
install an asynchronous function

SYNOPSIS

```
declare func              entry variable;
declare timeout           fixed binary(31);
call xsm_async(func, timeout);
```

DESCRIPTION

This routine installs a a function that will be called regularly during keyboard processing (ie. - `xsm_input`). The first parameter is the address of the function. Use the operating system subroutine `s$find_entry` to find the entry point. The second parameter is the timeout, in tenths of a second, between subsequent function calls.

The asynchronous function is called only when the keyboard is being read, and only if a keystroke does not arrive within the specified timeout. The authoring utility, `jxform`, uses an asynchronous function to update its cursor position display. An asynchronous function might also be used to implement a real–time clock display.

RELATED FUNCTIONS

```
status = xsm_uinstall( usage, func, func_name);
```

backtab

backtab to the start of the last unprotected field

SYNOPSIS

```
call xsm_backtab();
```

DESCRIPTION

When the cursor is in a field unprotected from tabbing into, but not in the first enterable position, it is moved to the first enterable position of that field. However, if the cursor is in a field with a previous–field edit and one of the fields specified by the edit is unprotected from tabbing, the cursor is moved to the first enterable position of that field. Otherwise, the cursor is moved to the first enterable position of the tab–unprotected field with the next lowest field number. If the cursor is in the first position of the first unprotected field on the screen, or before the first unprotected field on the screen, it wraps backward into the last unprotected field. When there are no unprotected fields, the cursor doesn't move.

If the destination field is shiftable, it is reset according to its justification. The first enterable position depends on the justification of the field and, in fields with embedded punctuation, on the presence of punctuation.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is called when the JAM logical key BACK is struck.

RELATED FUNCTIONS

```
field_number = xsm_home();
call xsm_last();
call xsm_nl();
call xsm_tab();
```

base_fldno

get the field number of the first element of an array

SYNOPSIS

```
declare field_number      fixed binary(31);
declare base_number       fixed binary(31);
base_number = xsm_base_fldno(field_number);
```

DESCRIPTION

A base field number is the field number of the first element of an array. Use `xsm_base_fldno` to obtain the base field number of an array.

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.

bel

beep!

SYNOPSIS

```
  call xsm_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, `xsm_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.

bitop

manipulate validation and data editing bits

SYNOPSIS

```
%include 'smbitops.incl.pl1';

declare field_number        fixed binary(31);
declare action              fixed binary(31);
declare bit                 fixed binary(31);
declare status              fixed binary(31);
status = xsm_bitop(field_number, action, 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, which are defined in `smbitops.incl.pl1`:

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	*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_ADDLEDS	N_EPROTECT	N_TPROTECT
N_CPROTECT	N_VPROTECT	N_ALLPROTECT	N_SELECTED	

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 to individual offscreen and onscreen occurrences. The protection operations can apply to an array as a whole, including offscreen occurrences (see `xsm_aprotect`). All other bit operations are attached to fixed onscreen positions.

The variants `xsm_e_bitop` and `xsm_n_bitop` can take a group name as an argument. The function will then affect the group bits.

This function has two additional variants, `xsm_a_bitop` and `xsm_t_bitop`, which perform the requested bit operation on all elements of an array. Their synopsis appear below. If you include `BIT_TST`, these variants return 1 only if `bit` is set for *every* element of the array. The variants `xsm_i_bitop` and `xsm_o_bitop` are restricted to `N_VALIDED` and `N_MDT`.

RETURNS

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

VARIANTS

```
status = xsm_a_bitop(array_name, action, bit);
status = xsm_e_bitop(array_name, element, action, bit);
status = xsm_i_bitop(array_name, occurrence, action, bit);
status = xsm_n_bitop(name, action, bit);
status = xsm_o_bitop(field_number, occurrence, action, bit);
status = xsm_t_bitop(array_number, action, bit);
```

bkrect
set background color of rectangle

SYNOPSIS

```
declare start_line         fixed binary(31);
declare start_column       fixed binary(31);
declare num_of_lines       fixed binary(31);
declare number_of_columns  fixed binary(31);
declare background_colors  fixed binary(31);
declare status             fixed binary(31);
status = xsm_bkrect(start_line, start_column, num_of_lines,
         number_of_columns, background_colors);
```

DESCRIPTION

This function changes the background color of a rectangular area of the current screen. Any fields or elements that begin within the rectangular area will have their background attributes changed to the specified attribute. This means that if there are any fields or elements that are not entirely contained within the rectangular area, a ragged edge will result. Display text that falls with in the rectangular area will have its background attribute set.

The arguments `start_line` and `start_column` can have any value from 1 through the number of lines (or columns) on the screen.

The background color must be one of the mnemonics defined in `smdefs.incl.pl1` (B_BLACK, B_BLUE, etc.). You can highlight the background color by oring the background color attribute with B_HILIGHT.

RETURNS

–1 if the starting line or column was invalid.
1 if the starting line and column were valid, but the rectangle had to be truncated to fit.
0 if no error.

blkinit

initialize (and turn on) block mode terminal

SYNOPSIS

```
declare return_value        fixed binary(31);
return_value = xsm_blkinit();
```

DESCRIPTION

This routine must be called by the application program to initiate block mode terminal action. A block mode terminal driver must have been previously installed.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored. If the terminal cannot be put into block mode it will still work (possibly better) in interactive mode.

If the driver signifies that all is OK, the global variable `sm_blkcontrol` is set to point to the local block terminal control handler. All Screen Manager calls for block mode support are made through this control routine.

On the first call to the present routine the driver is called with BLK_INIT to perform any required initialization.

On subsequent calls BLK_BLOCK is called instead of BLK_INIT.

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
return_value = xsm_blkreset();
```

blkreset

reset (and turn off) block mode terminal

SYNOPSIS

```
declare return_value       fixed binary(31);
return_value = xsm_blkreset();
```

DESCRIPTION

This routine must be called by the application program to reset block mode terminal action. A block mode terminal driver must have been previously installed.

This routine checks that a block mode terminal driver is installed. If a driver is found, it is called. The driver should return 0 if all is successful.

Generally the return code can be ignored as the terminal is often already in interactive mode. The exception is on those systems that are normally block mode. Many JAM programs rely on the fact that the terminal can be put into interactive mode.

Note that the driver is called with BLK_CHAR, not with BLK_RESET. The only time the driver is called for a full reset is when JAM is about to go to the operating system – either exiting or performing a "shell escape".

RETURNS

return value from driver if one exists.
–1 otherwise.

RELATED FUNCTIONS

```
return_value = xsm_blkinit();
```

c_keyset
close a keyset

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope              fixed binary(31);
declare status             fixed binary(31);
status = xsm_c_keyset(scope);
```

DESCRIPTION

This function closes the keyset of the given scope. It frees all memory associated with the keyset and marks that scope as free. If the keyset was currently displayed, the keyset labels are changed to reflect the new keyset.

See the keyset chapter of the Author's Guide for a detailed explanation of keyset scopes.

Scope Mnemonic from `smsoftk.incl.pl1`	*Description*
KS_APPLIC	Application scope.
KS_FORM	Form or window scope.
KS_SYSTEM	`jxform` system key sets.

Use `xsm_d_keyset` and `xsm_r_keyset` to open keysets.

RETURNS

 0 if there is no error
–2 if there is no keyset currently at that scope
–3 if the scope is out of range

RELATED FUNCTIONS

```
status = xsm_r_keyset(name, scope);
status = xsm_d_keyset(address, scope);
```

c_off
turn the cursor off

SYNOPSIS

```
call xsm_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, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

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

Use xsm_c_on to turn the cursor on.

RELATED FUNCTIONS

```
call xsm_c_on();
```

c_on

turn the cursor on

SYNOPSIS

```
call xsm_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, the cursor is off.
- While Screen Manager functions are writing to the display the cursor is off.
- Within certain error message display functions the cursor is on.

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

Use `xsm_c_off` to turn the cursor off.

RELATED FUNCTIONS

```
call xsm_c_off();
```

c_vis
turn cursor position display on or off

SYNOPSIS

```
declare display           fixed binary(31);
call xsm_c_vis(display);
```

DESCRIPTION

Assigning a non–zero value to `display` displays subsequent status line messages with the cursor's position display. This includes background status messages. Messages that would overlap the cursor position display are truncated.

Setting `display` to zero will cause subsequent status line messages to be displayed without the cursor's position display.

This function will have no effect if the `CURPOS` entry in the video file is not defined. In that case the cursor position display will never 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 override it.

calc

execute a math edit style expression

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
declare expression         char(256) varying;
declare status             fixed binary(31);
status = xsm_calc(field_number, occurrence, expression);
```

DESCRIPTION

Use `xsm_calc` to execute a math edit style expression. With this function you can perform mathematical operations that use the contents of one or more fields and then insert the result into a field.

The third parameter `expression` is a math edit style expression. See the JAM Author's Guide for a complete description on how to create the expression.

The first two parameters, `field_number` and `occurrence` identify the field and occurrence with which the calculation is associated. Normally you will not need to use them and should set them both to 0.

If you want to use relative references to fields in your expression, use the arguments `field_number` and `occurrence` to specify the field to which they should be relative.

If in the event of a math error you want the cursor to move a specific field, specify that field with `field_number`. In addition, if the desired field is an occurrence within an array, specifying the `occurrence` will cause the referenced array to scroll to `field_number`.

RETURNS

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

cancel

reset the display and exit

SYNOPSIS

```
declare arg             fixed binary(31);
call xsm_cancel(arg);
```

DESCRIPTION

This function is installed by `xsm_initcrt` to be executed if a keyboard interrupt occurs. It calls `xsm_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 `xsm_resetcrt`, you may find your terminal in an odd state; `xsm_cancel` can prevent that.

The argument `arg` is a dummy argument. It should have the value zero.

chg_attr

change the display attribute of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare display_attribute  fixed binary(31);
declare status             fixed binary(31);
status = xsm_chg_attr(field_number, display_attribute);
```

DESCRIPTION

Use this function to change the display attribute of an individual field or an element within an array. To change an occurrence attribute so that the attribute moves with the occurrence use `xsm_o_achg`.

If the field is part of a scrolling array, then each occurrence may also have a display attribute that overrides the field display attribute when the occurrence arrives onto the screen.

Possible values for `display_attribute` are defined in `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	
Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN

Foreground Colors	*Background Colors*
CYAN	B_CYAN
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

NOTE: The variant `xsm_o_chg_attr` does not take the usual arguments. The second argument is an element rather than an occurrence.

RETURNS

–1 if the field is not found
0 otherwise.

VARIANTS

```
status = xsm_e_chg_attr(field_name, element,
         display_attribute);
status = xsm_n_chg_attr(field_name, display_attribute);
status = xsm_o_chg_attr(field_number, element,
         display_attribute);
```

RELATED FUNCTIONS

```
status = xsm_o_achg(field_number, occurrence,
         display_attribute);
```

ckdigit
validate check digit

SYNOPSIS

```
declare field_number       fixed binary(31);
declare field_data         char(256) varying;
declare occurrence         fixed binary(31);
declare modulus            fixed binary(31);
declare minimum_digits     fixed binary(31);
declare status             fixed binary(31);
status = xsm_ckdigit(field_number, field_data, occurrence,
         modulus, minimum_digits);
```

DESCRIPTION

This function is called by field validation. It verifies that `field_data` contains the required minimum number of digits terminated by the proper check digit. If not, it posts an error message before returning. It can also be used to check any character string or field. If `field_data` is null, the string to check is obtained from the `field_number` and `occurrence` and an error message is displayed if the 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.

A fuller description of `sm_ckdigit` is included with the source code, which is distributed with JAM.

Note that this function can be replaced by a user-installed check digit function which field validation will call instead. See the chapter on installing functions.

RETURNS

0 If the field contents are available and valid.
–1 If the field contents do not contain the minimum number of digits or the proper check digit.
–2 If the length of `field_data` is zero and the field or occurrence cannot be found

cl_all_mdts

clear all MDT bits

SYNOPSIS

```
call xsm_cl_all_mdts();
```

DESCRIPTION

Clears the MDT (modified data tag) of every occurrence, 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 `xsm_putfield`, since the screen was first displayed (i.e., after the screen entry function returns).

RELATED FUNCTIONS

```
field_number = xsm_tst_all_mdts(occurrence);
```

cl_unprot

clear all unprotected fields

SYNOPSIS

```
call xsm_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. Fields with the null edit are reset to their null indicator values.

This function is normally bound to the CLEAR ALL key.

RELATED FUNCTIONS

```
status = xsm_aprotect(field_number, mask);
```

clear_array

clear all data in an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_clear_array(field_number);

status = xsm_1clear_array(field_number);
```

DESCRIPTION

Both functions clear all data from the array containing the field specified by field_number. The value returned by `xsm_num_occurs` is changed to zero. The array is cleared even if it is protected from clearing (CPROTECT).

`xsm_clear_array` also clears arrays synchronized with the specified array, except for synchronized arrays that are protected from clearing.

`xsm_1clear_array` only clears the specified array.

RETURNS

−1 if the field does not exist;
0 otherwise.

VARIANTS

```
status = xsm_n_clear_array(field_name);
status = xsm_n_1clear_array(field_name);
```

RELATED FUNCTIONS

```
status = xsm_aprotect(field_number, mask);
status = xsm_protect(field_number);
```

close_window
close current window

SYNOPSIS

```
declare                     fixed binary(31);
status = xsm_close_window();
```

DESCRIPTION

xsm_close_window is used to close a window opened by xsm_r_window (or variant), xsm_r_at_cur (or variant), or xsm_mwindow.

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 xsm_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.

When using the JAM Executive, use xsm_jclose to close a form. xsm_jclose will call xsm_jform to pop the form stack and open the new top form on the stack. In the case of a window, xsm_jclose will call xsm_close_window to close the window.

RETURNS

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

RELATED FUNCTIONS

```
status = xsm_r_window(screen_name, start_line, start_column);
return_value = xsm_wselect(window_number);
```

d_msg_line

display a message on the status line

SYNOPSIS

```
declare message           char(256) varying;
declare display_attribute fixed binary(31);
call xsm_d_msg_line(message, display_attribute);
```

DESCRIPTION

The message in `message` is displayed on the status line, with an initial display attribute of `display_attribute`. If the cursor position display has been turned on (see `xsm_c_vis`), the end of the status line will contain the cursor's current row and column. Messages displayed with `xsm_d_msg_line` override both background and field status text.

Messages posted with `xsm_d_msg_line` are displayed until the status line is cleared by `xsm_d_msg_line`. They will persist from screen to screen until cleared. Clearing is accomplished by passing `xsm_d_msg_line` an empty string for `message` and a 0 for `display_attribute`. Once cleared, any currently overidden message will resume. The function `xsm_d_msg_line` will itself be overridden by `xsm_err_reset` and related functions, or by the ready/wait message enabled by `xsm_setstatus`.

Possible values for `display_attribute` are defined in `smdefs.incl.pl1`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper-case. Note that, if a message containing percent escapes is displayed before `xsm_initcrt` is called, the percent escapes will show up in the message.

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 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 zeros. If the following character is not a legal hex digit, then leading zeros 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.incl.pl1`; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.

If the message begins with a `%B`, JAM will beep the terminal (using `xsm_bel`) before issuing the message.

RELATED FUNCTIONS

```
call xsm_err_reset(message);
call xsm_msg(column, disp_length, text);
status = xsm_mwindow(text, line, column);
```

dblval

get the value of a field as a real number

SYNOPSIS

```
declare field_number      fixed binary(31);
declare value             float binary(53);
value = xsm_dblval(field_number);
```

DESCRIPTION

This function returns the contents of `field_number` as a real number. It calls `xsm_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

```
value = xsm_e_dblval(field_name, element);
value = xsm_i_dblval(field_name, occurrence);
value = xsm_n_dblval(field_name);
value = xsm_o_dblval(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_dtofield(field_number, value, format);
outbuf = xsm_strip_amt_ptr(field_number, inbuf, );
```

dd_able
turn LDB write–through on or off

SYNOPSIS

```
declare flag              fixed binary(31);
call xsm_dd_able(flag);
```

DESCRIPTION

During normal JAM processing, named fields in the screen and local data block are kept in sync. When a screen is displayed (and after the screen entry function completes), values are copied in from the LDB; when control passes from the screen (before the screen entry function is executed), values are copied back to the LDB. Normally, when application code reads or writes a value to or from a named field/LDB entry JAM treats the name as a field name unless no such field exists, in which case JAM treats the name as an LDB entry name. During screen entry and exit processing, this logic is reversed in order to preserve the illusion that screen and LDB entries that share the same name also share the same data.

`xsm_dd_able` turns this feature off if `flag` is "0" and on if it is "1". The feature is on by default. When it is off, the LDB is never accessed.

deselect

deselect a checklist occurrence

SYNOPSIS

```
declare group_name         char(256) varying;
declare group_occurrence   fixed binary(31);
declare status             fixed binary(31);
status = xsm_deselect(group_name, group_occurrence);
```

DESCRIPTION

This function allows you to deselect a specific occurrence within a checklist. The group name and occurrence number is used to reference the desired selection. See the Author's Guide for a more detailed discussion of groups.

Use `xsm_select` to select a group occurrence and `xsm_isselected` to check whether or not a particular group occurrence is currently selected.

NOTE: You can not deselect a radio button occurrence. Using `xsm_select` on a radio button occurrence will automatically deselect the current selection.

RETURNS

–1	arguments do not reference a checklist occurrence.
0	occurrence not previously selected.
1	occurrence previously selected.

RELATED FUNCTIONS

```
status = xsm_isselected(group_name, group_occurrence);
status = xsm_select(group_name, group_occurrence);
```

dicname

set data dictionary name

SYNOPSIS

```
declare dic_name          char(256) varying;
declare status            fixed binary(31);
status = xsm_dicname(dic_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 `xsm_ldb_init` is called to initialize the local data block from the data dictionary. The argument `dic_name` is a character string giving the file name; 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.

Use the function `xsm_pinquire` to find the name of the data dictionary in use.

RETURNS

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

RELATED FUNCTIONS

```
buffer = xsm_pinquire(which);
```

disp_off

get displacement of cursor from start of field

SYNOPSIS

```
declare offset          fixed binary(31);
offset = xsm_disp_off();
```

DESCRIPTION

Returns the difference between the first column of the current field and the current cursor location. This function ignores offscreen data; use `xsm_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.

RELATED FUNCTIONS

```
call field_number = xsm_getcurno();
call offset = xsm_sh_off();
```

dlength
get the length of a field's contents

SYNOPSIS

```
declare field_number       fixed binary(31);
declare data_length        fixed binary(31);
data_length = xsm_dlength(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 `xsm_getfield`). It does include data that have been shifted offscreen.

RETURNS

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

VARIANTS

```
data_length = xsm_e_dlength(field_name, element);
data_length = xsm_i_dlength(field_name, occurrence);
data_length = xsm_n_dlength(field_name);
data_length = xsm_o_dlength(field_number, occurrence);
```

RELATED FUNCTIONS

```
field_length = xsm_length(field_number);
```

do_region
rewrite part or all of a screen line

SYNOPSIS

```
declare line              fixed binary(31);
declare column            fixed binary(31);
declare length            fixed binary(31);
declare display_attribute fixed binary(31);
declare text              char(256) varying;
call xsm_do_region(line, column, length, display_attribute,
         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 screen region is redrawn with whatever `display_attribute` has been assigned. If `text` is shorter than `length`, it is padded out with blanks. In either case, the display attribute of the whole area is changed to `display_attribute`.

Possible values for `display_attribute` are defined in `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	

Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN
CYAN	B_CYAN
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

doccur

delete occurrences

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
declare count              fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_o_doccur(field_number, occurrence, count);
```

DESCRIPTION

NOTE: This function only exists in the o_ and i_ variations. There is NO xsm_doccur since this function only applies to arrays.

This function deletes the data in count occurrences beginning with the specified occurrence. If the array is scrollable, then it deallocates count occurrences. The data in occurrences following the last deleted occurrence are moved up in the array so that there are no gaps. Fewer than count occurrences will be deleted if the number of remaining allocated occurrences, starting with the referenced occurrence, is less than count.

If count is negative, occurrences are inserted instead, subject to limitations explained at xsm_ioccur. The function xsm_ioccur is normally used to add blank occurrences.

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.

Any clearing–unprotected synchronized arrays will have the same operations performed on them as the referenced array.

This function is normally bound to the DELETE LINE key.

RETURNS

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

VARIANTS

```
return_value = xsm_i_doccur(field_name, occurrence, count);
```

dtofield

write a real number to a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare value             float binary(53);
declare format            char(256) varying;
declare status            fixed binary(31);
status = xsm_dtofield(field_number, value, format);
```

DESCRIPTION

The real number `value` is converted to human–readable form, according to `format`, and moved into `field_number` via a call to `xsm_amt_format`. If the `format` string is empty, 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 number of decimal places may be forced to be an arbitrary number n, via rounding, by using the format string `%.nf`". The format string `%t.nf`" may be used to truncate instead of to round.

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

```
status = xsm_e_dtofield(field_name, element, value, format);
status = xsm_i_dtofield(field_name, occurrence, value, format);
status = xsm_n_dtofield(field_name, value, format);
status = xsm_o_dtofield(field_number, occurrence, value,
         format);
```

RELATED FUNCTIONS

```
status = xsm_amt_format(field_number, buffer);
value = xsm_dblval(field_number);
```

e_
variants that take a field name and element number

SYNOPSIS

```
declare field_name        char(256) varying;
declare element           fixed binary(31);
call xsm_e_...(field_name, element, ...);
```

DESCRIPTION

The e_ variant functions access one element of an array by field name and element number. For a description of any particular function, look under the related function without e_ in its name. For example, xsm_e_amt_format is described under xsm_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 because an element number is ambiguous when referring to the LDB.

edit_ptr

get special edit string

SYNOPSIS

```
declare buffer          char(256) varying;
declare field_number    fixed binary(31);
declare edit_type       fixed binary(31);
buffer = xsm_edit_ptr(field_number, edit_type);
```

DESCRIPTION

This function searches the special edits area of a field or group for an edit of type `edit_type`. The `edit_type` should be one of the following values, which are defined in `smdefs.incl.pl1`:

Edit type	*Contents of edit string*
`NAMED`	Field name
`CPROG`	Name of field validation function
`FE_CPROG`	Name of field entry function
`FX_CPROG`	Name of field exit function
`HELPSCR`	Name of help screen
`HARDHLP`	Name of automatic help screen
`HARDITM`	Name of automatic item selection screen
`ITEMSCR`	Name of item selection screen
`SUBMENU`	Name of pull–down menu screen
`TABLOOK`	Name of screen for table–lookup validation
`NEXTFLD`	Next field (contains both primary and alternate fields)
`PREVFLD`	Previous field (contains both primary and alternate fields)
`TEXT`	Status line prompt

Edit type	*Contents of edit string*
MEMO1 ... MEMO9	Nine arbitrary user-supplied text strings
JPLTEXT	Attached JPL code
CALC	Math expression executed at field exit
CKDIGIT	Flag and parameters for check digit
FTYPE	Data type for inclusion in structure
RETCODE	Return value for menu or return entry field
CMASK	Regular expression for field validation
CCMASK	Regular expression for character validation
CKBOX	Offset and attribute of checkbox in a group
ALTSC_CPROG	Name of alternate scrolling function
KEYSET	Name of keyset associated with screen.
SDATETIME	Date/time field with user format, initialized with system values.
UDATETIME	Date/time field with user format, initialized by the user.
CURRED	Currency field format, see `smdefs.incl.pl1` for details.
NULLFIELD	Null field representation.
RANGEL	Low bound on range; up to 9 permitted
RANGEH	High bound on range; up to 9 permitted
EDT_BITS	Normally for internal use (see `smdefs.incl.pl1` for more information.)

The string returned by `xsm_edit_ptr` contains:

- The total length of the string (including the two overhead bytes and any terminators) in its first byte.

- The edit_type code in its second byte.
- The body of the edit in the subsequent bytes. Refer to the source listing for the file `smdefs.incl.pl1` for specific information on how to interpret each individual edit.

If the field has no edit of type `edit_type`, the returned buffer will contain a zero. If a field has multiple edits of one type, such as RANGEH or RANGEL, then each additional edit is added onto the end of the string following the same pattern as the first one. For example, the first byte would contain the length of the string up to the end of the body of the edit of RANGEH. Adding one to this number would give you the byte that contains the length of the string containing information on RANGEL and so forth.

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

In the case of groups, the edits `PREVFLD`, `NEXTFLD`, `CPROG`, `FE_CPROG`, and `FE_CPROG` may be used to obtain group information.

RETURNS

The first (length) byte of the special edit of the field.
0 if the field or edit is not found.

VARIANTS

```
  buffer = xsm_n_edit_ptr(field_name, edit_type);
```

emsg

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

SYNOPSIS

```
declare message           char(256) varying;
call xsm_emsg(message);
```

DESCRIPTION

This function displays `message` on the status line, if it fits, or in a window if it is too long. If the cursor position display has been turned on (see `xsm_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. The message 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 `xsm_option`.

`xsm_emsg` is identical to `xsm_err_reset`, except that it does not attempt to turn the cursor on before displaying the message. It is similar to `xsm_qui_msg`, which inserts a constant string (normally "ERROR:") before the message.

- Several *percent escapes* provide control over the content and presentation of status messages. The character following the percent sign must be in upper-case. Note that, if a message containing percent escapes is displayed before `xsm_initcrt` is called, the percent escapes will show up in the message.

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 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 zeros. If the following character is not a legal hex digit, then leading zeros 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.incl.pl1`; it is of course the name, not the number, that you want here. The mnemonic must be in upper-case.

If the message begins with a `%B`, JAM will beep the terminal (using `xsm_bel`) before issuing the message.

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 new lines.

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`, JAM will ignore the default error message acknowledgement flag and process (for `%Mu`) or discard (for `%Md`) the next character typed.

Possible hex values for display attribute are defined in `smdefs.incl.pl1`, as shown in the table below:

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Highlights		Background Highlights	
BLANK	0008	B_HILIGHT	8000
REVERSE	0010		
UNDERLN	0020		
BLINK	0040		
HILIGHT	0080		
STANDOUT	0800		
DIM	1000		
ACS (alternate character set)	2000		

Attribute Mnemonic	*Hex Code*	*Attribute Mnemonic*	*Hex Code*
Foreground Colors		Background Colors	
BLACK	0000	B_BLACK	0000
BLUE	0001	B_BLUE	0100
GREEN	0002	B_GREEN	0200
CYAN	0003	B_CYAN	0300
RED	0004	B_RED	0400
MAGENTA	0005	B_MAGENTA	0500
YELLOW	0006	B_YELLOW	0600
WHITE	0007	B_WHITE	0700

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

RELATED FUNCTIONS

```
call xsm_err_reset(message);
call xsm_qui_msg(message);
call xsm_quiet_err(message);
```

err_reset

display an error message and reset the status line

SYNOPSIS

```
declare message           char(256) varying;
call xsm_err_reset(message);
```

DESCRIPTION

The `message` is displayed on the status line until acknowledged it by pressing a key. If `message` is too long to fit on the status line, it is displayed in a window instead. If the cursor position display has been turned on (see `xsm_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. The exact behavior of error message acknowledgement is governed by `xsm_option`. The initial message attribute is set by `xsm_option`, 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 `xsm_emsg`, which does not turn on the cursor, and to `xsm_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. See `xsm_emsg` for details.

RELATED FUNCTIONS

```
call xsm_emsg(message);
call xsm_qui_msg(message);
call xsm_quiet_err(message);
```

fi_path

return the full path name of a file

SYNOPSIS

```
declare buffer            char(256) varying;
declare file_name         char(256) varying;
buffer = xsm_fi_path(file_name);
```

DESCRIPTION

Use this function to find the full path name of a file. The file may be a screen or any other type of file. The file's full path name is returned in `buffer`.

The file name is first sought in the current directory. If that fails, the path given to `xsm_initcrt` is checked. Finally the path defined by `SMPATH` is searched.

RETURNS

0 if the file cannot be found in any path.
Else, The path is returned in `buffer`.

finquire

obtain information about a field

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare field_number      fixed binary(31);
declare which             fixed binary(31);
declare value             fixed binary(31);
value = xsm_finquire(field_number, which);
```

DESCRIPTION

Use this function to obtain various information about a field. The variable `which` is a mnemonic that specifies the particular piece of information desired.

Mnemonics for `which` are defined in the file `smglobs.incl.pl1`. The following values are available:

Mnemonic	*Meaning*
FD_LINE	Line that field is on.
FD_COLM	Column of field's first position.
FD_ATTR	Field attributes (see `smdefs.incl.pl1`).
FD_LENG	Onscreen field length.
FD_ASIZE	Onscreen array size (1 if scalar).
FD_ELT	Onscreen element number.
FD_SHLENG	Shiftable length.
FD_SHINCR	Shift increment.
FD_SHOFS	Current shift offset (number of positions field has been shifted; 0 if shifted to left edge).
FD_SCINCR	Scrolling increment (for Next/Prev page keys).
FD_SCFLAG	Scrolling array circular? (T/F).

Mnemonic	*Meaning*
`FD_SCATTR`	Scrolling occurrence display attributes set with `xsm_i_achg`; zero if onscreen element attributes is to be used. For `xsm_i_finquire` variant only.
`FD_FELT`	First onscreen occurrence of scrolling array (1 if scrolled to top).

RETURNS

The value of `which` if found.
0 otherwise.

VARIANTS

```
value = xsm_e_finquire(field_name, element, which);
value = xsm_i_finquire(field_name, occurrence, which);
value = xsm_n_finquire(field_name, which);
value = xsm_o_finquire(field_number, occurrence, which);
```

RELATED FUNCTIONS

```
value = xsm_gp_inquire(group_name, which);
value = xsm_inquire(which);
value = xsm_iset(which, newval);
buffer = xsm_pinquire(which);
buffer = xsm_pset(which, newval);
```

fldno

get the field number of an array element or occurrence

SYNOPSIS

```
declare field_name        char(256) varying;
declare field_number      fixed binary(31);
field_number = xsm_n_fldno(field_name);
```

DESCRIPTION

NOTE: This function only exists in the `e_`, `i_`, `n_`, and `o_` variations. There is NO `xsm_fldno` since this function determines the field number given other information.

The `e_` variant returns the field number of an array element specified by `field_name` and `element`. If `element` is zero, then `xsm_e_fldno` returns the field number of the named field, or the base element of the named array.

The `i_` and `o_` variants return the number of the field containing the specified occurrence if the occurrence is onscreen, or 0 if the occurrence is offscreen.

The `n_` variant returns the field number of a field specified by name, or the base field number of an array specified by name.

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 occurrence is offscreen.
Otherwise, returns an integer between 1 and the maximum number of fields on the current screen that represents the field number.

VARIANTS

```
field_number = xsm_e_fldno(field_name, element);
field_number = xsm_i_fldno(field_name, occurrence);
field_number = xsm_o_fldno(field_number, occurrence);
```

flush

flush delayed writes to the display

SYNOPSIS

```
call xsm_flush();
```

DESCRIPTION

This function performs delayed writes and flushes all buffered output to the display. It is called automatically via `xsm_input` 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.

RELATED FUNCTIONS

```
call xsm_flush();
call xsm_rescreen();
```

form

display a screen as a form

SYNOPSIS

```
declare screen_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_r_form(screen_name);

declare screen_address    bit(0);
declare status            fixed binary(31);
status = xsm_d_form(screen_address);

declare lib_desc          fixed binary(31);
declare screen_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_l_form(lib_desc, screen_name);
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. These functions do not update the form stack, so it is generally not a good idea to use them with the JAM Executive. To open a form while under the control of the JAM Executive, use a JAM control string or `xsm_jform`.

These functions display the named screen as a base form. Bringing up a screen as a form with `xsm_d_form`, `xsm_l_form`, `xsm_r_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)).

If an error occurs a return of -1 or -2 means that the previously displayed form is still displayed and may be used. Other negative return codes indicate that the display is undefined. The caller should display another form before using Screen Manager functions.

When you use `xsm_r_form` the named screen is sought first in the memory-resident screen list, and if found there is displayed using `xsm_d_form`. It is next sought in all the open screen libraries, and if found is displayed using `xsm_l_form`. Next it is sought on disk in the current directory; then under the path supplied to `xsm_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.

You may save processing time by using `xsm_d_form` to display screens that are memory-resident. Use `bin2pl1` 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. `xsm_r_form` can also display memory–resident screens, if they are properly installed using `xsm_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). `screen_address` is the address of the screen in memory.

You may also save processing time by using `xsm_l_form` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `xsm_l_open`, which you must call before trying to read any screens from a library. Note that `xsm_r_form` also searches any open libraries.

To display a window use `xsm_r_at_cur`, `xsm_r_window`, or one of their variants.

RETURNS

0 if no error occurred
–1 if the screen file's format is incorrect; previous form still displayed and available
–2 if the screen cannot be found or the maximum allowable number of files is already open; previous form still displayed and available
–4 if, after the screen has been cleared, the screen cannot be successfully displayed because of a read error;
–5 if, after the screen was cleared, the system ran out of memory;

RELATED FUNCTIONS

```
status = xsm_r_window(screen_name, start_line, start_column);
status = xsm_r_at_cur(screen_name);
```

formlist

update list of memory–resident files

SYNOPSIS

```
declare name              char(256) varying;
declare address           bit(0);
declare status            fixed binary(31);
status = xsm_formlist(name, address);
```

DESCRIPTION

This function adds a JPL module, keyset, or screen to the memory resident form list. Each member of the list is a structure giving the name of the JPL module, screen, or keyset, as a character string, and its address in memory. This function is commonly called from `main`. It can be called any number of times from an application program to augment to the memory resident list.

The library functions `xsm_r_form`, `xsm_r_window`, `xsm_r_at_cur`, and `xsm_r_keyset` all take a screen or keyset name as a parameter and search for it in the memory–resident list before attempting to read the screen or keyset from disk. The `jpl` command (see the JPL Programmer's Guide) and the function `xsm_jplcall` search the memory resident form list when looking for a JPL procedure to execute.

To make a JPL module, keyset, or screen memory resident, you can use the `bin2pl1` utility to create a static PL/1 structure initialized with the binary content of the object. You must then compile and link the structure with the application executable.

RETURNS

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

RELATED FUNCTIONS

```
call xsm_rmformlist;
```

fptr
get the content of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare buffer             char(256) varying;
buffer = xsm_fptr(field_number);
```

DESCRIPTION

This routine 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.

RETURNS

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

VARIANTS

```
buffer = xsm_e_fptr(field_name, element);
buffer = xsm_i_fptr(field_name, occurrence);
buffer = xsm_n_fptr(field_name);
buffer = xsm_o_fptr(field_number, occurrence);
```

RELATED FUNCTIONS

```
length = xsm_getfield(buffer, field_number);
status = xsm_putfield(field_number, data);
```

ftog
convert field references to group references

SYNOPSIS

```
declare field_number      fixed binary(31);
declare group_occurrence  fixed binary(31);
declare buffer            char(256) varying
buffer = xsm_ftog(field_number, group_occurrence);
```

DESCRIPTION

This function converts field references to group references. Use `xsm_i_gtof` to convert them back.

This function returns the name of the group containing the referenced field and inserts its group occurrence number into the address of `occurrence`.

RETURNS

The group name if found and indirectly through `group_occurrence` the group occurrence number.
0 otherwise and `group_occurrence` is unchanged.

VARIANTS

```
buffer = xsm_e_ftog(field_name, element, group_occurrence);
buffer = xsm_i_ftog(field_name, occurrence, group_occurrence);
buffer = xsm_n_ftog(field_name, group_occurrence);
buffer = xsm_o_ftog(field_number, occurrence,
         group_occurrence);
```

RELATED FUNCTIONS

```
field_number = xsm_i_gtof(group_name, group_occurrence,
         occurrence);
```

ftype

get the data type and precision of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare precision_ptr      fixed binary(31);
declare type               fixed binary(31);
type = xsm_ftype(field_number, precision_ptr);
```

DESCRIPTION

This function analyzes the edits of a field or LDB entry, and returns data type information. First the "type" (FTYPE) edit is checked, then the "currency" edit, the "date/time" edit, and finally the "character" edit.

Note that this differs from the functionality of `xsm_rdstruct`, `xsm_wrtstuct`, `xsm_rrecord`, and `xsm_wrecord`. These functions only test the type and character edits. They use the currency edit only to determine the precision of a numeric field that has no type edit.

This function returns an integer containing the data type code, plus any applicable flags. The data type codes and flags are detailed in the tables below.

Data Type Code	*Meaning*
FT_CHAR	Type edit is *char string*; or character edit is *unfiltered*, *letters only*, *alphanumeric*, or *regular expression*
FT_INT	Type edit is *int*
FT_UNSIGNED	Type edit is *unsigned int*; or character edit is *digit*
FT_SHORT	Type edit is *short int*
FT_LONG	Type edit is *long int*
FT_FLOAT	Type edit is *float*
FT_DOUBLE	Type edit is *double*; or character edit is *numeric*
FT_ZONED	Type edit is *zoned dec.*
FT_PACKED	Type edit is *packed dec.*

Data Type Code	*Meaning*
`DT_YESNO`	Character edit is *yes/no*
`DT_CURRENCY`	Currency edit
`DT_DATETIME`	Date/time edit

Flag	*Meaning*
`DF_NULL`	Null edit
`DF_REQUIRED`	Data required edit (not applicable to LDB)
`DF_WRAP`	Word wrap edit
`DF_OMIT`	Type edit is *omit.*

To determine the data type code, check this integer for each flag in the fashion of the example field function shown on page 14, starting with DF_OMIT and working up the list.. The value remaining will be the data type code.

Note that `FT_OMIT` is not listed as one of the data types. A field that has the type edit *omit* will return the data type determined by any of the other edits, as well as a flag indicating that it has the *omit* type edit.

The function will put the precision of float, double and currency values in the `precison_ptr` argument.

RETURNS

major data type code plus any applicable flags (see tables above).
0 if field is not found

VARIANTS

```
  type = xsm_n_ftype(field_number, precision_ptr);
```

fval
force field validation

SYNOPSIS

```
declare field_number      fixed binary(31);
declare status            fixed binary(31);
status = xsm_fval(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 `xsm_aprotect`. 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
field validation	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 field with embedded punctuation must contain at least one non–blank non–punctuation character in order to be considered non–empty; otherwise any non blank character makes the field non–empty.

Math expressions, JPL functions and field validation functions are never skipped, since they can alter fields other than the one being validated.

Field validation is performed automatically within `xsm_input` when the cursor exits a field via the `TAB` or `NL` logical keys. All fields on a screen are validated when `XMIT` is pressed (see `xsm_s_val`). 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

```
status = xsm_e_fval(array_name, element);
status = xsm_i_fval(field_name, occurrence);
status = xsm_n_fval(field_name);
status = xsm_o_fval(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_n_gval(group_name);
status = xsm_s_val();
```

getcurno

get current field number

SYNOPSIS

```
declare field_number      fixed binary(31);
field_number = xsm_getcurno();
```

DESCRIPTION

This function 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.

RELATED FUNCTIONS

```
occurrence = xsm_occur_no();
```

getfield

copy the contents of a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare length              fixed binary(31);
length = xsm_getfield(buffer, field_number);
```

DESCRIPTION

This function 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. The variants that reference a field by name will attempt to get data from the corresponding LDB entry if there is no such field on the screen (except that the order is reversed during screen entry/exit processing).

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.

In variants that take `name` as an argument, either the name of a field or a group may be used. In the case of groups, `xsm_isselected` is preferred to `xsm_getfield` for determining whether or not a group occurrence is selected. If `xsm_n_getfield` is called on a radio button, the value in `buffer` will be the occurrence number of the selected item. If `xsm_i_getfield` is called on a checklist, the value in the first occurrence of the array will be the number of the first selected item in the group, the value in the second occurrence will be the number of the next selected item in the group and so on. If a checklist has, for example, three items selected, the fourth array occurrence will be empty.

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

RETURNS

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

VARIANTS

```
length = xsm_e_getfield(buffer, name, element);
length = xsm_i_getfield(buffer, name, occurrence);
length = xsm_n_getfield(buffer, name);
length = xsm_o_getfield(buffer, field_number, occurrence);
```

RELATED FUNCTIONS

```
buffer = xsm_fptr(field_number);
status = xsm_isselected(group_name, group_occurrence);
status = xsm_putfield(field_number, data);
```

getjctrl
get control string associated with a key

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key             fixed binary(31);
declare default         fixed binary(31);
declare buffer          char(256) varying
buffer = xsm_getjctrl(key, 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 `SMINICTRL` setup variables. See the section on setup in the Configuration Guide for further information.

This function searches one of the tables for `key`, a logical key mnemonic found in `smkeys.incl.pl1`, and returns a the associated control string. If `default` is zero, the table for the current screen is searched; otherwise, the system-wide table is searched.

RETURNS

The control string
0 if none is found.

RELATED FUNCTIONS

```
status = xsm_putjctrl(key, control_string, default);
```

getkey
get logical value of the key hit

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                fixed binary(31);
key = xsm_getkey();
```

DESCRIPTION

This function gets and interprets keyboard input and returns the logical value to the calling program. Normal characters are returned unchanged. Logical keys are interpreted according to a key translation file for the particular terminal you are using. See the Keyboard Input section in this guide, the Key Translation section in the Configuration Guide, and the modkey section in the Utilities Guide. xsm_getkey is normally not needed for application programming, since it is called by xsm_input.

Logical 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.incl.pl1. A few logical keys, such as LOCAL PRINT and RESCREEN, are processed locally in xsm_getkey and not returned to the caller.

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

There are a number of user–installed functions that may be called by xsm_getkey. For further information see the section on installing functions in the Programmer's Guide.

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 xsm_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 (i.e., functions that call xsm_getkey. If you call this function, do not expect function key control strings to work.

The multiplicity of calls to user functions in `xsm_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 `xsm_getkey`. The process of key translation is deliberately omitted, for the sake of clarity; that algorithm is presented separately, in the keyboard translation section of the Programmer's Guide.

***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. When the keyboard is read, then the asynchronous function (if one is installed) is called during periods of keyboard inactivity.

*** 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.
- If the key is a soft key, return its logical value.

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.

RELATED FUNCTIONS

```
old_flag = xsm_keyfilter(flag);
return_value = xsm_ungetkey(key);
```

gofield

move the cursor into a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_gofield(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 and in a left-justified field, in the leftmost. In either case, if the field has embedded punctuation, the cursor goes to the nearest position not occupied by a punctuation character. Use `xsm_off_gofield` to place the cursor in position other than that of the first character of a field.

When called to position the cursor in a scrollable array, `xsm_o_gofield` and `xsm_i_gofield` return an error if the occurrence number passed exceeds by more than 1 the number of allocated occurrences in the specified array. If the desired occurrence is offscreen, it is scrolled on-screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

RETURNS

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

VARIANTS

```
status = xsm_e_gofield(field_name, element);
status = xsm_i_gofield(field_name, occurrence);
status = xsm_n_gofield(field_name);
status = xsm_o_gofield(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_off_gofield(field_number, offset);
```

gp_inquire
obtain information about a group

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare group_name        char(256) varying;
declare which             fixed binary(31);
declare value             fixed binary(31);
value = xsm_gp_inquire(group_name, which);
```

DESCRIPTION

Use this function to obtain various information about group. The variable `which` is a mnemonic that specifies the particular piece of information desired.

Mnemonics for `which` are defined in the file `smglobs.incl.pl1`. They are:

Mnemonic	*Meaning*
GP_NOCCS	Number of occurrences in the group (sum of number of occurrences of all fields/arrays in group)
GP_FLAGS	Flags

RETURNS

The value of `which`, if found, or
–1 otherwise.

gtof

convert a group name and index into a field number and occurrence

SYNOPSIS

```
declare group_name        char(256) varying;
declare group_occurrence  fixed binary(31);
declare occurrence        fixed binary(31);
declare field_number      fixed binary(31);
field_number = xsm_i_gtof(group_name, group_occurrence,
         occurrence);
```

DESCRIPTION

NOTE: This function only exists in the `i_` variation. There is no `xsm_gtof` since groups cannot be referenced by number.

Use this function to convert a group name and group_occurrence into a field number and occurrence. The variable `group_name` is the name of the group and `group_occurrence` is the specific field within the group.

The function returns the field number of the referenced field and inserts the occurrence number into the memory location addressed by `occurrence`.

Using this function allows you to use other JAM library routines to manipulate group fields by converting group references into field references. For instance, if you wanted to access text from a specific field within a group you would need to use `xsm_i_gtof` to get the field and occurrence number before you could use the function `xsm_o_getfield` to retrieve the text.

RETURNS

The field number if found.
0 otherwise.

RELATED FUNCTIONS

```
buffer = xsm_ftog(field_number, group_occurrence);
```

gval

force group validation

SYNOPSIS

```
declare group_name        char(256) varying;
declare status            fixed binary(31);
status = xsm_n_gval(group_name);
```

DESCRIPTION

NOTE: This function only exists in the `xsm_n_gval` variation. There is no `xsm_gval` since groups cannot be referenced by number.

Use this function to force the execution of a group's validation function. Use `xsm_s_val` to validate all fields and groups on the screen.

RETURNS

–1 if the group fails any validation.
–2 if the group name is invalid.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_fval(field_number);
status = xsm_s_val();
```

gwrap
get the contents of a wordwrap array

SYNOPSIS

```
declare buffer              char(256) varying;
declare field_number        fixed binary(31);
declare buffer_length       fixed binary(31);
declare length              fixed binary(31);
length = xsm_gwrap(buffer, field_number, buffer_length);
```

DESCRIPTION

This function copies the contents of the array specified by `field_number`, one occurrence at a time, into `buffer`, up to the size specified by `buffer_length`. A space is inserted before every non–empty occurrence, except the first.

The variant `xsm_o_gwrap` copies the contents of the array, beginning with the specified occurrence.

RETURNS

The length of transferrable data. If this is greater than `buffer_length`, then the data was truncated.

–1 if the field number is invalid or `buffer_length` is ≤ 0.

VARIANTS

```
status = xsm_o_gwrap(buffer, field_number, occurrence,
         buffer_length);
```

RELATED FUNCTIONS

```
status = xsm_pwrap(field_number, text);
```

hlp_by_name
display help window

SYNOPSIS

```
declare help_screen        char(256) varying;
declare status             fixed binary(31);
status = xsm_hlp_by_name(help_screen);
```

DESCRIPTION

The named screen is displayed and processed as a normal help screen, including input processing for the current field (if any).

Refer to the Author's Guide for instructions on how to create various kinds of help screens and for details of the behaviour of help screens.

RETURNS

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

home

home the cursor

SYNOPSIS

```
declare field_number      fixed binary(31);
field_number = xsm_home();
```

DESCRIPTION

This function moves the cursor to the first enterable position of the first tab–unprotected field on the screen. If the screen has no tab–unprotected fields, the cursor is moved to the first line and column of the topmost screen. However, if you are using the JAM Executive, the cursor may not be visible if there are no tab–unprotected fields.

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

This function doesn't immediately trigger field entry, exit, or validation processing. Processing is based on the cursor position when control returns to `xsm_input`.

When the JAM logical key HOME is hit, `xsm_home` is called.

RETURNS

The number of the 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.

RELATED FUNCTIONS

```
call xsm_backtab();
status = xsm_gofield(field_number);
call xsm_last();
call xsm_nl();
call xsm_tab();
```

i_
variants that take a field name and occurrence number

SYNOPSIS

```
declare field_name        char(256) varying;
declare occurrence        fixed binary(31);
call xsm_i_...(field_name, occurrence, ...);
```

DESCRIPTION

The `i_` variants each refer to data by field name and occurrence number. An occurrence is a slot within an array in which data may be stored. Occurrences may be either on or off-screen. Since JAM treats an individual field as an array with one field, even a single non-scrolling field is considered to have one occurrence. The JAM library contains routines that allow you to manipulate individual occurrences during run-time.

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, `xsm_i_amt_format` is described under `xsm_amt_format`.

If the named field is not part of the screen currently being displayed, these functions will attempt to retrieve or change its value in the local data block.

ininames

record names of initial data files for local data block

SYNOPSIS

```
declare name_list         char(256) varying;
declare status            fixed binary(31);
status = xsm_ininames(name_list);
```

DESCRIPTION

Use this routine to set up a list of initialization files for local data block entries. The file names in the single string `name_list` should be separated by commas, semicolons or blanks. There may be up to ten file names. You may achieve the same effect by defining the `SMININAMES` variable in your setup file to the list of names. See setup files in the Configuration Guide and the Data Dictionary chapter of the Author's Guide for details.

The files contain pairs of names and values, which are used to initialize local data block entries by `xsm_ldb_init`. This function is called during JAM initialization, so `xsm_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"
```

Entries of all scopes may be freely mixed within all files. We recommend, however, that entries be grouped in files by scope if you are planning to use `xsm_lreset`. Use `xsm_lreset` to clear all entries of a given scope before reinitializing them from a single file.

RETURNS

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

RELATED FUNCTIONS

```
call xsm_ldb_init();
status = xsm_lreset(file_name, scope);
```

initcrt

initialize the display and JAM data structures

SYNOPSIS

```
declare path                char(256) varying;
call xsm_initcrt(path);


declare path                char(256) varying;
call xsm_jinitcrt(path);


declare path                char(256) varying;
call xsm_jxinitcrt(path);
```

DESCRIPTION

The function `xsm_initcrt` is intended for use only with a user-written executive. It is called automatically by the JAM Executive.

`xsm_initcrt` 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 `xsm_option`, and those that install functions or configuration files such as `xsm_uinstall` or `xsm_vinit`, are the only kind that may be called before `xsm_initcrt`.

The argument `path` is a directory to be searched for screen files by `xsm_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, `xsm_initcrt` performs several initializations:

1. It calls a user-defined initialization routine.
2. It determines the terminal type, if possible by examining the environment (TERM or SMTERM), otherwise by asking the user.
3. It executes the setup files defined by the environment variables SMVARS 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 display and exits.
6. It initializes the operating system display and keyboard channels, and clears the display.

The functions `xsm_jinitcrt` and `xsm_jxinitcrt` are called by `jmain.pl1` and `jxmain.pl1` respectively for applications that use the JAM Executive. They, in turn, call `xsm_initcrt`.

RELATED FUNCTIONS

```
call xsm_resetcrt();
call xsm_jresetcrt();
call xsm_jxresetcrt();
```

input

open the keyboard for data entry and menu selection

SYNOPSIS

```
declare initial_mode       fixed binary(31);
declare key                fixed binary(31);
key = xsm_input(initial_mode);
```

DESCRIPTION

This routine is only used if you are writing your own executive. Use `xsm_input` to open the keyboard for either data entry or menu selection.

You specify which mode you wish to be in with the argument `initial_mode`. Possible choices are defined in `smdefs.incl.pl1`. They are:

- IN_AUTO JAM checks whether you specified the screen to begin menu mode or data entry mode (See Author's Guide).
- IN_DATA Start in data entry mode.
- IN_MENU Start in menu mode.

In most cases you will want to use IN_AUTO mode. Use IN_DATA or IN_MENU if you wish to override the setting that you specified via the Screen Editor.

This routine calls `xsm_getkey` to get and interpret keyboard entry. While in data entry mode ASCII data is entered into fields on the screen, subject to any restrictions or edits that were defined for the fields. The routine returns to the calling program when it encounters a logical key, when a "return entry" field is filled or tabbed from, or a key with the return bit set in the routing table.

If the logical value returned by `xsm_getkey` is TRANSMIT, EXIT, HELP, or a cursor position key, the processing is determined by a routing table. The routing options are set with `xsm_keyoption`. See `xsm_keyoption` for more information.

This function replaces version 4.0 `xsm_choice`, `xsm_menu_proc`, and `xsm_openkeybd`. These functions only exist in your version 5.0 library for backward compatibility. We strongly suggest that you do not use them in the future.

RETURNS

The key hit by the end–user that terminated the call to `xsm_input`, or the first character of the selected menu item.

inquire

obtain value of a global integer variable

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare which            fixed binary(31);
declare value            fixed binary(31);
value = xsm_inquire(which);
```

DESCRIPTION

This function is used to obtain the current integer value of a global variable. The desired variable is specified by `which`. If the value of `which` is a true/false (the flag is on or off) value then `xsm_inquire` returns `1` for true and `0` for false. If you wish to modify a global integer value use `xsm_iset`. The permissible values for `which` are defined in `smglobs.incl.pl1`. The following values are available:

Mnemonic	*Meaning*
I_NODISP	In non-display mode? (T/F). Initially FALSE, setting TRUE causes no further changes to the actual display, although JAM's internal screen image is kept up to date. This was release 4's `sm_do_not_display` flag.
I_INSMODE	In insert mode? (T/F).
I_INXFORM	In JAM screen editor? (T/F) Field validation routines are generally still called when in editor; they can check this flag to disable certain features.
I_MXLINES	Number of lines available for use by JAM on the hardware display.
I_MXCOLMS	Number of columns available for use by JAM on the hardware display.
I_NLINES	Maximum number of lines available on the current screen, not including the status line.
I_NCOLMS	Maximum number of columns available on the current screen, not including the status line.
I_INHELP	Help screen is currently displayed? (T/F)

Mnemonic	*Meaning*
I_BSNESS	Screen manager is in control of display? (T/F). (Replaces rel. 4 inbusiness function).
I_BLKFLGS	Block mode is turned on? (T/F)
SC_VFLINE	First screen line of viewport (0–based).
SC_VFCOLM	First screen column of viewport (0–based).
SC_VNLINE	Number of lines in viewport.
SC_VNCOLM	Number of columns in viewport.
SC_VOLINE	Line offset of viewport.
SC_VOCOLM	Column offset of viewport.
SC_NLINE	Number of lines in screen.
SC_NCOLM	Number of columns in screen.
SC_CLINE	Current line number in screen.
SC_CCOLM	Current column number in screen.
SC_NFLDS	Number of fields on screen.
SC_NGRPS	Number of groups on screen.
SC_BKATTR	Background attributes of screen.
SC_BDCHAR	Border character of screen.
SC_BDATTR	Border attributes of screen.

RETURNS

If the argument corresponds to an integer global variable, the current value of that variable is returned.

1 true, flag is set to on.
0 false, flag is set to off.
–1 otherwise.

RELATED FUNCTIONS

```
value = xsm_finquire(field_number, which);
value = xsm_gp_inquire(group_name, which);
value = xsm_iset(which, newval);
buffer = xsm_pinquire(which);
buffer = xsm_pset(which, newval);
```

intval

get the integer value of a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare value             fixed binary(31);
value = xsm_intval(field_number);
```

DESCRIPTION

This function returns the integer value of the data contained in the field specified by `field_number`. Any punctuation characters in the field, except a leading plus or minus sign, are ignored.

RETURNS

The integer value of the specified field.
0 if the field is not found.

VARIANTS

```
value = xsm_e_intval(field_name, element);
value = xsm_i_intval(field_name, occurrence);
value = xsm_n_intval(field_name);
value = xsm_o_intval(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_itofield(field_number, value);
```

ioccur
insert blank occurrences into an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
declare count              fixed binary(31);
declare lines_inserted     fixed binary(31);
lines_inserted = xsm_o_ioccur(field_number, occurrence, count);
```

DESCRIPTION

NOTE: This function only exists in the `i_` and `o_` variations. There is no `xsm_ioccur`, since this function applies only to arrays.

Inserts `count` blank occurrences before the specified occurrence, moving that occurrence and all following occurrences down. If inserting that many would move an occurrence past the end of its array, fewer will be inserted. If the array is scrollable, then this function may allocate up to `count` new occurrences. This function never increases the maximum number of occurrences an array can contain; `xsm_sc_max` does that. If `count` is negative, occurrences will be deleted instead, subject to limitations described in the page for `xsm_doccur`. In addition, this function never inserts more blank occurrences than the number of blank occurrences following the last non-blank occurrence (that is, it won't push data off the end of an array).

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.

Any clearing-unprotected synchronized arrays will have the same operations performed on them as the referenced array. Synchronized arrays that are protected from clearing will remain constant. Therefore, a protected array may be used to number a list of data stored in a non-protected synchronized array as it grows and shrinks.

This function is normally bound to the INSERT LINE key.

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

```
lines_inserted = xsm_i_ioccur(field_name, occurrence, count);
```

is_no
test field for no

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_is_no(field_number);
```

DESCRIPTION

The first character of the field contents specified by `field_number` is compared with the first letter of the `SM_NO` entry in the message file, ignoring case. If they match this function will return a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to a `Y` in the field or because of some other problem. If you wish to check for a `Y` response use `xsm_is_yes`.

This function is ordinarily used with one-letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are `y`, `n`, or space (which means `n`). Unlike other functions, `xsm_is_no` does ***not ignore leading blanks.***

RETURNS

1 if the field's first character matches the first character of the `SM_NO` entry in the message file.
0 otherwise.

VARIANTS

```
status = xsm_e_is_no(field_name, element);
status = xsm_i_is_no(field_name, occurrence);
status = xsm_n_is_no(field_name);
status = xsm_o_is_no(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_is_yes(field_number);
```

is_yes
test field for yes

SYNOPSIS

```
declare field_number        fixed binary(31);
declare status              fixed binary(31);
status = xsm_is_yes(field_number);
```

DESCRIPTION

The first character of the field contents specified by `field_number` is compared with the first letter of the `SM_YES` entry in the message file, ignoring case. If they match this function will return a 1 for true. If they do not match for any reason, the function returns a 0 for failure. There is no way to tell if the failure is due to an `N` in the field or because of some other problem. If you wish to check for an `N` response use `xsm_is_no`.

This function is ordinarily used with one-letter fields possessing the yes/no character edit. In this case, the only characters allowed in the field are `y`, `n`, or space (which means `n`). Unlike other functions, `xsm_is_yes` does *not ignore leading blanks*.

RETURNS

1 if the field's first character matches the first character of the `SM_YES` entry in the message file.
0 otherwise.

VARIANTS

```
status = xsm_e_is_yes(field_name, element);
status = xsm_i_is_yes(field_name, occurrence);
status = xsm_n_is_yes(field_name);
status = xsm_o_is_yes(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_is_no(field_number);
```

isabort

test and set the abort control flag

SYNOPSIS

```
declare flag              fixed binary(31);
declare old_flag          fixed binary(31);
old_flag = xsm_isabort(flag);
```

DESCRIPTION

Use `xsm_isabort` to set the abort flag to the value of `flag`, and return the old value. `flag` must be one of the following as defined in `smdefs.incl.pl1`:

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 `xsm_input`. The triggering event is the detection of an abort condition by `xsm_getkey`, either an ABORT keystroke or a call to this function with `ABT_ON` (such as from an asynchronous function).

This function enables application code to verify the existence of an abort condition by testing the flag, as well as to establish one.

RETURNS

The previous value of the abort flag.

iset

change value of integer global variable

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare which           fixed binary(31);
declare newval          fixed binary(31);
declare value           fixed binary(31);
value = xsm_iset(which, newval);
```

DESCRIPTION

JAM has a number of global parameters and settings. This function is used to modify the current value of integer globals. The variable to change is specified by `which`. The new value is specified by `newval`. If you wish to get the value of a global integer use `xsm_inquire`.

The permissible values for the argument `which` are defined in the header file `smglobs.incl.pl1`. The following values are available:

Mnemonic	*Quantity*	*Meaning*
`I_NODISP`	0	Disable updating of display.
	1	Enable updating of display.
`I_INSMODE`	0	Enter overtype mode.
	1	Enter insert mode.

RETURNS

If `which` is one of the permissible values, the former value of the appropriate variable is returned.
1 True, the flag was set to on.
0 False, the flag was set to off.
–1 otherwise.

RELATED FUNCTIONS

```
value = xsm_finquire(field_number, which);
value = xsm_gp_inquire(group_name, which);
value = xsm_inquire(which);
buffer = xsm_pinquire(which);
buffer = xsm_pset(which, newval);
```

isselected

determine whether a radio button or checklist occurrence has been selected

SYNOPSIS

```
declare group_name        char(256) varying;
declare group_occurrence  fixed binary(31);
declare status            fixed binary(31);
status = xsm_isselected(group_name, group_occurrence);
```

DESCRIPTION

This function lets you check to see whether or not a specific occurrence within a check list or radio button has been selected. The selection is referenced by the group name and occurrence number. If the occurrence is selected, `xsm_isselected` returns a `1`. A 0 is returned if the occurrence is not selected. See the Author's Guide for a more detailed discussion of groups.

Radio button and checklist occurrences are selected by using `xsm_select`. Using `xsm_select` on a radio button occurrence causes the current selection to be deselected. Checklist occurrences are deselected with `xsm_deselect`.

RETURNS

–1 arguments do not reference a checklist or radio button occurrence.
0 not selected.
1 selected.

RELATED FUNCTIONS

```
status = xsm_deselect(group_name, group_occurrence);
length = xsm_getfield(buffer, field_number);
value = xsm_intval(field_number);
status = xsm_select(group_name, group_occurrence);
```

issv
determine if a screen is in the saved list

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_issv(screen_name);
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function searches the save list for a single screen and returns 1 is the screen is found (See `xsm_svscreen`).

This function is generally called by applications at screen entry to avoid re-acquiring data (via a database query) for previously saved screens. To accomplish this, first use `xsm_svscreen` to add the screen to the save list upon screen exit. Next, use `xsm_issv` to check the save list upon screen entry. If the screen is on the save list, you know that it has been previously displayed.

RETURNS

1 if the screen is in the saved list.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_svscreen(screen_list, count);
```

itofield

write an integer value to a field

SYNOPSIS

```
declare field_number        fixed binary(31);
declare value               fixed binary(31);
declare status              fixed binary(31);
status = xsm_itofield(field_number, value);
```

DESCRIPTION

The integer passed to `xsm_itofield` 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

```
status = xsm_e_itofield(field_name, element, value);
status = xsm_i_itofield(field_name, occurrence, value);
status = xsm_n_itofield(field_name, value);
status = xsm_o_itofield(field_number, occurrence, value);
```

RELATED FUNCTIONS

```
value = xsm_intval(field_number);
```

jclose

close current window or form under **JAM** Executive control

SYNOPSIS

```
declare status            fixed binary(31);
status = xsm_jclose();
```

DESCRIPTION

The active screen is closed, and the display is restored to the state before the screen was opened. `xsm_jclose` should only be used when the JAM Executive is in use.

In the case of closing a form, `xsm_jclose` pops the form stack and calls `xsm_jform` to display the screen on the top of the form stack.

In the case of closing a window, `xsm_jclose` calls `xsm_close_window`. Since windows are stacked, the effect of closing a window is to return to the previous window. The cursor reappears at the same position it had before the window was opened.

RETURNS

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

RELATED FUNCTIONS

```
status = xsm_close_window();
status = xsm_jform(screen_name);
status = xsm_jwindow(screen_name);
```

jform

display a screen as a form under **JAM** control

SYNOPSIS

```
declare screen_name        char(256) varying;
declare status             fixed binary(31);
status = xsm_jform(screen_name);
```

DESCRIPTION

This function must be used with the JAM Executive. If you are not using the JAM Executive, use `xsm_r_form` or one of its variants. If you wish to display a window under JAM control, use `xsm_jwindow`.

This function displays the named screen as a form. You may close the form with `xsm_jclose`, or leave the task to the JAM Executive (e.g., when the user presses the EXIT key). 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 placed on top of the JAM's form stack.

The difference between `xsm_jform` and `xsm_r_form`, other than the function arguments, is that only `xsm_jform` manipulates the form stack. Since `xsm_jform` calls `xsm_r_form`, refer to `xsm_r_form` for information on other details, such as how the screen to be displayed is found.

The character string `screen_name` uses the same format as that of a JAM control string that displays a form. In addition to the screen's name, you may optionally specify the position of the form on the physical display, the size of the viewport, and which portion of the form will be positioned in the viewport's top-left corner. See the Authoring Reference in the Author's Guide for details of viewport positioning. The following are all legal strings:

```
status = xsm_jform('form');
```

Display form's first row and column at the top-left corner of the physical display.

```
status = xsm_jform('(20,10)form');
```

Display form's first row and column at the 20th row and 10th column of the physical display.

```
status = xsm_jform('(20,10,15,8)form');
```

Display the first row and column of the form at the 20th row and 10th column of the physical display in viewport that is 15 rows by 8 columns.

A form may be larger than the viewport. If the viewport does not fit on the screen where indicated, JAM will attempt to place it entirely on the display at a different location. If you specify a viewport that is larger than the physical display, the viewport will be the size of the physical display. If you wish to change the viewport size after the window is displayed, use `xsm_viewport`.

RETURNS

0 if no error occurred.
–1 if the screen file's format is incorrect.
–2 if the screen cannot be found.
–4 if, after the display has been cleared, the screen cannot be successfully displayed because of a read error.
–5 if, after the display was cleared, the system ran out of memory.

RELATED FUNCTIONS

```
status = xsm_r_form(screen_name);
status = xsm_jwindow(screen_name);
```

jplcall

execute a JPL jpl procedure

SYNOPSIS

```
declare jplcall_text      char(256) varying;
declare return_value      fixed binary(31);
return_value = xsm_jplcall(jplcall_text);
```

DESCRIPTION

This function executes a JPL procedure precisely as if the following JPL statement were executed from within a JPL procedure:

```
jpl jplcall_text
```

For example, if the value of `jplcall_text` were:

```
verifysal :name 50000
```

then

and

```
jpl verifysal :name 50000
```

would be equivalent. See the JPL Programmer's Guide for further information on the JPL `jpl` command.

RETURNS

–1 if the procedure could not be loaded.
Otherwise, the value returned by the JPL procedure.

jplload

execute the JPL load command

SYNOPSIS

```
declare module_name_list  char(256) varying;
declare status            fixed binary(31);
status = xsm_jplload(module_name_list);
```

DESCRIPTION

This function is the PL/1 interface to the JPL `load` command. Use this command to load one or more modules into memory.

The character string `module_name_list` may be one or more module names. Separate module names with a space.

Calling `xsm_jplload` has precisely the same effect as using the JPL `load` command. See the JPL Programmer's Guide for further information on the JPL `load` command.

Use `xsm_jplunload` to remove a module from memory.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_jplpublic(module_name_list);
status = xsm_jplunload(module_name);
```

jplpublic

execute the JPL public command

SYNOPSIS

```
declare module_name_list  char(256) varying;
declare status            fixed binary(31);
status = xsm_jplpublic(module_name_list);
```

DESCRIPTION

This function is the PL/1 interface to the JPL `public` command. Use this command to load one or more modules into memory.

The character string `module_name` may be one or more module names. Separate module names with a space.

Calling `xsm_jplpublic` has precisely the same effect as using the JPL `public` command. See the JPL Programmer's Guide for further information on the JPL `public` command.

Use `xsm_jplunload` to remove a module from memory.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_jplload(module_name_list);
status = xsm_jplunload(module_name);
```

jplunload

execute the JPL unload command

SYNOPSIS

```
declare module_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_jplunload(module_name);
```

DESCRIPTION

This function is the PL/1 interface to the JPL `unload` command. Use this command to remove one or more modules from memory. Modules are read into memory by using either `xsm_jplpublic` or `xsm_jplload` or via the corresponding JPL commands.

Calling `xsm_jplunload` has precisely the same effect as using the JPL `unload` command. See the JPL Programmer's Guide for further information on the JPL `unload` command.

The character string `module_name` may be one or more module names. Separate module names with a space.

RETURNS

–1 if there is an error.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_jplload(module_name_list);
status = xsm_jplpublic(module_name_list);
```

jtop
start the **JAM** Executive

SYNOPSIS

```
declare screen_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_jtop(screen_name);
```

DESCRIPTION

All applications using the JAM Executive must include a call to `xsm_jtop`. This function starts the JAM Executive. The argument `screen_name` is the name of the first screen that your application displays. It will be displayed as a form. Once `xsm_jtop` is called the JAM Executive is in control until the user exits the application.

The JAM Executive makes calls to various JAM functions that handle all of the tasks needed to control the flow of an application such as opening the keyboard for input, opening and closing forms and windows, and processing all control strings.

If you do not use `xsm_jtop` you will have to write your own procedures to control the flow of your application. See the JAM Development Overview for a more detailed discussion of the JAM Executive.

RETURNS

0 Always.

jwindow

display a window at a given position under **JAM** control

SYNOPSIS

```
declare screen_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_jwindow(screen_name);
```

DESCRIPTION

This function must be used with the JAM Executive. If you are not using the JAM Executive, use `xsm_r_window` or one of its variants. If you wish to display a form under JAM control, use `xsm_jform`.

This function displays the named screen as a window, by calling `xsm_r_window`. You may close the window with a call to `xsm_jclose`, or leave the task to the JAM Executive (e.g., when the user presses the EXIT key).

There is currently no difference between `xsm_jwindow` and `xsm_r_window` except for their arguments (although `xsm_jwindow` is not supported unless the JAM Executive is in use). See the description of `xsm_r_window` for the details of the behavior of `xsm_jwindow`.

The character string `screen_name` uses a format similar to that of a JAM control string that displays a window. Use a single ampersand to specify a stacked window and a double ampersand to specify a sibling window. If the ampersand is omitted, then the screen will be opened as a stacked window. In addition to the screen's name, you may optionally specify the position of the window on the physical display, the size of the viewport, as well as which portion of the window will be positioned in the viewport's top-left corner. The positioning and sizing syntax is identical to that of `xsm_jform`. See `xsm_jform` for examples of acceptable strings.

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

RELATED FUNCTIONS

```
status = xsm_jclose();
status = xsm_jform(screen_name);
status = xsm_r_window(screen_name, start_line, start_column);
```

keyfilter

control keystroke record/playback filtering

SYNOPSIS

```
declare flag              fixed binary(31);
declare old_flag          fixed binary(31);
old_flag = xsm_keyfilter(flag);
```

DESCRIPTION

This function turns the keystroke record/playback mechanism of `xsm_getkey` on (`flag` = 1) or off (`flag` = 0). If no key recording or playback function has been installed, turning the mechanism on has no effect.

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

RETURNS

The previous value of the filter flag.

RELATED FUNCTIONS

```
key = xsm_getkey();
```

keyhit

test whether a key has been typed ahead

SYNOPSIS

```
declare interval          fixed binary(31);
declare status            fixed binary(31);
status = xsm_keyhit(interval);
```

DESCRIPTION

This function 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

0 if no key is available,
non-0 otherwise.

RELATED FUNCTIONS

```
key = xsm_getkey();
```

keyinit

initialize key translation table

SYNOPSIS

```
declare key_address       bit(0);
declare status            fixed binary(31);
status = xsm_keyinit(key_address);
```

DESCRIPTION

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

To install a memory-resident key translation file, `key_address` must contain the address of a key translation table created using the `key2bin` and `bin2pl1` utilities.

RETURNS

0 if the key file is successfully installed.
Program exit if the key file is invalid.

VARIANTS

```
status = xsm_n_keyinit(key_file);
```

keylabel

get the printable name of a logical key

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare buffer             char(256) varying;
declare key                fixed binary(31);
buffer = xsm_keylabel(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:

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	
PF1-PF24		SPF1-SPF24		APP1-APP24		SFT1-SFT24	

If the key code is invalid (not one defined in `smkeys.incl.pl1`), this function returns an empty string.

RETURNS

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

keyoption

set cursor control key options

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key              fixed binary(31);
declare mode             fixed binary(31);
declare newval           fixed binary(31);
declare oldval           fixed binary(31);
oldval = xsm_keyoption(key, mode, newval);
```

DESCRIPTION

Use `xsm_keyoption` to alter at run–time the behavior of `xsm_input` when a particular key is pressed. The default values for key options are built in to JAM. This function only works with cursor control keys. Cursor control keys include all JAM logical keys, *except* for PF, SPF, and APP keys. See "Key File" in the Configuration Guide.

There are three different possible values for `mode`: `KEY_ROUTING`, `KEY_GROUP`, and `KEY_XLATE`. The mnemonics that they use are defined in `smkeys.incl.pl1`. All of these modes draw on the following values for `key`.

Logical Key Mnemonics							
EXIT	XMIT	HELP	FHLP	BKSP	TAB	NL	BACK
HOME	DELE	INS	LP	FERA	CLR	SPGU	SPGD
LSHF	RSHF	LARR	RARR	DARR	UARR	REFR	EMOH
INSL	DELL	ZOOM	SFTS	MTGL	VWPT	MOUS	

■`KEY_ROUTING`

Allows access to the EXECUTE and RETURN bits of the routing table. This mode is generally used to disable a key or to control explictly what action is taken when a key is hit. The following mnemonics may be assigned to `newval`:

1. `KEY_IGNORE` Disables `key`. JAM does nothing when `key` is struck.
2. `EXECUTE` The action normally associated with `key` is executed. May be ored with `RETURN`.

3. RETURN No action is performed, but the function returns to the caller in your code. Used to gain direct control of key's action. May be ored with EXECUTE.

■KEY_GROUP

Allows access to the group action bits. Use this function to control the action of the cursor when it is within a group. The following values may be assigned to newval:

1. VF_GROUP Obey group semantics. Hitting key will cause the cursor to move to the next field within the group in the indicated direction. If this mnemonic is ored with VF_CHANGE the cursor will exit the group in the indicated direction.
2. VF_CHANGE This value has no effect, unless it is ored with VF_GROUP. In this case the cursor will exit the group in the indicated direction.
3. 0 Assigning zero to newval will cause key to treat a field within a group as if it were not part of a group.
4. VF_OFFSCREEN Offscreen data will scroll onscreen from the direction indicated.
5. VF_NOPROT key will move cursor into a field protected from tabbing.

■KEY_XLATE

Allows access to the cursor table. Use this routine to assign key the action preformed by newval. newval may be any of the logical keys listed in the table above. This can often replace a user-supplied key change function.

RETURNS

–1 if some parameter is out of range.
the old value otherwise.

keyset

open a keyset

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare name              char(256) varying;
declare scope             fixed binary(31);
declare status            fixed binary(31);
status = xsm_r_keyset(name, scope);


declare address           bit(0);
declare scope             fixed binary(31);
declare status            fixed binary(31);
status = xsm_d_keyset(address, scope);
```

DESCRIPTION

Use `xsm_d_keyset` and `xsm_r_keyset` to display a keyset. The parameter `name` is the name of the keyset. `scope` must be one of the mnemonics listed in `smsoftk.incl.pl1`. Application programs will normally use scope `KS_APPLIC`. Values for `scope` are defined in `smsoftk.incl.pl1`. For a more detailed explanation of scope see the Key Set chapter of the Author's Guide.

If there is currently a keyset of the specified scope the name of that keyset is compared with the name passed. If they are the same the present routine returns immediately. This means that if you want to "refresh" a keyset with a new copy from disk, you must first close the keyset with a call to `xsm_c_keyset`.

If the call is not successful then the current keyset remains displayed and an error message is posted to the end–user, except where noted otherwise.

The most commonly used variant is `xsm_r_keyset`. You do not need to know where the keyset resides because `xsm_r_keyset` searches for you. It looks first in the memory resident form list, next in any open libraries, then on disk in the directory specified by the argument to `xsm_initcrt`, and finally in the directories specified by `SMPATH`. Keyset files may be mixed freely with screen files in the screen list and in libraries.

You may save processing time by using `xsm_d_keyset` to display a memory–resident keyset. `address` is a pointer to the keyset in memory. Use the utility `bin2pl1` to create program data structures, from disk–based keysets, that you can compile into your application.

To close a keyset use `xsm_c_keyset`.

RETURNS

0 If no error occurred during display of the keyset.
–1 If the format incorrect (not a keyset).
–2 if the keyset cannot be found. No message is posted to the end–user.
–3 If the terminal doesn't support soft keys (or scope out of range).
–4 If there is a read error.
–5 If there is a malloc failure.

kscscope

query current keyset scope

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope             fixed binary(31);
scope = xsm_kscscope();
```

DESCRIPTION

This routine returns the scope of the current keyset or `-1` if no keyset is currently active.

This function can be used to determine whether or not the application keyset (as opposed to the system keyset) is currently displayed.

Values for `scope` are defined in `smsoftk.incl.pl1`.

RETURNS

Current scope, or
-1 if not found.

RELATED FUNCTIONS

```
status = xsm_ksinq(scope, number_keys, number_rows,
         current_row, maximum_len, keyset_name);
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

ksinq

inquire about keyset information

SYNOPSIS

```
%include 'smsoftk.incl.pll';

declare scope             fixed binary(31);
declare number_keys       fixed binary(31);
declare number_rows       fixed binary(31);
declare current_row       fixed binary(31);
declare maximum_len       fixed binary(31);
declare keyset_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_ksinq(scope, number_keys, number_rows,
         current_row, maximum_len, keyset_name);
```

DESCRIPTION

Use this routine to obtain the name, number of rows, number of items within a row, and current row of a keyset currently in memory. You supply the keyset's scope and five addresses to hold the information returned by xsm_skinq. scope must be one of the mnemonics defined in smsoftk.incl.pll.

The function places the number of rows in the keyset in number_row, the number of soft keys per row in number_keys, and the current row number in current_row. The name of the keyset is placed in the pre-allocated buffer keyset_name. The size of keyset_name is specified by maximum_len. If the name of the keyset in longer then keyset_name, then xsm_ksinq fills the buffer to the end without adding a null character, otherwise a null character is added to the end of the string. The null pointer may be used for any or all of the parameters about which you do not desire information.

RETURNS

0 if information is returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.

RELATED FUNCTIONS

```
scope = xsm_kscscope();
size = xsm_skinq(scope, row, softkey, value, display_attribute,
         label1, label2);
```

```
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

ksoff

turn off soft key labels

SYNOPSIS

```
call xsm_ksoff();
```

DESCRIPTION

When a keyset is opened with any of the library routines, the labels are automatically displayed. If you do not wish to display the labels at any point within your application, use `xsm_ksoff` to turn the display off.

If you wish to turn them the label display back on, use `xsm_kson`.

RELATED FUNCTIONS

```
call xsm_kson();
```

kson
turn on soft key labels

SYNOPSIS

```
call xsm_kson();
```

DESCRIPTION

Normally, keyset labels are displayed when a keyset is called up. The only way the display can be turned off is with the library routine, `xsm_ksoff`. Use this routine to turn the label display back on.

RELATED FUNCTIONS

```
call xsm_ksoff();
```

l_close
close a library

SYNOPSIS

```
declare lib_desc          fixed binary(31);
declare status            fixed binary(31);
status = xsm_l_close(lib_desc);
```

DESCRIPTION

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

RELATED FUNCTIONS

```
status = xsm_l_at_cur(lib_desc, screen_name);
status = xsm_l_form(lib_desc, screen_name);
lib_desc = xsm_l_open(lib_name);
status = xsm_l_window(lib_desc, screen_name, start_line,
         start_column);
```

l_open
open a library

SYNOPSIS

```
declare lib_name          char(256) varying;
declare lib_desc          fixed binary(31);
lib_desc = xsm_l_open(lib_name);
```

DESCRIPTION

You must use xsm_l_open to open a library before you use a JPL module, a keyset, or a screen that is stored in the library. Use the utility formlib to create a library. (See the JAM Utilites Guide).

This routine allocates space in which to store information about the library, leaves the library file open, and returns a descriptor identifying the library. The descriptor may subsequently be used by xsm_l_window and related functions, to display screens stored in the library. The library can also be referenced implicitly by xsm_r_window, xsm_r_keyset, and xsm_jplcall, as well as related functions, which search all open libraries.

The library file is sought in all the directories identified by SMPATH and the parameter to xsm_initcrt. If you define the SMFLIBS variable in your setup file as a list of library names xsm_l_open will automatically be called for those libraries. The xsm_r_ routines will then search in the specified libraries.

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.

RELATED FUNCTIONS

```
return_value = xsm_jplcall(jplcall_text);
status = xsm_jplload(module_name_list);
status = xsm_jplpublic(module_name_list);
status = xsm_l_at_cur(lib_desc, screen_name);
status = xsm_l_close(lib_desc);
status = xsm_l_form(lib_desc, screen_name);
status = xsm_l_window(lib_desc, screen_name, start_line,
         start_column);
status = xsm_r_at_cur(screen_name);
status = xsm_r_form(screen_name);
status = xsm_r_keyset(name, scope);
status = xsm_r_window(screen_name, start_line, start_column);
```

last

position the cursor in the last field

SYNOPSIS

```
call xsm_last();
```

DESCRIPTION

Use this function to place the cursor at the first enterable position of the last tab–unprotected field of the current screen. If the last field unprotected from tabbing is right justified, the cursor is placed in the rightmost position of the field. By the same token, if the last unprotected field is left justified, the cursor is placed in the leftmost position of the field.

Unlike `xsm_home`, `xsm_last` will not reposition the cursor if the screen has no unprotected fields.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is called when the JAM logical key EMOH is struck.

RELATED FUNCTIONS

```
call xsm_backtab();
field_number = xsm_home();
call xsm_nl();
call xsm_tab();
```

lclear

erase LDB entries of one scope

SYNOPSIS

```
declare scope             fixed binary(31);
declare status            fixed binary(31);
status = xsm_lclear(scope);
```

DESCRIPTION

This function erases the values stored in the local data block for all names having a scope of the argument `scope`. Legal values for scope are between `1` and `9`. Constant variables having scope `1` *can* be erased.

Refer to the LDB chapter of the Programmer's Guide for a discussion of the scope of LDB entries.

RETURNS

–1 if scope is invalid.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_lreset(file_name, scope);
```

ldb_init
initialize (or reinitialize) the local data block

SYNOPSIS

```
call xsm_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. Data Dictionary entries with a scope of 0 are not loaded into the LDB. There is no LDB prior to the first execution of this function.

Selected parts of the LDB, namely those assigned a certain scope, can be reinitialized using `xsm_lclear` or `xsm_lreset`.

This function is called explicitly in `jmain.pl1` and `jxmain.pl1`. Other functions that affect its behavior, such as `xsm_dicname` and `xsm_ininames`, should be called first.

RELATED FUNCTIONS

```
status = xsm_dicname(dic_name);
status = xsm_ininames(name_list);
status = xsm_lreset(file_name, scope);
```

leave

prepare to leave a **JAM** application temporarily

SYNOPSIS

```
call xsm_leave();
```

DESCRIPTION

At times it may be necessary to leave a **JAM** application temporarily. For example you may need to escape to the command interpreter or to execute some graphics functions. 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).

RELATED FUNCTIONS

```
call xsm_return();
```

length

get the maximum length of a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare field_length       fixed binary(31);
field_length = xsm_length(field_number);
```

DESCRIPTION

This function 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 the JAM Screen Editor, and has no relation to the current contents of the field. Use `xsm_dlength` to get the length of the contents.

RETURNS

Length of the field.
0 if the field is not found.

VARIANTS

```
field_length = xsm_n_length(field_name);
```

RELATED FUNCTIONS

```
data_length = xsm_dlength(field_number);
```

lngval
get the long integer value of a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare value             fixed binary(31);
value = xsm_lngval(field_number);
```

DESCRIPTION

This function returns the contents of `field_number`, converted to a long integer. All non-digit characters are ignored, except for a leading plus or minus sign.

RETURNS

The long value of the field.
0 if the field is not found.

VARIANTS

```
value = xsm_e_lngval(field_name, element);
value = xsm_i_lngval(field_name, occurrence);
value = xsm_n_lngval(field_name);
value = xsm_o_lngval(field_number, occurrence);
```

RELATED FUNCTIONS

```
value = xsm_intval(field_number);
status = xsm_ltofield(field_number, value);
```

lreset
reinitialize LDB entries of one scope

SYNOPSIS

```
declare file_name        char(256) varying;
declare scope            fixed binary(31);
declare status           fixed binary(31);
status = xsm_lreset(file_name, scope);
```

DESCRIPTION

This function sets local data block entries to values read from `file_name`. The `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. This means that `xsm_lreset` 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 whites space 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 `xsm_ininames` to list a number of files for initialization.

RETURNS

–1 if file not found or scope out of range.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_lclear(scope);
```

lstore

copy everything from screen to LDB

SYNOPSIS

```
declare status            fixed binary(31);
status = xsm_lstore();
```

DESCRIPTION

This function copies data from the screen to local data block entries with matching names.

The JAM Executive automatically calls `xsm_lstore` when bringing up a new screen or before closing a window. This function need not be called by application code except under special circumstances.

RETURNS

–3 if sufficient memory is not available.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_allget(respect_flag);
```

ltofield

place a long integer in a field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare value              fixed binary(31);
declare status             fixed binary(31);
status = xsm_ltofield(field_number, 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

```
status = xsm_e_ltofield(field_name, element, value);
status = xsm_i_ltofield(field_name, occurrence, value);
status = xsm_n_ltofield(field_name, value);
status = xsm_o_ltofield(field_number, occurrence, value);
```

RELATED FUNCTIONS

```
status = xsm_itofield(field_number, value);
value = xsm_lngval(field_number);
```

m_flush

flush the message line

SYNOPSIS

```
call xsm_m_flush();
```

DESCRIPTION

This function 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 `xsm_d_msg_line`, without flushing the entire display as `xsm_flush` does.

RELATED FUNCTIONS

```
call xsm_flush();
```

max_occur

get the maximum number of occurrences

SYNOPSIS

```
declare field_number       fixed binary(31);
declare maximum            fixed binary(31);
maximum = xsm_max_occur(field_number);
```

DESCRIPTION

This function returns the maximum number of occurrences that the array can hold as defined in the JAM Screen Editor or by `xsm_sc_max`. If you wish to find out the highest occurrence number of an array that actually contains data, use `xsm_num_occurs`.

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 occurrences in a scrollable array.

VARIANTS

```
maximum = xsm_n_max_occur(field_name);
```

RELATED FUNCTIONS

```
number = xsm_num_occurs(field_number);
```

mnutogl

switch between menu mode and data entry mode on a dual–purpose screen

SYNOPSIS

```
declare screen-mode        fixed binary(31);
declare old_mode           fixed binary(31);
old_mode = xsm_mnutogl(screen_mode);
```

DESCRIPTION

JAM supports the use of a single screen as both a menu and a data entry screen, but the screen must be in one or the other "mode" at any given moment. This function can be used to change the mode of the screen and to test which mode the screen is in currently. The `mode` argument may have one of four values as defined in `smdefs.incl.pl1`:

Value	*Meaning*
`IN_AUTO`	No action (generally used just to test the return value).
`IN_DATA`	Change the screen to data entry mode.
`IN_MENU`	Change the screen to menu mode.
`IN_TOGL`	Toggle the screen from one mode to the other (akin to the `MTGL` logical key).

This function is similar to the built–in control function `jm_mnutogl`.

RETURNS

The mode that the screen was in before the function was called (`IN_DATA` or `IN_MENU`.)
–1 if the mode specification is invalid.

msg
display a message at a given column on the status line

SYNOPSIS

```
declare column            fixed binary(31);
declare disp_length       fixed binary(31);
declare text              char(256) varying;
call xsm_msg(column, disp_length, text);
```

DESCRIPTION

The message is merged with the current contents of the status line, and displayed beginning at `column`. `disp_length` gives the number of characters to display.

On terminals with onscreen attributes, the column position may need to be adjusted to allow for attributes embedded in the status line. Refer to `xsm_d_msg_line` for an explanation of how to embed attributes and function key names in a status line message.

This function is called by the function that updates the cursor position display (see `xsm_c_vis`).

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
```

msg_get
find a message given its number

SYNOPSIS

```
%include 'smerror.incl.pl1';

declare buffer             char(256) varying;
buffer = xsm_msg_get(number);
```

DESCRIPTION

The messages used by **JAM** library routines are stored in binary message files, which are created from text files using the **JAM** utility, `msg2bin`. Use `xsm_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.incl.pl1`.

RETURNS

The desired message, if found
otherwise, the message class and number, as a string

RELATED FUNCTIONS

```
buffer = xsm_msgfind(number);
status = xsm_msgread(code, class, mode, arg);
```

msgfind
find a message given its number

SYNOPSIS

```
%include 'smerror.incl.pl1';

declare buffer            char(256) varying;
declare number            fixed binary(31);
buffer = xsm_msgfind(number);
```

DESCRIPTION

This function takes the `number` of a Screen Manager message, and returns the message string. It is identical to `xsm_msg_get`, except that it returns zero if the message number is not found.

Screen Manager message numbers are defined in `smerror.incl.pl1`.

RETURNS

The message
0 if the message number is out of range

RELATED FUNCTIONS

```
buffer = xsm_msg_get(number);
status = xsm_msgread(code, class, mode, arg);
```

msgread
read message file into memory

SYNOPSIS

```
%include 'smerror.incl.pl1';

declare code              char(256) varying;
declare class             fixed binary(31);
declare mode              fixed binary(31);
declare arg               char(256) varying;
declare status            fixed binary(31);
status = xsm_msgread(code, class, mode, arg);
```

DESCRIPTION

Reads a single set of messages from a binary message file into memory, after which they can be accessed using `xsm_msg_get` and `xsm_msgfind`. The `code` argument selects a single message class from a file that may contain several classes:

Code	*Class*	*Message Type*
SM	SM_MSGS	Screen Manager
FM	FM_MSGS	Screen Editor
JM	JM_MSGS	JAM run-time
JX	JX_MSGS	Data Dictionary & Control Strings
UT	UT_MSG	Utilities
(blank)		Undesignated user

`class` identifies a class of messages. Classes 0–7 are reserved for user messages, and several classes are reserved to JAM; see `smerror.incl.pl1`. 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 composed from the following list. The first five indicate where to get the message file; at least one of these must be supplied. The latter 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 setup variable SMMSGS.
`MSG_FILENAME`	Use the file named by `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 `xsm_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 `xsm_msg_get` and `xsm_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 and the environment variable was undefined.
–2 if the mode was MSG_ENVIRON or MSG_FILENAME and the message file could not be read from disk; other negative values if the message file was bad or insufficient memory was available.

RELATED FUNCTIONS

```
buffer = xsm_msg_get(number);
buffer = xsm_msgfind(number);
```

mwindow

display a status message in a window

SYNOPSIS

```
declare text              char(256) varying;
declare line              fixed binary(31);
declare column            fixed binary(31);
declare status            fixed binary(31);
status = xsm_mwindow(text, line, 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 `0`. 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.

Due to the delayed write feature in JAM, you should call `xsm_flush` to cause the screen to be updated and the message to be displayed, unless you call `xsm_input` directly after the call to `xsm_mwindow`. `xsm_close_window` may be used to close a window called with `xsm_mwindow`.

All the percent escapes for status messages, except %M and %W, are effective. Refer to `xsm_err_reset` for a list and full description. If either `line` or `column` is negative, the window will be displayed according to the rules given for `xsm_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.

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
```

n_

variants that take a field name only

SYNOPSIS

```
declare field_name        char(256) varying;
call xsm_n_...(field_name, ...);
```

DESCRIPTION

The n_ functions access a field by means of the field/group name. For a complete description of individual functions, look under the related function without n_ in its name. For example, `xsm_n_amt_format` is described under `xsm_amt_format`. If the named field/group is not on the screen, these functions will attempt to access a similarly named entry in the local data block.

name

obtain field name given field number

SYNOPSIS

```
declare buffer            char(256) varying;
declare field_number      fixed binary(31);
buffer = xsm_name(field_number);
```

DESCRIPTION

Given a field number, `xsm_name` returns a buffer that contains the field name referenced by `field_number`.

RETURNS

The name of the field referenced, if found.
0 otherwise.

nl

position cursor to the first unprotected field beyond the current line

SYNOPSIS

```
call xsm_nl();
```

DESCRIPTION

This function moves the cursor to the next occurrence of an array, scrolling if necessary. Unlike the down-arrow, it will allocate an empty scrolling occurrence 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 the current field, the cursor is positioned to the first unprotected field of the screen.

If the screen 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 screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

This function is ordinarily bound to the RETURN key.

RELATED FUNCTIONS

```
call xsm_backtab();
field_number = xsm_home();
call xsm_last();
call xsm_tab();
```

novalbit

forcibly invalidate a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare status            fixed binary(31);
status = xsm_novalbit(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 `xsm_putfield`.

RETURNS

–1 if the field is not found.
0 otherwise.

VARIANTS

```
status = xsm_e_novalbit(field_name, element);
status = xsm_i_novalbit(field_name, occurrence);
status = xsm_n_novalbit(field_name);
status = xsm_o_novalbit(field_number, occurrence);
```

RELATED FUNCTIONS

```
status = xsm_fval(field_number);
status = xsm_s_val();
```

null

test if field is null

SYNOPSIS

```
declare field_number       fixed binary(31);
declare status             fixed binary(31);
status = xsm_null(field_number);
```

DESCRIPTION

Use `xsm_null` to test a field to see whether it has both the null edit and contains the null character string that has been assigned to that field. See null edits in the Author's Guide.

RETURNS

1 If the field has the null edit and contains the appropriate null character string.
-1 if the field does not exist.
0 otherwise.

VARIANTS

```
status = xsm_e_null(field_name, element);
status = xsm_i_null(field_name, occurrence);
status = xsm_n_null(field_name);
status = xsm_o_null(field_number, occurrence);
```

num_occurs

find the highest numbered occurrence containing data

SYNOPSIS

```
declare field_number      fixed binary(31);
declare number            fixed binary(31);
number = xsm_num_occurs(field_number);
```

DESCRIPTION

This function returns the highest occurrence number of the array specified by `field_number` that actually contains data. The field number may be that of any field with the array.

Most of the time the highest numbered occurrence containing data will be the same as the number of occurrences actually containing data. However, it is possible to have blank occurrences preceding occurrences containing data.

This count is different from the maximum capacity of an array, which you can retrieve with `xsm_max_occur`.

RETURNS

The highest numbered occurrence containing data.
0 if there is no data in the field.
–1 if the field is not found.

VARIANTS

```
number = xsm_n_num_occurs(field_name);
```

o_

variants that take a field number and occurrence number

SYNOPSIS

```
declare field_number       fixed binary(31);
declare occurrence         fixed binary(31);
call xsm_o_...(field_number, occurrence, ...);
```

DESCRIPTION

The o_ functions refer to data by field number and occurrence number. An occurrence is a slot within an array of fields in which data may be stored. Occurrences may be either on or off-screen. Since JAM treats an individual field as an array with one field, even a single non-scrolling field is considered to have one occurrence. The JAM library contains routines that allow you to manipulate individual occurrences during run-time.

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, `xsm_o_amt_format` is described under `xsm_amt_format`.

occur_no

get the current occurrence number

SYNOPSIS

```
declare occurrence        fixed binary(31);
occurrence = xsm_occur_no();
```

DESCRIPTION

This function returns the occurrence number of the field beneath the cursor. If the field is an element of a non-scrollable array, the occurrence number is the same as the field's element number. Likewise, the occurrence number of a single non-scrolling field is `1`.

RETURNS

0 if the cursor is not in a field.
Otherwise, the occurrence number.

RELATED FUNCTIONS

```
field_number = xsm_getcurno();
```

off_gofield

move the cursor into a field, offset from the left

SYNOPSIS

```
declare field_number       fixed binary(31);
declare offset             fixed binary(31);
declare status             fixed binary(31);
status = xsm_off_gofield(field_number, offset);
```

DESCRIPTION

This function 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

```
status = xsm_e_off_gofield(field_name, element, offset);
status = xsm_i_off_gofield(field_name, occurrence, offset);
status = xsm_n_off_gofield(field_name, offset);
status = xsm_o_off_gofield(field_number, occurrence, offset);
```

RELATED FUNCTIONS

```
offset = xsm_disp_off();
status = xsm_gofield(field_number);
offset = xsm_sh_off();
```

option

set a Screen Manager option

SYNOPSIS

```
declare option          fixed binary(31);
declare newval          fixed binary(31);
declare oldval          fixed binary(31);
oldval = xsm_option(option, newval);
```

DESCRIPTION

Use `xsm_option` to alter during run–time the default Screen Manager options defined in `smsetup.incl.pl1`. Possible options include, error window attributes, delayed write options, cursor display and zoom options. See the "Setup File" section in the *Configuration Guide* for a list of options and possible values. Use `xsm_keyoption` to alter the behavior of cursor control keys.

If you wish to simply inquire as to an option's current value, use the value NOCHANGE (defined in `smsetup.incl.pl1`) for `newval`.

This function replaces the following version 4.0 functions: `xsm_ch_emsgatt`, `xsm_ch_form_atts`, `xsm_ch_qmsgatt`, `xsm_ch_umsgatt`, `xsm_dw_options`, `xsm_er_options`, `xsm_fcase`, `xsm_fextension`, `xsm_ind_set`, `xsm_mp_options`, `xsm_mp_string`, `xsm_ok_options`, `xsm_stextatt`, and `xsm_zm_options`. They are included in your version 5.0 library only for backward compatibility. We strongly recommend that you do not use them in-the-future.

RETURNS

The old value for the specified option.
–1 if the option is out of range.

RELATED FUNCTIONS

```
oldval = xsm_keyoption(key, mode, newval);
```

oshift

shift a field by a given amount

SYNOPSIS

```
declare field_number      fixed binary(31);
declare offset            fixed binary(31);
declare return_value      fixed binary(31);
return_value = xsm_oshift(field_number, offset);
```

DESCRIPTION

This function 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 with less shifting.

RETURNS

The number of positions actually shifted.
0 if the field is not found or is not shifting.

VARIANTS

```
return_value = xsm_n_oshift(field_name, offset);
```

pinquire

obtain value of a global strings

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare buffer             char(256) varying;
declare which              fixed binary(31);
buffer = xsm_pinquire(which);
```

DESCRIPTION

This function is used to obtain the current value of a global pointer variable. The mnemonics for `which` are defined in `smglobs.incl.pl1`. If you wish to modify a global string use `xsm_pset`.

Pointer values for `which` are defined in `smglobs.incl.pl1`. They are:

Mnemonic	*Meaning*
P_YES	The Y character for YES/NO field. This is returned as a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
P_NO	The N character for YES/NO field. This is returned as a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
P_DECIMAL	This is returned as a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.
P_FLDPTRS	Pointer to an array of field structures. The implementation of these structures is very release dependent.
P_TERM	Returns the name JAM uses as the terminal identifier or the null string if not found.
P_SPMASK	Pointer to an memory-resident full size form containing all blanks.

Mnemonic	*Meaning*
`P_USER`	Pointer to developer-specified region of memory. This pointer is not set by JAM; it is set and maintained, if desired, by the application.
`SP_NAME`	Name of the active screen.
`SP_STATLINE`	Text of current status line.
`SP_STATATTR`	Attributes of current status line (pointer to array of unsigned short integers).
`P_DICNAME`	Name of data dictionary file.
`V_`	Any of the "V_" mnemonics defined in `smvideo.incl.pl1` may be passed to obtain various video related information.

In general, the objects pointed to by the pointers returned by `xsm_pinquire` have limited duration and should be used or copied quickly (except for P_USER, which is maintained by the application). The P_ pointers point to the actual objects within JAM. The SP_ pointers point to copies of the objects. Since the characteristics of these objects are implementation dependent, they may change in future releases of JAM. In no case (except P_USER) should the objects be modified directly through the pointers returned by `xsm_pinquire`. Use `xsm_pset` to modify selected objects).

RETURNS

If the argument corresponds to a global pointer variable, the value of that variable is returned.
0 otherwise.

RELATED FUNCTIONS

```
value = xsm_finquire(field_number, which);
value = xsm_gp_inquire(group_name, which);
value = xsm_iset(which, newval);
buffer = xsm_pset(which, newval);
```

protect

protect an array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare mask               fixed binary(31);
declare status             fixed binary(31);

status = xsm_aprotect(field_number, mask);
status = xsm_aunprotect(field_number, mask);
status = xsm_protect(field_number);
status = xsm_unprotect(field_number);
status = xsm_lprotect(field_number, mask);
status = xsm_lunprotect(field_number, mask);
```

DESCRIPTION

There are four types of protection associated with fields and arrays, any combination of which may be assigned: data entry, tabbing into, clearing, and validation. `xsm_protect` and `xsm_unprotect` always set and clear all four types of protection. The remaining protection functions set and clear any combination of protection, as specified by mask. The mnemonics for mask are defined in `smdefs.incl.pl1` and are listed below. Combinations may be specified by oring mnemonics together.

Mnemonic for mask	*Meaning*
EPROTECT	protect from data entry
TPROTECT	protect from tabbing into and from entering via any other key
CPROTECT	protect from clearing
VPROTECT	protect from validation
ALLPROTECT	protect from all of the above

Protection is associated an individual field (i.e. an element), and with an array as a whole. Therefore, all offscreen array occurrences always share the same level of protection, while the onscreen occurrences have the levels of protection (possibly all different) associated with their host fields (i.e. elements). Since protection is associated with individual fields, and not with individual occurrences, deleting an occurrence with `xsm_doccur` will not scroll up the protection with the occurrences.

`xsm_protect`, `xsm_unprotect`, `xsm_lprotect`, and `xsm_lunprotect` set and clear protection for individual fields. `xsm_aprotect` and `xsm_aunprotect` set and clear protection for all of the fields of an array, and for the array as a whole (the `field_number` may specify any field in the array). For example, unprotecting an array with `xsm_aunprotect` will undo protection done by `xsm_lprotect`. A subsequent call to `xsm_lprotect` will re-protect the specified field of the array, but can never affect the offscreen occurrences of the array.

Caution: It is generally safer to protect and unprotect arrays with `xsm_aprotect` and `xsm_aunprotect`, rather than with the field-oriented protection functions.

RETURNS

–1 if the field does not exist;
0 otherwise.

VARIANTS

```
status = xsm_n_protect(field_name);
status = xsm_e_protect(field_name, element);
status = xsm_n_unprotect(field_name);
status = xsm_e_unprotect(field_name, element);
status = xsm_n_lprotect(field_name, mask);
status = xsm_e_lprotect(field_name, element, mask);
status = xsm_n_lunprotect(field_name, mask);
status = xsm_e_lunprotect(field_name, element, mask);
status = xsm_n_aprotect(field_name, mask);
status = xsm_n_aunprotect(field_name, mask);
```

pset

Modify value of global strings

SYNOPSIS

```
%include 'smglobs.incl.pl1';

declare buffer              char(256) varying;
declare which               fixed binary(31);
declare newval              char(256) varying;
buffer = xsm_pset(which, newval);
```

DESCRIPTION

This function is used to modify the contents of a global string. The string you wish to change is specified by `which`. The value that you wish to change the variable to is specified by `newval`. If you wish only to get the value of a global string use `xsm_pinquire`.

The following values for `which`, defined in `smglobs.incl.pl1`, are available:

Mnemonic	*Meaning*
P_YES	The Y character for YES/NO field. This is specified by a three character string. The first character is the lowercase yes value, the second character is the uppercase yes value, and the third character is the null terminator.
P_NO	The N character for YES/NO field. This is specified by a three character string. The first character is the lowercase no value, the second character is the uppercase no value, and the third character is the null terminator.
P_DECIMAL	This is specified by a three character string. The first character is the user's decimal point marker, the second character is the operating system's decimal point marker, and the third character is the null terminator.

RETURNS

If `which` is one of the above, the old contents of the corresponding array are returned. 0 otherwise.

RELATED FUNCTIONS

```
value = xsm_iset(which, newval);
```

```
buffer = xsm_pinquire(which);
```

putfield

put a string into a field

SYNOPSIS

```
declare field_number      fixed binary(31);
declare data              char(256) varying;
declare status            fixed binary(31);
status = xsm_putfield(field_number, data);
```

DESCRIPTION

The string `data` is moved into the field specified by `field_number`. Strings that are too long will be truncated without warning, while strings shorter than the destination field are blank filled (to the left if the field is right justified, otherwise to the right). If `data` is a null string, then the field is cleared. This causes date and time fields that take system values to be 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. `xsm_n_putfield` and `xsm_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.

In variants that take `name` as an argument, `name` can be either the name of a field or a group. In the case of a group, the functions `xsm_select` and `xsm_deselect` should be used to change the group's value.

Notice that the order of arguments to this function is different from that of arguments to the related function `xsm_getfield`.

RETURNS

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

VARIANTS

```
status = xsm_e_putfield(name, element, data);
status = xsm_i_putfield(name, occurrence, data);
status = xsm_n_putfield(name, data);
status = xsm_o_putfield(field_number, occurrence, data);
```

RELATED FUNCTIONS

```
status = xsm_deselect(group_name, group_occurrence);
```

```
length = xsm_getfield(buffer, field_number);
status = xsm_select(group_name, group_occurrence);
```

putjctrl

associate a control string with a key

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key               fixed binary(31);
declare control_string    char(256) varying;
declare default           fixed binary(31);
declare status            fixed binary(31);
status = xsm_putjctrl(key, control_string, 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 `SMINICTRL` setup variables. See the section on setup in the Configuration Guide for further information.

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 `default` is zero, the control string will be installed in the current screen, and will disappear when you exit the screen; otherwise, it will go into the system-wide default table. If `control_string` is empty, the existing control string, if any, will be deleted. If both screen and default control strings exist for a given key, deleting the control string for the screen will put the default control string into effect.

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 the form or window stack for function key definitions; only the current screen and the default table are consulted. Mnemonics for `key` are in `smkeys.incl.pl1`. The syntax for control strings is defined in the Author's Guide.

RETURNS

–5 if insufficient memory is available; 0 otherwise.

pwrap

put text to a wordwrap field

SYNOPSIS

```
declare field_number       fixed binary(31);
declare text               char(256) varying;
declare status             fixed binary(31);
status = xsm_pwrap(field_number, text);
```

DESCRIPTION

This function copies `text` to a wordwrap field specified by `field_number`. Wraps occur at the end of words. The last character of every line is a space. If a word is longer than one less than the length of the field, the word is broken one character short of the end of the field, a space is appended, and the remainder of the word wraps to the next line.

The variant `xsm_o_pwrap` copies the text into an array beginning at the specified occurrence.

Warning: If you attempt to copy data that is too large for the wordwrap field to hold, `xsm_pwrap` will truncate the excess text.

RETURNS

–1 if the field number is invalid.
–2 if the text was truncated because it was too long for the field.
0 otherwise.

VARIANTS

```
status = xsm_o_pwrap(field_number, occurrence, text);
```

RELATED FUNCTIONS

```
length = xsm_gwrap(buffer, field_number, buffer_length);
```

query_msg

display a question, and return a yes or no answer

SYNOPSIS

```
declare message           char(256) varying;
declare reply             fixed binary(31);
reply = xsm_query_msg(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 (or the XMIT key), and a no key is the first letter of the `SM_NO` entry (or the EXIT key); case is ignored. At that point, this function returns the lower case letter as defined in the message file to its caller.

All keys other than yes and no keys are ignored.

RETURNS

Lower-case ASCII 'y' or 'n', according to the response.

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
status = xsm_is_no(field_number);
status = xsm_is_yes(field_number);
```

qui_msg

display a message preceded by a constant tag, and reset the message line

SYNOPSIS

```
declare message           char(256) varying;
call xsm_qui_msg(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 the description of setup in the Configuration Guide 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. If the cursor position display has been turned on (see `xsm_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.

This function is identical to `xsm_quiet_err`, except that it does not turn the cursor on. It is similar to `xsm_emsg`, which does not prepend a tag.

Several *percent escapes* provide control over the content and presentation of status messages. See `xsm_emsg` for details.

RELATED FUNCTIONS

```
call xsm_emsg(message);
call xsm_err_reset(message);
oldval = xsm_option(option, newval);
call xsm_quiet_err(message);
```

quiet_err

display error message preceded by a constant tag, and reset the status line

SYNOPSIS

```
declare message           char(256) varying;
call xsm_quiet_err(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). This function is identical to `xsm_qui_msg`, except that it turns the cursor on. It is similar to `xsm_err_reset`, which does not prepend a tag. Refer to `xsm_emsg` for an explanation of how to change display attributes and insert function key names within a message.

RELATED FUNCTIONS

```
call xsm_emsg(message);
call xsm_err_reset(message);
oldval = xsm_option(option, newval);
call xsm_qui_msg(message);
```

rd_part

read part of a data structure to the current screen

SYNOPSIS

```
declare screen_struct       bit(0);
declare first_field         fixed binary(31);
declare last_field          fixed binary(31);
call xsm_rd_part(screen_struct, first_field, last_field);
```

DESCRIPTION

This function copies data from a structure to all fields between `first_field` and `last_field` within the current screen, converting individual members as appropriate. An array and its scrolling occurrences will be copied only if the *first* element falls between `first_field` and `last_field`. This routine is commonly used with `xsm_wrt_part`, which writes part of the screen to a structure. If you wish to read information into the entire screen, use `xsm_rdstruct`. To read information into a data dictionary record, use `xsm_rrecord`. Use `xsm_putfield` to write a string to an individual field.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data will not be written into the field. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. You can ignore the items that represent fields which do not fall within the bounds of the specifed fields. However, the structure definition must contain all of the fields on the screen. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The arguments that represent the range of fields to be copied, `first_field` and `last_field` are passed as field numbers.

The structure may be initialized with `xsm_wrt_part` or with data from elsewhere. Structure members within the specified range which will not be initialized prior to calling `xsm_rd_part` must be zeroed–out or you risk crashing your application when garbage is read into the screen.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rd_struct(screen_struct, byte_count);
call xsm_rrecord(structure_ptr, record_name, byte_count);
call xsm_wrt_part(screen_struct, first_field, last_field);
```

rdstruct

read data from a structure to the screen

SYNOPSIS

```
declare screen_struct      bit(0);
declare byte_count         fixed binary(31);
call xsm_rd_struct(screen_struct, byte_count);
```

DESCRIPTION

This function copies data from a structure to the current screen, converting individual members as appropriate. It is commonly used with `xsm_wrtstruct`, which writes data from fields on the current screen to a structure. If you wish to read information into a group of consecutively numbered fields, use `xsm_rd_part`. To read information from a data dictionary record, use `xsm_rrecord`. Use `xsm_putfield` to write a string to an individual field.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specifed in the Screen Editor. If you specify the type `omit`, data will not be written into the field. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The argument `byte_count` is an integer variable. `xsm_rdstruct` will store in `byte_count` the number of bytes copied from the structure.

The structure may be initialized with `xsm_wrtstruct` or with data from elsewhere. Members within the structure that will not be initialized prior to calling `xsm_rdstruct` must be zeroed-out or you risk crashing your application when garbage is read into the screen.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
```

```
call xsm_rd_part(screen_struct, first_field, last_field);
call xsm_rrecord(structure_ptr, record_name, byte_count);
call xsm_wrtstruct(screen_struct, byte_count);
```

rescreen

refresh the data displayed on the screen

SYNOPSIS

```
call xsm_rescreen();
```

DESCRIPTION

This function repaints the entire display from JAM's internal screen and attribute buffers. Anything written to the screen by means other than JAM library functions will be erased. This function is normally bound to the RESCREEN key and executed automatically within `xsm_getkey`.

You may need to use this function after doing screen I/O with the flag `xsm_do_not_display` turned on, or after escaping from an JAM application to another program (see `xsm_leave`). If all you want is to force writes to the display, use `xsm_flush`.

RELATED FUNCTIONS

```
call xsm_flush();
call xsm_return();
```

resetcrt

reset the terminal to operating system default state

SYNOPSIS

```
call xsm_resetcrt();
call xsm_jresetcrt();
call xsm_jxresetcrt();
```

DESCRIPTION

The function `xsm_resetcrt` is generally used only when you are writing your own Executive. It resets terminal characteristics to the operating system's normal state. Be sure to call `xsm_resetcrt` be called when leaving the Screen Manager environment (before program exit).

All the memory associated with the display and open screens is freed However, the buffers holding the message file, key translation file, etc. are not released. A subsequent call to `xsm_initcrt` will find them in place. Then `xsm_resetcrt` clears the screen and turns on the cursor, transmits the `RESET` sequence defined in the video file, and resets the operating system channel.

The JAM Executive calls `xsm_resetcrt` via `xsm_jresetcrt` (or via `xsm_jxresetcrt` in the case of an authoring executable) automatically as part of its exit processing. It should not be called by application programs except in case of abnormal termination.

RELATED FUNCTIONS

```
call xsm_cancel();
call xsm_leave();
```

resize
notify **JAM** of a change in the display size

SYNOPSIS

```
declare rows               fixed binary(31);
declare columns            fixed binary(31);
declare status             fixed binary(31);
status = xsm_resize(rows,  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. Applications can be run in different sized windows with each application setting 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).

If the specified 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.

return
prepare for return to JAM application

SYNOPSIS

```
call xsm_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 `xsm_leave` was called. Use `xsm_rescreen` to accomplish that, if desired.

RELATED FUNCTIONS

```
call xsm_leave();
call xsm_resetcrt();
```

rmformlist

empty the memory–resident form list

SYNOPSIS

```
call xsm_rmformlist;
```

DESCRIPTION

This function erases the memory–resident form list established by `xsm_formlist`, and releases the memory used to hold it. It does not release any of the memory–resident JPL modules, key sets, or screens themselves. Calling this function will prevent `xsm_r_window`, `xsm_r_keyset`, `xsm_jplcall`, and related functions from finding memory–resident objects.

RELATED FUNCTIONS

```
status = xsm_formlist(name, address);
```

rrecord

read data from a structure to a data dictionary record

SYNOPSIS

```
declare structure_ptr       bit(0);
declare record_name         char(256) varying;
declare byte_count          fixed binary(31);
call xsm_rrecord(structure_ptr, record_name, byte_count);
```

DESCRIPTION

This function reads data from a PL/1 structure into fields on the current screen that are part of a common data dictionary record. If a field is not on the current screen then the data is written to the LDB. This routine is commonly used with `xsm_wrecord`, which writes data from a data dictionary record to a PL/1 structure. If you wish to read data into all of the fields within the current screen, use `xsm_rdstruct`. To copy data to a group of consecutively numbered fields, use `xsm_rd_part`. Use `xsm_putfield` to write a string to an individual field.

A data structure named `record` can be created from the data dictionary file `data.dic` via the `dd2struct` utility as follows:

```
dd2struct -gPL1 data.dic
```

Each structure member is a field within a data dictionary record that is of the type specified in the Screen Editor. Data will be written into the field onscreen even if the `omit` type is specified. See "Data Type" in the Author's Guide and `dd2struct` in the Utilities Guide for further information.

Once created, the declarations may be treated exactly like any other structure declarations. The argument `struct_ptr` is the address of a variable of one of the structure types generated by `dd2struct`. The argument `record_name` is the name of the data dictionary record from which the structure was created.

The argument `byte_count` is a pointer to an integer. Upon return from `xsm_rrecord`, the value contained in the integer will be the number of bytes or characters read from the structure. The value will be 0 if an error occurred.

The structure may be initialized with `xsm_wrecord` or with data from elsewhere. Members within the structure that will not be initialized prior to calling `xsm_rrecord` must be zeroed-out or you risk crashing your application when garbage is read into the screen or the LDB.

Remember, you must update the structure declaration whenever you alter the data dictionary from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rd_part(screen_struct, first_field, last_field);
call xsm_rd_struct(screen_struct, byte_count);
call xsm_wrecord(structure_ptr, record_name, byte_count);
```

rscroll
scroll an array

SYNOPSIS

```
declare field_number      fixed binary(31);
declare req_scroll        fixed binary(31);
declare lines             fixed binary(31);
lines = xsm_rscroll(field_number, req_scroll);
```

DESCRIPTION

This function scrolls an array along with any synchronized arrays by `req_scroll` occurrences. If `req_scroll` is positive, the array scrolls down (towards the bottom of the data); otherwise, it scrolls up.

The function returns the actual amount scrolled. This could be the amount requested, or a smaller value if the requested amount would bring the array past its beginning or end. If 0 is returned it means that the array was at its beginning or end, or an error occurred. Negative numbers indicate scrolling up occurred.

RETURNS

The actual amount scrolled. Positive numbers indicate downward scrolling while negative numbers mean upward scrolling.
0 if no scrolling or error.

VARIANTS

```
lines = xsm_n_rscroll(field_name, req_scroll);
```

RELATED FUNCTIONS

```
status = xsm_ascroll(field_number, occurrence);
status = xsm_t_scroll(field_number);
```

s_val
validate the current screen

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_s_val();
```

DESCRIPTION

This function validates each field and occurrence, whether on or offscreen, that is not protected from validation (VPROTECT). It is called automatically from `xsm_input` when the TRANSMIT key is hit while in data entry mode. `xsm_sval` also validates groups.

When the first element of a scrolling array is encountered, earlier offscreen occurrences are validated first. When the last element of a scrolling array is encountered, later offscreen occurrences are validated immediately after that element.

If synchronized arrays exist, the following occurs. When an offscreen occurrence is validated, the corresponding occurrences from synchronized arrays are validated as well. Synchronized array are validated in order according to their base field number. The offscreen occurrences *preceding* the synchronized arrays are validated before the first onscreen occurrence of the first (lowest base field number) of the synchronized arrays. Similarly, the offscreen occurrences *following* the arrays are validated immediately after the last onscreen occurrence of the last (highest base field number) array.

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

Validation	*Skip if valid*	*Skip if empty*
field validation	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 field with embedded punctuation must contain at least one non–blank non–punctuation character in order to be considered non–empty; otherwise any non blank character makes the field non–empty.

If an occurrence fails validation, the cursor is positioned to it and an error message displayed. If the occurrence was offscreen, its the array is first scrolled to bring it onscreen. This routine returns at the first error; any fields past will not be validated.

RETURNS

–1 if any field fails validation.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_fval(field_number);
```

sc_max

alter the maximum number of occurrences allowed in a scrollable array

SYNOPSIS

```
declare field_number       fixed binary(31);
declare new_max            fixed binary(31);
declare actual_max         fixed binary(31);
actual_max = xsm_sc_max(field_number, new_max);
```

DESCRIPTION

This function changes the maximum number of occurrences allowed in `field_number`, and in all synchronized arrays. The original maximum is set when the screen is created. If the desired new maximum is less than the highest numbered occurrence that contains data, the new maximum will be set to the number of that occurrence (i.e., the value returned by `xsm_num_occurs`). The maximum can decrease only to a value between the highest numbered occurrence containing data and the previous maximum. It can never be less than the number of elements in the array.

RETURNS

The actual new maximum (see above).
0 if the desired maximum is invalid, or if the array is not scrollable.

VARIANTS

```
actual_max = xsm_n_sc_max(field_name, new_max);
```

RELATED FUNCTIONS

```
maximum = xsm_max_occur(field_number);
number = xsm_num_occurs(field_number);
```

sdtime

get formatted system date and time

SYNOPSIS

```
declare buffer          char(256) varying;
declare format          char(256) varying;
buffer = xsm_sdtime(format);
```

DESCRIPTION

This function gets the current date and/or time from the operating system and returns it in the form specified by `format`.

`format` is a string beginning with `y` or `n` followed by any combination of date/time tokens and literal text. `y` indicates a 12–hour clock; `n` (or any other character) indicates a 24–hour clock. This character must be given, even if the format does not include time tokens. The tokens are described in the table below. These tokens are case–sensitive.

Unit	*Description*	*Token*
Year	4 digit (e.g., 1990)	`%4y`
	2 digit (e.g., 90)	`%2y`
Month	1 or 2 digit (1 – 12)	`%m`
	2 digit (01 – 12)	`%0m`
	full name (e.g., January)	`%*m`
	3 character name (e.g., Jan)	`%3m`
Day	1 or 2 digit (1 – 31)	`%d`
	2 digit (01 – 31)	`%0d`
Day of the Week	full name (e.g. Sunday)	`%*d`
	3 character name (e.g., Sun)	`%3d`
Day of the Year	digit (1 – 365)	`%+d`

Unit	*Description*	*Token*
Hour	1 or 2 digit (1 – 12 or 1 – 24)	`%h`
	2 digit (01 –12 or 01 –24)	`%0h`
Minute	1 or 2 digit (1 – 59)	`%M`
	2 digit (01 – 59)	`%0M`
Second	1 or 2 digit (1 – 59)	`%s`
	2 digit (01 – 59)	`%0s`
AM or PM	for use with a 12–hour clock	`%p`
Literal Percent	use `%` as a literal character	`%%`
Ten Default Formats	`SM_0DEF_DTIME`	`%0f`
(from the message file)	`SM_1DEF_DTIME`	`%1f`
	...	...
	`SM_9DEF_DTIME`	`%09f`

At runtime, JAM strips off the first character of `format`. If the character is `y`, it uses a 12–hour clock; else it uses the default 24–hour clock. Next it examines the rest of `format`, replacing any tokens with the appropriate values. All other characters are used literally. Therefore, be sure to put a `y` or an `n` (or perhaps a blank) at the beginning of `format`. If you do not, JAM strips off the first token's percent sign and it treats the rest of the token as literal text.

You may also retrieve a date/time format from a field using `xsm_edit_ptr`.

The text for day and month names, AM and PM, as well as the tokens for the ten default formats, are all stored in the message file. These entries may be modified. See the Configuration Guide for details.

Note: This function replaces Release 4's `xsm_sdate` and `xsm_stime` function.

RETURNS

The current date/time in the specified format.
Empty if `format` is invalid.

RELATED FUNCTIONS

```
  status = xsm_calc(field_number, occurrence, expression);
```

select

select a checklist or radio button occurrence

SYNOPSIS

```
declare group_name        char(256) varying;
declare group_occurrence  fixed binary(31);
declare status            fixed binary(31);
status = xsm_select(group_name, group_occurrence);
```

DESCRIPTION

This function allows you to select a specific occurrence within a checklist or radio button. The group name and occurrence number are used to reference the desired selection.

Use `xsm_deselect` to deselect a checklist occurrence.

Selecting a radio button occurrence automatically causes the currently selected radio button to be deselected, because exactly one occurrence in a radio button group must be selected at all times. See the Author's Guide for a more detailed discussion of groups.

Use `xsm_isselected` to check whether or not a particular radio button or checklist occurrence is currently selected.

RETURNS

–1	arguments do not reference a checklist or radio button occurrence.
0	occurrence not previously selected.
1	occurrence previously selected.

RELATED FUNCTIONS

```
status = xsm_deselect(group_name, group_occurrence);
status = xsm_isselected(group_name, group_occurrence);
```

setbkstat

set background text for status line

SYNOPSIS

```
declare message           char(256) varying;
declare display_attribute fixed binary(31);
call xsm_setbkstat(message, display_attribute);
```

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 `xsm_d_msg_line`, `xsm_err_reset`, `xsm_quiet_err`, or `xsm_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.

Possible values for the `display_attribute` argument are defined in the header file `smdefs.incl.pl1`, as shown in the table below:

Foreground Attributes	*Background Attributes*
BLANK	B_HILIGHT
REVERSE	
UNDERLN	
BLINK	
HILIGHT	
STANDOUT	
DIM	
ACS (alternate character set)	
Foreground Colors	*Background Colors*
BLACK	B_BLACK
BLUE	B_BLUE
GREEN	B_GREEN
CYAN	B_CYAN

Foreground Colors	*Background Colors*
RED	B_RED
MAGENTA	B_MAGENTA
YELLOW	B_YELLOW
WHITE	B_WHITE

Foreground colors may be used alone or ored with one or more highlights, a background mnemonic, and a background highlight. If you do not specify a highlight or a background mnemonic, the attribute defaults to white against a black background. Omitting the foreground mnemonic will cause the attribute to default to black.

`xsm_setstatus` sets the background status to an alternating ready/wait flag; you should turn that feature off before calling this routine.

Refer to `xsm_d_msg_line` for an explanation of how to embed attribute changes and function key names into your message.

RELATED FUNCTIONS

```
call xsm_d_msg_line(message, display_attribute);
call xsm_setstatus(mode);
```

setstatus

turn alternating background status message on or off

SYNOPSIS

```
declare mode                fixed binary(31);
call xsm_setstatus(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 **JAM** is waiting 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 `xsm_err_reset` or a related routine. They will overwrite messages posted with `xsm_d_msg_line` or `xsm_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 `xsm_d_msg_line` for instructions.

RELATED FUNCTIONS

```
call xsm_setbkstat(message, display_attribute);
```

sh_off

determine the cursor location relative to the start of a shifting field

SYNOPSIS

```
declare offset            fixed binary(31);
offset = xsm_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 `xsm_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.

RELATED FUNCTIONS

```
offset = xsm_disp_off();
```

shrink_to_fit

remove trailing empty array elements and shrink screen

SYNOPSIS

```
  call xsm_shrink_to_fit();
```

DESCRIPTION

Use this routine to dynamically downsize the current screen when you don't know how many elements of an array are going to be populated with data at run time. This routine removes all trailing elements in all arrays on screen and then shrinks the screen to a size just large enough to accommodate the displayed data. If no data is placed in the array, the entire array will be removed. Only the currently displayed copy of the screen in memory is altered.

This routine only downsizes the array and screen. It will not enlarge an array or screen that is too small to hold the information, so be sure to create, within the Screen Editor, an array and screen that can hold the largest amount of data that you plan on inserting.

sibling

define the current window as being or not being a sibling window

SYNOPSIS

```
declare should_it_be      fixed binary(31);
call xsm_sibling(should_it_be);
```

DESCRIPTION

Users may switch between the active window and all siblings of that window while they are in viewport mode. Sibling windows must be next to each other on the window stack. When a window is defined as a sibling, then it and the window immediately beneath it on the window stack are considered to be siblings of one another. The user enters viewport mode when either the VWPT (viewport) logical key is pressed or when the application program makes a call to `xsm_winsize`.

Use this function to define whether or not the current window is defined as sibling To change the current sibling status of a window assign `should_it_be` to:

0	No, it is not a sibling window.
1	Yes, it is a sibling window.

To understand how sibling windows work, imagine you have a stack of three windows: `window_top`, `window_middle`, and `window_bottom`. To make `window_top` and `window_middle` siblings of each other, define `window_top` as a sibling window. They are now considered siblings of each other. You can then add a third sibling to the pair, by defining `window_middle` as a sibling window. This results in `window_middle` and `window_bottom` becoming siblings of one another and consequently, `window_top` and `window_bottom` are also siblings of each other. There is no limit to the number of siblings window you may chain together in this fashion, as long as the windows are adjacent to each other on the stack.

If you wish to bring a different window to the top of the stack, use `xsm_wselect`. To get the number of windows currently in the window stack use `xsm_wcount`.

The base form can be a sibling of the windows adjacent to it.

RELATED FUNCTIONS

```
return_value = xsm_wcount();
status = xsm_winsize();
```

```
return_value = xsm_wselect(window_number);
```

size_of_array
get the number of elements

SYNOPSIS

```
declare field_number        fixed binary(31);
declare size                fixed binary(31);
size = xsm_size_of_array(field_number);
```

DESCRIPTION

This function returns the number of elements in the array containing `field_number`. Elements are the onscreen portion of an array. An array always has at least 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

```
size = xsm_n_size_of_array(field_name);
```

RELATED FUNCTIONS

```
maximum = xsm_max_occur(field_number);
```

skinq
obtain soft key information by position

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope             fixed binary(31);
declare row               fixed binary(31);
declare softkey           fixed binary(31);
declare value             fixed binary(31);
declare display_attribute fixed binary(31);
declare label1            char(256) varying;
declare label2            char(256) varying;
declare status            fixed binary(31);
size = xsm_skinq(scope, row, softkey, value, display_attribute,
         label1, label2);
```

DESCRIPTION

Use this routine to obtain the value, attributes, and label of a soft key contained in a keyset currently in memory, given a soft key's position within a keyset.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Mnemonics for `scope` are defined in `smsoftk.incl.pl1`. For a more detailed explanation of scope see the Keyset chapter of the Programmer's Guide.

The logical value of the specified soft key is placed in `value`. This will be a number that corresponds to a mnemonic defined in `smkeys.incl.pl1`. A value of 0 means the key is inactive.

The attributes (color, blinking etc...) of the label will be placed in `display_attribute`. The attribute should be one of the mnemonics listed in `smdefs.incl.pl1`.

The first and second row labels are placed in `label1` and `label2` respectively. You should pre-allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

If you want general information about a keyset, see `xsm_ksinq`. If you want the scope of the current keyset, use `xsm_kscscope`.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvinq`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
scope = xsm_kscscope();
status = xsm_ksinq(scope, number_keys, number_rows,
         current_row, maximum_len, keyset_name);
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

skmark

mark or unmark a soft key label by position

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

declare scope              fixed binary(31);
declare row                fixed binary(31);
declare softkey            fixed binary(31);
declare mark               fixed binary(31);
declare status             fixed binary(31);
status = xsm_skmark(scope, row, softkey, mark);
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The argument `mark` may be any single ASCII character. An asterisk (`*`) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvmark`.

RETURNS

0 if the marking was successful.
–1 if there is no keyset of the specified scope.
–2 if the scope is out of range.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
status = xsm_skvmark(scope, value, occurrence, mark);
```

skset
set characteristics of a soft key by position

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope            fixed binary(31);
declare row              fixed binary(31);
declare softkey          fixed binary(31);
declare value            fixed binary(31);
declare attribute        fixed binary(31);
declare label1           char(256) varying;
declare label2           char(256) varying;
declare status           fixed binary(31);
status = xsm_skset(scope, row, softkey, value, attribute,
         label1, label2);
```

DESCRIPTION

This routine can be used to modify a soft key's scope, value, attribute, or label of any currently open keysets. You may modify one or more of these specifications with each call of `xsm_skset`.

The soft key is referenced by the keyset it belongs to, its row within the keyset, and its position within that row. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The argument `row` is the row number in which the desired `softkey` resides. Rows are counted from top to bottom, beginning with `1`. The argument `softkey` is the position number within `row` of the desired soft key. Positions are numbered left to right, beginning with `1`.

The `value` refers to the logical key name to be assigned to the soft key. Available mnemonics are defined in `smkeys.incl.pl1`. If you do not want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using mnemonics listed in `smdefs.incl.pl1`. If you do not want to change `attribute`, assign it `0`. (Note: If you set both the background and foreground to black, `xsm_skset` will set the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

WARNING: This routine can not be used when the keyset contains a greater number of keys per row than the terminal does. When this occurs JAM automatically breaks the rows to position them correctly on the monitor. This means that you will not be able to reliably reference a particular soft key by its row and position. Instead, use `xsm_skvset`.

RETURNS

0 if no error has occurred.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if the row/soft key is out of range.

RELATED FUNCTIONS

```
status = xsm_skvset(scope, value, occurrence, newval,
         attribute, label1, label2);
```

skvinq
obtain soft key information by value

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare value              fixed binary(31);
declare occurrence         fixed binary(31);
declare attribute          fixed binary(31);
declare label1             char(256) varying;
declare label2             char(256) varying;
declare status             fixed binary(31);
status = xsm_skvinq(scope, value, occurrence, attribute,
         label1, label2);
```

DESCRIPTION

Use this routine to obtain the label text and attributes of a soft key contained in a keyset currently in memory, given the soft key's value. It can be used when the terminal has a different number of keys than the keyset was designed for.

The soft key is referenced by the keyset it belongs to, its value, and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The `value` of the soft key is one of the mnemonic defined in `smkeys.incl.pl1`. The argument `occurrence` specifies which occurrence of a key with the specified value is desired (in case of duplicates).

The attributes (color, blinking etc . . .) of the label will be placed in `attribute`. The value of the attributes correspond to a mnemonic, or some combination of ored mnemonics listed in `smdefs.incl.pl1`.

-- The first and second row labels are placed in `label1` and `label2` respectively. You should pre-allocate at least nine elements for `label1` and `label2` buffers (eight for the label characters and one for the null character).

If you do not desire information about one or more of these parameters you may assign the parameters the null pointer.

For general information about a keyset, see `xsm_ksinq`. If you want the scope of the current keyset, use `xsm_kscscope`.

RETURNS

0 if information has been returned.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
size = xsm_skinq(scope, row, softkey, value, display_attribute,
        label1, label2);
```

skvmark
mark a soft key by value

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope              fixed binary(31);
declare value              fixed binary(31);
declare occurrence         fixed binary(31);
declare mark               fixed binary(31);
declare status             fixed binary(31);
status = xsm_skvmark(scope, value, occurrence, mark);
```

DESCRIPTION

Use this routine to mark or unmark a soft key label in an open keyset. The mark is made in the last position of the first label.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The `value` of the soft key is one of the mnemonic defined in `smkeys.incl.pl1`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to mark all occurrences of `value` assign 0 to `occurrence`.

The argument `mark` may be any single ASCII character. An asterisks (*) is the most commonly used mark. To unmark the key use the space character (' ') for `mark`.

The marking or unmarking of a soft key is often done to indicate a selection on a function key that toggles between two options.

RETURNS

0 if the mark was successful.
–1 if there is no active keyset for the given scope.
–2 for an invalid scope.
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
status = xsm_skmark(scope, row, softkey, mark);
```

skvset

set characteristics of a soft key by value

SYNOPSIS

```
%include 'smsoftk.incl.pl1';

%include 'smkeys.incl.pl1';

declare scope            fixed binary(31);
declare value            fixed binary(31);
declare occurrence       fixed binary(31);
declare newval           fixed binary(31);
declare attribute        fixed binary(31);
declare label1           char(256) varying;
declare label2           char(256) varying;
declare status           fixed binary(31);
status = xsm_skvset(scope, value, occurrence, newval,
         attribute, label1, label2);
```

DESCRIPTION

This routine can be used to modify the scope, value, attribute, or label of a soft key within a currently open keyset. You may modify one or more of these specifications with each call of `xsm_skset`.

The soft key is referenced by the keyset it belongs to, its value and its occurrence within the keyset. Use `scope` to reference a particular keyset. Possible values for `scope` are defined in `smsoftk.incl.pl1`. The `value` of the soft key is one of the mnemonic defined in `smkeys.incl.pl1`. The argument `occurrence` is the nth time that `value` appears in the keyset. If you wish to change all occurrences of `value` assign 0 to `occurrence`.

The value of `newvalue` refers to the logical key name to be assigned to the soft key. Available mnemonics are defined in `smkeys.incl.pl1`. If you do want to change the logical name, assign `-1` to `value`.

The `attribute` (color, blinking, etc.) is specified by using mnemonics listed in `smdefs.incl.pl1`. If you do not want to change `attribute`, assign it 0. (Note: If you set both the background and foreground to black, `xsm_skset` will set the foreground to white, provided that the terminal supports background color.)

The variables `label1` and `label2` are the first and second lines of the labels respectively. If you do not wish to change one of the labels, assign it the null pointer.

RETURNS

0 if no error occurred
–1 if there is no active keyset for the given scope
–2 for an invalid scope
–3 if there is no soft key with the given value/occurrence.

RELATED FUNCTIONS

```
status = xsm_skset(scope, row, softkey, value, attribute,
         label1, label2);
```

strip_amt_ptr

strip amount editing characters from a string

SYNOPSIS

```
declare field_number      fixed binary(31);
declare inbuf             char(256) varying;
declare outbuf            char(256) varying;
outbuf = xsm_strip_amt_ptr(field_number, inbuf);
```

DESCRIPTION

Strips all non–digit characters from the string, except for an optional leading minus sign and decimal point. If `inbuf` is not empty, `field_number` is ignored and the passed string is processed in place.

If `inbuf` is empty, the contents of `field_number` are used.

RETURNS

The stripped text,
0 if `inbuf` is empty and the field number is invalid.

RELATED FUNCTIONS

```
status = xsm_amt_format(field_number, buffer);
value = xsm_dblval(field_number);
```

submenu_close

close the current submenu

SYNOPSIS

```
declare status            fixed binary(31);
status = xsm_submenu_close();
```

DESCRIPTION

Submenus are ordinarily closed before `xsm_input` returns. It may, however, be told to leave them open by using the OK_LEAVEOPEN option, either in the setup file or via `xsm_option`. See the *Configuration Guide* for details. Regardless of how this option is set, submenus are automatically closed whenever the underlying window is closed with `xsm_close_window`.

This function, then, is needed only when all of the following conditions are true.

1. OK_LEAVEOPEN is in use.
2. The submenu is no longer needed.
3. Access is needed to the underlying window.

RETURNS

–1 if there is no submenu currently open.
0 otherwise.

RELATED FUNCTIONS

```
status = xsm_close_window();
```

svscreen

register a list of screens on the save list

SYNOPSIS

```
declare screen_list       char(256) varying;
declare count             fixed binary(31);
declare status            fixed binary(31);
status = xsm_svscreen(screen_list, count);
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. The number of screens to be added is given by `count`. You may add screens to the list anywhere within your code, however the screen is not actually placed in memory until it is closed for the first time. This means that the time saving factor only comes into play in subsequent openings of the screen. Any data entered into a screen will not be saved until the screen is closed.

Screens are removed from the list with `xsm_unsvscreen`. You can check to see if a screen is on the save list with `xsm_issv`. Checking the list prior to calling `xsm_svscreen`, however, is not crucial as any attempt to add a screen that is already on the list will have no effect.

This routine saves processing time at the expense of memory. It is best suited for use with screens that both require large amounts of data to be read in from elsewhere (databases, other files, etc.) and do not allow the user to enter data. For instance, if you have a help screen that needs to be populated by a data base and is going to be called up more then once, you can re-display the screen much more quickly by saving the screen in memory.

RETURNS

0 is returned if no error occurred.
1 is returned if registration failed (out of memory).

RELATED FUNCTIONS

```
status = xsm_issv(screen_name);
call xsm_unsvscreen(screen_list, count);
```

t_scroll

test whether an array can scroll

SYNOPSIS

```
  declare field_number       fixed binary(31);
  declare status             fixed binary(31);
  status = xsm_t_scroll(field_number);
```

DESCRIPTION

This function returns 1 if the array in question is scrollable, and 0 if not. The argument `field_number` may be any field within the array.

RETURNS

1 if the array is scrolling.
0 if it is not scrolling or if no such `field_number`.

RELATED FUNCTIONS

```
  status = xsm_t_shift(field_number);
```

t_shift

test whether field can shift

SYNOPSIS

```
declare field_number      fixed binary(31);
declare status            fixed binary(31);
status = xsm_t_shift(field_number);
```

DESCRIPTION

This function 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 `field_number` is invalid.

RELATED FUNCTIONS

```
status = xsm_t_scroll(field_number);
```

tab
move the cursor to the next unprotected field

SYNOPSIS

```
call xsm_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 from tabbing, 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 tab unprotected field on the screen.

This function doesn't immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `xsm_input`.

RELATED FUNCTIONS

```
call xsm_backtab();
field_number = xsm_home();
call xsm_last();
call xsm_nl();
```

tst_all_mdts

find first modified occurrence

SYNOPSIS

```
declare occurrence          fixed binary(31);
declare field_number        fixed binary(31);
field_number = xsm_tst_all_mdts(occurrence);
```

DESCRIPTION

This function tests the MDT bits of all occurrences of all fields on the current screen, and returns the base field and occurrence numbers of the first occurrence with its MDT set, if there is one. The MDT bit indicates that an occurrence 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 `xsm_bitop`).

This function returns zero if no occurrences have been modified. If one has been modified, it returns the base field number, and stores the occurrence number in `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 `occurrence`.

RELATED FUNCTIONS

```
status = xsm_bitop(field_number, action, bit);
call xsm_cl_all_mdts();
```

uinstall

install an application function

SYNOPSIS

```
declare usage              fixed binary(31);
declare func_name          char(256) varying;
declare func               entry variable;
declare language           fixed binary(31);
declare status             fixed binary(31);
status = xsm_uinstall( usage, func_name, func, language);
```

DESCRIPTION

This function installs an application routine that will be called from JAM library functions. Installation enables JAM to pass control to your code in the proper function context.

The possible values for `usage` are defined in the table below (and in the file: `smdefs.incl.pl1`). See section 2.1.1. for more detailed descriptions of the various function types.

If an application is bound with the *-retain_all* option, then JAM can find the entrypoint `func` from the name. Most functions will install themselves automatically the first time they are called. Functions may also be explicitly installed. `func_name` is the name of the function. Use the operating system subroutine `s$find_entry` to find the entry point, or use the variant `xsm_n_uinstall`, which will find it for you. `language` should be set to 1 when programming in PL/1.

Value for `usage`	*Function type*	*Section – Page*
`UINIT_FUNC`	Initialization	2.2.9. – p. 22
`URESET_FUNC`	Reset	2.2.9. – p. 22
`VPROC_FUNC`	Video processing	2.2.12. – p. 24
`CKDIGIT_FUNC`	Check digit computation	2.2.8. – p. 21
`KEYCHG_FUNC`	Keychange	2.2.4. – p. 17
`INSCRSR_FUNC`	Insert/overwrite toggle	2.2.1. – p. 11
`PLAY_FUNC`	Playback recorded keys	2.2.10. – p. 23

Value for usage	*Function type*	*Section – Page*
RECORD_FUNC	Record keys for playback	2.2.10. – p. 23
AVAIL_FUNC	Check for recorded keys	2.2 10. – p. 23
BLKDRVR_FUNC	Block Driver function	
STAT_FUNC	Status line function	2.2.11. – p. 23
DFLT_FIELD_FUNC	Default Field function	2.2.1. – p. 11
DFLT_SCREEN_FUNC	Default Screen function	2.2.2. – p. 15
DFLT_SCROLL_FUNC	Default Scroll driver	
DFLT_GROUP_FUNC	Default Group function	2.2.5. – p. 18

RETURNS

1 if function was successfully installed.
–1 if malloc failure occurred.

VARIANTS

```
status = xsm_n_uinstall( usage, func_name, language);
```

RELATED FUNCTIONS

```
call xsm_async(func, timeout);
```

ungetkey
push back a translated key on the input

SYNOPSIS

```
%include 'smkeys.incl.pl1';

declare key                fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_ungetkey(key);
```

DESCRIPTION

This function saves the translated key given by `key` so that it will be retrieved by the next call to `xsm_getkey`. Multiple calls are permitted. The key values are pushed onto a stack (LIFO).

When `xsm_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 `xsm_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.

RELATED FUNCTIONS

```
key = xsm_getkey();
```

unsvscreen

remove screens from the save list

SYNOPSIS

```
declare screen_list       char(256) varying;
declare count             fixed binary(31);
call xsm_unsvscreen(screen_list, count);
```

DESCRIPTION

JAM maintains a list of screens that are saved in memory. This function is used to remove screens from the save list. The argument `count` specifies the number of screens to be removed from the save list. See `xsm_svscreen`.

This function can be used at any point within your code. It is not necessary for the screen to be open at the time of the call. Any memory allocated to hold the screen is freed at the time of the call unless the screen is open. The memory associated with an open screen is de-allocated when that screen is closed. If a screen is not on the save list, a call to `xsm_unsvscreen` has no effect.

RELATED FUNCTIONS

```
status = xsm_issv(screen_name);
status = xsm_svscreen(screen_list, count);
```

viewport

modify viewport size and offset

SYNOPSIS

```
declare position_row       fixed binary(31);
declare position_col       fixed binary(31);
declare size_row           fixed binary(31);
declare size_col           fixed binary(31);
declare offset_row         fixed binary(31);
declare offset_col         fixed binary(31);
call xsm_viewport(position_row, position_col, size_row,
         size_col, offset_row, offset_col);
```

DESCRIPTION

This function dynamically sizes the current screen's viewport. A viewport has a maximum size of the screen or physical display – whichever is smaller. Use `size_row` and `size_column` to specify the number of rows and columns, respectively.

You can position the viewport anywhere on the physical display. To do this, think of your physical display as a grid made up of rows and columns that are one character apart. The top left corner of your screen monitor is at position row 0, column 0 . Now use the arguments `position_row` and `position_col` to specify the coordinates of the viewport's position.

Likewise, you can also specify which row and column of the screen will initially appear at top left corner of the viewport. Again starting at row 0, column 0, count from the top left of the screen to get the coordinates for `offset_row` and `offset_col`.

This function performs range checks on all parameters and suitably modifies them if necessary. In particular, be aware that a non-positive value of `size_row` and `size_col` will set the viewport to the maximum size in that dimension.

vinit
initialize video translation tables

SYNOPSIS

```
declare video_address      bit(0);
declare status             fixed binary(31);
status = xsm_vinit(video_address);
```

DESCRIPTION

This routine is called by `xsm_initcrt` as part of the initialization process. It can also be called directly by an application program. `video_address` contains the address of a memory resident video file. Such a file must be created by the `vid2bin` and `bin2c` utilities, then compiled into the application.

RETURNS

0 if initialization is successful.
program exit if video file is invalid or if `video_address` is zero and `SMVIDEO` is undefined.

Note: The variant `xsm_n_vinit` has no return value.

VARIANTS

```
call xsm_n_vinit(video_file);
```

wcount

obtain number of currently open windows

SYNOPSIS

```
declare return_value       fixed binary(31)
return_value = xsm_wcount();
```

DESCRIPTION

This function returns the number of windows currently open. The number is equivalent to the number of windows in the window stack.

To select the screen beneath the current window, subtract 1 from the value returned by `xsm_wcount`, and then use the result as the argument to `xsm_wselect`.

This routine is useful when you are bringing another window to the top of the window stack (making the window active) with `xsm_wselect`.

RETURNS

The number of windows.
0 if the base form is the only open screen.
–1 if there is no current screen.

RELATED FUNCTIONS

```
return_value = xsm_wselect(window_number);
```

wdeselect

restore the formerly active window

SYNOPSIS

```
declare status              fixed binary(31);
status = xsm_wdeselect();
```

DESCRIPTION

This function restores a window to its original position in the window stack, after it has been moved to the top by a call to `xsm_wselect`. Information necessary to perform this task is saved during each call to `xsm_wselect`, but is not stacked. Therefore a call to this routine must follow a call to `xsm_wselect` if it is to properly restore the window to its original position. Note that `xsm_wdeselect` does not have to be called if the window ordering on the stack is acceptable.

RETURNS

–1 if there is no window to restore.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
return_value = xsm_wcount();
return_value = xsm_wselect(window_number);
```

window
display a window at a given position

SYNOPSIS

```
declare screen_name       char(256) varying;
declare start_line        fixed binary(31);
declare start_column      fixed binary(31);
declare status            fixed binary(31);
status = xsm_r_window(screen_name, start_line, start_column);


declare screen_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_r_at_cur(screen_name);


declare screen_address    bit(0);
declare start_line        fixed binary(31);
declare start_column      fixed binary(31);
declare status            fixed binary(31);
status = xsm_d_window(screen_address, start_line,
         start_column);


declare screen_address    bit(0);
declare status            fixed binary(31);
status = xsm_d_at_cur(screen_address);


declare lib_desc          fixed binary(31);
declare screen_name       char(256) varying;
declare start_line        fixed binary(31);
declare start_column      fixed binary(31);
declare status            fixed binary(31);
status = xsm_l_window(lib_desc, screen_name, start_line,
         start_column);


declare lib_desc          fixed binary(31);
declare screen_name       char(256) varying;
declare status            fixed binary(31);
status = xsm_l_at_cur(lib_desc, screen_name);
```

DESCRIPTION

This set of functions is primarily intended to be used by developers who are writing their own executive. To open a window while under the control of the JAM Executive, use a JAM control string or `xsm_jwindow`.

Use `xsm_d_window`, `xsm_l_window`, or `xsm_r_window` to display `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 window is displayed starting at the *second* line of the screen.

Use `xsm_d_at_cur`, `xsm_l_at_cur`, and `xsm_r_at_cur` to display a window at the current cursor position, offset by one line to avoid hiding that line's current display.

Whatever part of the display the new window does not occupy will remain visible. However, only the topmost (active) window and its fields are accessible to keyboard entry and library routines. JAM will not allow the cursor outside the topmost window. If you wish to shuffle windows use `xsm_wselect`.

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. 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.

When you use `xsm_r_window` the named screen is sought first in the memory–resident screen list, and if found there is displayed using `xsm_d_window`. It is next sought in all the open libraries, and if found is displayed using `xsm_l_window`. Next it is sought on disk in the current directory; then under the path supplied to `xsm_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.

You may save processing time by using `xsm_d_window` and `xsm_d_at_cur` to display screens that are memory–resident. Use `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. `xsm_r_window` and `xsm_r_at_cur` can also display memory–resident screens, if they are properly installed using `xsm_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). `screen_address` is the address of the screen in memory.

You may also save processing time by using `xsm_l_window` and `xsm_l_at_cur` to display screens that are in a library. A library is a single file containing many screens (and/or JPL modules and keysets). You can assemble one from individual screen files using the utility `formlib`. Libraries provide a convenient way of distributing a large number of screens with an application, and can improve efficiency by cutting down on the number of paths searched.

The library descriptor, `lib_desc`, is an integer returned by `xsm_l_open`, which you must call before trying to read any screens from a library. Note that `xsm_r_window` and `xsm_r_at_cur` also search any open libraries.

If you want to display a form use `xsm_r_form` or one of its variants. Use `xsm_close_window` to close the window.

RETURNS

0 if no error occurred during display of the screen;
–1 if the screen file's format is incorrect;
–2 if the screen cannot be found;
–3 if the system ran out of memory but the previous screen was restored;
–5 is returned if, after the screen was cleared, the system ran out of memory.
–6 is returned if the library is corrupted.

RELATED FUNCTIONS

```
  status = xsm_close_window();
  status = xsm_r_form(screen_name);
  status = xsm_jwindow(screen_name);
```

winsize

allow end–user to interactively move and resize a window

SYNOPSIS

```
declare status           fixed binary(31);
status = xsm_winsize();
```

DESCRIPTION

Calling `xsm_winsize` has the same effect as if the end–user had just hit the VWPT (viewport) logical key. The viewport status line appears and the user can move, resize and change the offset of the screen as well as move to any sibling windows. When the end–user hits XMIT (transmit) the previous status line is restored. If you wish to resize the viewport yourself, use `xsm_viewport`.

In order for the end–user to able to move from one window to another, the windows must be siblings. Windows are defined as siblings of one another either with `xsm_sibling` or by calling up a window as a sibling with a JAM control string. See the sections on "Viewports and Positioning" and "Control Strings" in the Author's Guide for further information.

RETURNS

–1 if call fails.
0 otherwise.

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
call xsm_viewport(position_row, position_col, size_row,
        size_col, offset_row, offset_col);
```

wrecord

write data from a data dictionary record to a structure

SYNOPSIS

```
declare structure_ptr     bit(0);
declare record_name       char(256) varying;
declare byte_count        fixed binary(31);
call xsm_wrecord(structure_ptr, record_name, byte_count);
```

DESCRIPTION

This function writes data from fields within the current screen that are part of a common data dictionary record to a PL/1 structure. If a field is not on the current screen, then the data is read from the LDB. This routine is commonly used with `xsm_rrecord`, which reads data from a structure to a data dictionary record. If you wish to write data only from the current screen, use `xsm_wrtstruct`. To write data from a group of consecutively numbered fields, use `xsm_wrt_part`. Use `xsm_getfield` to write information from an individual field to a string.

A data structure named `record` can be created from the data dictionary file `data.dic` via the `dd2struct` utility as follows:

```
dd2struct -gPL1 data.dic
```

Each structure member is a field within a data dictionary record that is of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `dd2struct` in the Utilities Guide for further information.

Once created, the declarations may be treated exactly like any other structure declarations. The argument `struct_ptr` is the address of a variable of one of the structure types generated by `dd2struct`. The argument `record_name` is the name of the data dictionary record, from which the structure was created.

The argument `byte_count` is a pointer to an integer. Upon return from `xsm_wrecord`, the value contained in the integer will be the number of bytes or characters written to the structure. It will be 0 if an error occurred.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rrecord(structure_ptr, record_name, byte_count);
```

wrt_part
write part of the screen to a structure

SYNOPSIS

```
declare screen_struct      bit(0);
declare first_field        fixed binary(31);
declare last_field         fixed binary(31);
call xsm_wrt_part(screen_struct, first_field, last_field);
```

DESCRIPTION

This function writes the contents of all fields between `first_field` and `last_field` to a data structure in memory. An array and its scrolling occurrences will be copied only if the *first* element falls between `first_field` and `last_field`. Group selections are not copied. This routine is commonly used with `xsm_rd_part`, which reads part of a structure into the current screen. If you wish to write the contents of all of the fields within the screen use `xsm_wrtstruct`. To write information from a data dictionary record, use `xsm_wrecord`. Use `xsm_getfield` to write information from an individual field to a string.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. You can ignore the members that represent fields that do not fall within the bounds of the specified fields. However, the structure definition must contain all of the fields on screen. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`.

The arguments that represent the range of fields to be copied, `first_field` and `last_field` are passed as field numbers.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
```

```
call xsm_rd_part(screen_struct, first_field, last_field);
call xsm_wrtstruct(screen_struct, byte_count);
```

wrtstruct

write data from the screen to a structure

SYNOPSIS

```
declare screen_struct      bit(0);
declare byte_count         fixed binary(31);
call xsm_wrtstruct(screen_struct, byte_count);
```

DESCRIPTION

This function writes the contents of all of the fields within the current screen to a PL/1 structure. It will not copy group selections. This routine is commonly used with `xsm_rdstruct` which reads data from a structure to all of the fields within the current screen. If you wish to write the contents of a group of consecutively numbered fields into a structure use `xsm_wrt_part`. To write information from a data dictionary record, use `xsm_wrecord`. Use `xsm_getfield` to write the contents of an individual field into a string.

A data structure named `screen` can be created from the screen file `screen.jam` via the `f2struct` utility as follows:

```
f2struct -gPL1 screen.jam
```

Each member of the structure is a field of the type specified in the Screen Editor. See "Data Type" in the Author's Guide and `f2struct` in the Utilities Guide for further information.

Once created, the declaration may be treated exactly like any other structure declaration. The argument `screen_struct` is the address of a variable of the type of structure generated by `f2struct`. If you specify the type `omit`, data will not be written into the field.

The argument `byte_count` is an integer variable. `xsm_wrtstruct` will store there the number of bytes copied to the structure.

Remember, you must update the structure declaration whenever you alter the screen from which it was generated.

RELATED FUNCTIONS

```
status = xsm_putfield(field_number, data);
call xsm_rd_struct(screen_struct, byte_count);
call xsm_wrt_part(screen_struct, first_field, last_field);
```

wselect
activate a window

SYNOPSIS

```
declare window_number      fixed binary(31);
declare return_value       fixed binary(31);
return_value = xsm_wselect(window_number);
```

DESCRIPTION

Although JAM allows you to display multiple windows at one time, only one window may be active. Windows may overlap each other, or may be *tiled* (no overlap). The window at the top of the window stack is the active window, and the only window accessible to library routines and keyboard entry. Use `xsm_wselect` to bring a window to the active position on top of the window stack. If any of the referenced window is hidden by an overlying window, it will be brought to the forefront of the display. In either case, the cursor is placed within the window. JAM will restore the cursor to its position when the screen was most recently de-activated.

The window to be activated is referenced by its number in the window stack. Windows are numbered sequentially, starting from the bottom of the stack. The form underlying all the windows (the base form) is window `0`, the first window displayed is `1` and so forth. Since a screen's number depends on its position on the window stack, calling `xsm_wselect` will alter a window's number as well as it position on the stack.

Alternatively, windows may be referenced by their screen name with the variant `xsm_n_wselect`. If you use this routine, you do not have to worry about keeping track of the non-active window's position on the stack. However, `xsm_n_wselect` will not find windows displayed with `xsm_d_window` or related functions, because they do not record the screen name.

Here are two different ways of using window selection. One way to use this is to select a hidden screen, update it (using `xsm_putfield`) and deselect it (using `xsm_wdeselect`). The portion of the hidden screen that is visible will be updated with the new data. Because of *delayed write* the update will be done when the next keyboard input is sought. The other method 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 multi-page forms, or switch among several windows that *tile* the screen (do not overlap).

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 window was not found or the window was not open.

VARIANTS

```
return_value = xsm_n_wselect(window_name);
```

RELATED FUNCTIONS

```
call xsm_sibling(should_it_be);
return_value = xsm_wcount();
status = xsm_wdeselect();
```

Chapter 13. Library Function Index

This chapter lists all JAM library functions, sorted by name. Function names appear on the left, and the section of the Function Reference Chapter in which the function is described appears on the right.

`status = xsm_1clear_array(field_number);`	**clear_array**
`status = xsm_1protect(field_number, mask);`	**protect**
`status = xsm_1unprotect(field_number, mask);`	**protect**
`status = xsm_a_bitop(array_name, action, bit);`	**bitop**
`xsm_allget(respect_flag);`	**allget**
`status = xsm_amt_format(field_number, buffer);`	**amt_format**
`status = xsm_aprotect(field_number, mask);`	**protect**
`status = xsm_ascroll(field_number, occurrence);`	**ascroll**
`xsm_async(func, timeout);`	**async**
`status = xsm_aunprotect(field_number, mask);`	**protect**
`xsm_backtab();`	**backtab**
`base_number = xsm_base_fldno(field_number);`	**base_fldno**
`xsm_bel();`	**bel**
`status = xsm_bitop(field_number, action, bit);`	**bitop**
`status = xsm_bkrect(start_line, start_column, num_of_lines, number_of_columns, background_colors);`	**bkrect**
`return_value = xsm_blkinit()();`	**blkinit**
`return_value = xsm_blkreset()();`	**blkreset**
`status = xsm_c_keyset(scope);`	**c_keyset**
`xsm_c_off();`	**c_off**
`xsm_c_on();`	**c_on**
`xsm_c_vis(display);`	**c_vis**
`status = xsm_calc(field_number, occurrence, expression);`	**calc**
`xsm_cancel(arg);`	**cancel**
`status = xsm_chg_attr(field_number, display_attribute);`	**chg_attr**
`status = xsm_ckdigit(field_number, field_data, occurrence, modulus, minimum_digits);`	**ckdigit**

```
xsm_cl_all_mdts(); ......................................... cl_all_mdts
xsm_cl_unprot(); ............................................. cl_unprot
status = xsm_clear_array(field_number); ..................... clear_array
status = xsm_close_window()(); ............................ close_window
status = xsm_d_at_cur(screen_address); ........................... window
status = xsm_d_form(screen_address); ............................... form
status = xsm_d_keyset(address, scope); ........................... keyset
xsm_d_msg_line(message, display_attribute); .................. d_msg_line
status = xsm_d_window(screen_address, start_line, start_column); . window
value = xsm_dblval(field_number); ................................ dblval
xsm_dd_able(flag); .............................................. dd_able
status = xsm_deselect(group_name, group_occurrence); ........... deselect
status = xsm_dicname(dic_name); ................................. dicname
offset = xsm_disp_off()(); ..................................... disp_off
data_length = xsm_dlength(field_number); ......................... dlength
xsm_do_region(line, column, length, display_attribute, text); . do_region
status = xsm_dtofield(field_number, value, format); ............ dtofield
xsm_e_...(field_name, element, ...); .................................. e_
status = xsm_e_lprotect(field_name, element, mask); ............. protect
status = xsm_e_lunprotect(field_name, element, mask); .... ...... protect
status =  xsm_e_amt_format(field_name, element, buffer); ..... amt_format
status = xsm_e_bitop(array_name, element, action, bit); ........... bitop
status = xsm_e_chg_attr(field_name, element, display_attribute); chg_attr
value = xsm_e_dblval(field_name, element); ....................... dblval
data_length = xsm_e_dlength(field_name, element); ............... dlength
status = xsm_e_dtofield(field_name, element, value, format); ... dtofield
value = xsm_e_finquire(field_name, element, which); ............ finquire
field_number = xsm_e_fldno(field_name, element); .................. fldno
buffer = xsm_e_fptr(field_name, element); .......................... fptr
buffer = xsm_e_ftog(field_name, element, group_occurrence); ........ ftog
status = xsm_e_fval(array_name, element); .......................... fval
length = xsm_e_getfield(buffer, name, element); ................ getfield
status = xsm_e_gofield(field_name, element); ................... gofield
value = xsm_e_intval(field_name, element); ...................... intval
status = xsm_e_is_no(field_name, element); ....................... is_no
status = xsm_e_is_yes(field_name, element); ..................... is_yes
status = xsm_e_itofield(field_name, element, value); ........... itofield
value = xsm_e_lngval(field_name, element); ...................... lngval
status = xsm_e_ltofield(field_name, element, value); ........... ltofield
status = xsm_e_novalbit(field_name, element); .................. novalbit
status = xsm_e_null(field_name, element); .......................... null
status = xsm_e_off_gofield(field_name, element, offset); .... off_gofield
status = xsm_e_protect(field_name, element); .................... protect
status = xsm_e_putfield(name, element, data); .................. putfield
status = xsm_e_unprotect(field_name, element); .................. protect
buffer = xsm_edit_ptr(field_number, edit_type); ................ edit_ptr
xsm_emsg(message); ................................................ emsg
xsm_err_reset(message); ...................................... err_reset
buffer = xsm_fi_path(file_name); ................................ fi_path
value = xsm_finquire(field_number, which); ..................... finquire
xsm_flush(); ..................................................... flush
status = xsm_formlist(name, address); ......................... formlist
buffer = xsm_fptr(field_number); .................................. fptr
buffer = xsm_ftog(field_number, group_occurrence); ................ ftog
type = xsm_ftype(field_number, precision_ptr); .................... ftype
status = xsm_fval(field_number); .................................. fval
field_number = xsm_getcurno()(); .............................. getcurno
length = xsm_getfield(buffer, field_number); .................. getfield
buffer = xsm_getjctrl(key, default); .......................... getjctrl
key = xsm_getkey()(); ............................................ getkey
status = xsm_gofield(field_number); ............................ gofield
value = xsm_gp_inquire(group_name, which); .................. gp_inquire
length = xsm_gwrap(buffer, field_number, buffer_length); .......... gwrap
status = xsm_hlp_by_name(help_screen); ..................... hlp_by_name
field_number = xsm_home()(); ....................................... home
xsm_i_...(field_name, occurrence, ...); ............................ i_
status = xsm_i_achg(field_name, occurrence, display_attribute); .... achg
status = xsm_i_amt_format(field_name, occurrence, buffer); ... amt_format
status = xsm_i_bitop(array_name, occurrence, action, bit); ........ bitop
value = xsm_i_dblval(field_name, occurrence); .................... dblval
data_length = xsm_i_dlength(field_name, occurrence); ............ dlength
return_value = xsm_i_doccur(field_name, occurrence, count); ...... doccur
status = xsm_i_dtofield(field_name, occurrence, value, format); dtofield
value = xsm_i_finquire(field_name, occurrence, which); ......... finquire
field_number = xsm_i_fldno(field_name, occurrence); ............... fldno
buffer = xsm_i_fptr(field_name, occurrence); ....................... fptr
buffer = xsm_i_ftog(field_name, occurrence, group_occurrence); ..... ftog
status = xsm_i_fval(field_name, occurrence); ....................... fval
length = xsm_i_getfield(buffer, name, occurrence); ............. getfield
status = xsm_i_gofield(field_name, occurrence); ................. gofield
field_number = xsm_i_gtof(group_name, group_occurrence, occurrence); gtof
value = xsm_i_intval(field_name, occurrence); .................... intval
lines_inserted = xsm_i_ioccur(field_name, occurrence, count); .... ioccur
status = xsm_i_is_no(field_name, occurrence); ..................... is_no
status = xsm_i_is_yes(field_name, occurrence); ................... is_yes
status = xsm_i_itofield(field_name, occurrence, value); ........ itofield
value = xsm_i_lngval(field_name, occurrence); .................... lngval
status = xsm_i_ltofield(field_name, occurrence, value); ........ ltofield
status = xsm_i_novalbit(field_name, occurrence); ............... novalbit
status = xsm_i_null(field_name, occurrence); ....................... null
status = xsm_i_off_gofield(field_name, occurrence, offset); . off_gofield
status = xsm_i_putfield(name, occurrence, data); ............... putfield
status = xsm_ininames(name_list); ............................. ininames
xsm_initcrt(path); ............................................. initcrt
key = xsm_input(initial_mode); .................................... input
value = xsm_inquire(which); ...................................... inquire
value = xsm_intval(field_number); ................................ intval
status = xsm_is_no(field_number); ................................. is_no
status = xsm_is_yes(field_number); ............................... is_yes
old_flag = xsm_isabort(flag); ................................... isabort
value = xsm_iset(which, newval); ................................... iset
status = xsm_isselected(group_name, group_occurrence); ....... isselected
status = xsm_issv(screen_name); ..................................... issv
status = xsm_itofield(field_number, value); .................... itofield
status = xsm_jclose()(); ........................................ jclose
status = xsm_jform(screen_name); .................................. jform
xsm_jinitcrt(path); ............................................ initcrt
return_value = xsm_jplcall(jplcall_text); ....................... jplcall
status = xsm_jplload(module_name_list); ......................... jplload
status = xsm_jplpublic(module_name_list); ................ .... jplpublic
status = xsm_jplunload(module_name); .......................... jplunload
xsm_jresetcrt(); .............................................. resetcrt
status = xsm_jtop(screen_name); .................................... jtop
status = xsm_jwindow(screen_name); .............................. jwindow
xsm_jxinitcrt(path); ........................................... initcrt
xsm_jxresetcrt(); ............................................. resetcrt
old_flag = xsm_keyfilter(flag); .............................. keyfilter
status = xsm_keyhit(interval); .................................. keyhit
status = xsm_keyinit(key_address); ............................. keyinit
buffer = xsm_keylabel(key); ................................... keylabel
oldval = xsm_keyoption(key, mode, newval); .................... keyoption
scope = xsm_kscscope()(); ..................................... kscscope
status = xsm_ksinq(scope, number_keys, number_rows, current_row,
           maximum_len, keyset_name); ............................ ksinq
xsm_ksoff(); ...................................................... ksoff
xsm_kson(); ........................................................ kson
status = xsm_l_at_cur(lib_desc, screen_name); ................... window
status = xsm_l_close(lib_desc); ................................ l_close
status = xsm_l_form(lib_desc, screen_name); ....................... form
lib_desc = xsm_l_open(lib_name); ................................ l_open
status = xsm_l_window(lib_desc, screen_name, start_line,
           start_column); ....................................... window
```

`xsm_last();`	**last**
`status = xsm_lclear(scope);`	**lclear**
`xsm_ldb_init();`	**ldb_init**
`xsm_leave();`	**leave**
`field_length = xsm_length(field_number);`	**length**
`value = xsm_lngval(field_number);`	**lngval**
`status = xsm_lreset(file_name, scope);`	**lreset**
`status = xsm_lstore()();`	**lstore**
`status = xsm_ltofield(field_number, value);`	**ltofield**
`xsm_m_flush();`	**flush**
`maximum = xsm_max_occur(field_number);`	**max_occur**
`old_mode = xsm_mnutogl(screen_mode);`	**mnutogl**
`xsm_msg(column, disp_length, text);`	**msg**
`buffer = xsm_msg_get(number);`	**msg_get**
`buffer = xsm_msgfind(number);`	**msgfind**
`status = xsm_msgread(code, class, mode, arg);`	**msgread**
`status = xsm_mwindow(text, line, column);`	**mwindow**
`xsm_n_...(field_name, ...);`	**n_**
`status = xsm_n_1clear_array(field_name);`	**clear_array**
`status = xsm_n_1protect(field_name, mask);`	**protect**
`status = xsm_n_1unprotect(field_name, mask);`	**protect**
`status = xsm_n_amt_format(field_name, buffer);`	**amt_format**
`status = xsm_n_aprotect(field_name, mask);`	**protect**
`status = xsm_n_ascroll(field_name, occurrence);`	**ascroll**
`status = xsm_n_aunprotect(field_name, mask);`	**protect**
`status = xsm_n_bitop(name, action, bit);`	**bitop**
`status = xsm_n_chg_attr(field_name, display_attribute);`	**chg_attr**
`status = xsm_n_clear_array(field_name);`	**clear_**
`value = xsm_n_dblval(field_name);`	**dblval**
`data_length = xsm_n_dlength(field_name);`	**dlength**
`status = xsm_n_dtofield(field_name, value, format);`	**dtofield**
`buffer = xsm_n_edit_ptr(field_name, edit_type);`	**edit_ptr**
`value = xsm_n_finquire(field_name, which);`	**finquire**
`field_number = xsm_n_fldno(field_name);`	**fldno**
`buffer = xsm_n_fptr(field_name);`	**fptr**
`buffer = xsm_n_ftog(field_name, group_occurrence);`	**ftog**
`type = xsm_n_ftype(field_number, precision_ptr);`	**ftype**
`status = xsm_n_fval(field_name);`	**fval**
`length = xsm_n_getfield(buffer, name);`	**getfield**
`status = xsm_n_gofield(field_name);`	**gofield**
`status = xsm_n_gval(group_name);`	**gval**
`value = xsm_n_intval(field_name);`	**intval**
`status = xsm_n_is_no(field_name);`	**is_no**
`status = xsm_n_is_yes(field_name);`	**is_yes**

```
lines_inserted = xsm_o_ioccur(field_number, occurrence, count); .. ioccur
status = xsm_o_is_no(field_number, occurrence); .................. is_no
status = xsm_o_is_yes(field_number, occurrence); ................. is_yes
status = xsm_o_itofield(field_number, occurrence, value); ...... itofield
value = xsm_o_lngval(field_number, occurrence); .................. lngval
status = xsm_o_ltofield(field_number, occurrence, value); ...... ltofield
status = xsm_o_novalbit(field_number, occurrence); ............ novalbit
status = xsm_o_null(field_number, occurrence); ..................... null
status = xsm_o_off_gofield(field_number, occurrence, offset); off_gofield
status = xsm_o_putfield(field_number, occurrence, data); ....... putfield
status = xsm_o_pwrap(field_number, occurrence, text); ............. pwrap
occurrence = xsm_occur_no()(); ................................. occurno
status = xsm_off_gofield(field_number, offset); ............ off_gofield
oldval = xsm_option(option, newval); ............................. option
return_value = xsm_oshift(field_number, offset); ................. oshift
buffer = xsm_pinquire(which); ................................ pinquire
status = xsm_protect(field_number); ............................. protect
buffer = xsm_pset(which, newval); .................................. pset
status = xsm_putfield(field_number, data); ..................... putfield
status = xsm_putjctrl(key, control_string, default); ........... putjctrl
status = xsm_pwrap(field_number, text); ........................... pwrap
reply = xsm_query_msg(message); ............................. query_msg
xsm_qui_msg(message); ........................................... qui_msg
xsm_quiet_err(message); ....................................... quiet_err
status = xsm_r_at_cur(screen_name); .............................. window
status = xsm_r_form(screen_name); .................................. form
status = xsm_r_keyset(name, scope); ............................... keyset
status = xsm_r_window(screen_name, start_line, start_column); .... window
xsm_rd_part(screen_struct, first_field, last_field); ............ rd_part
xsm_rd_struct(screen_struct, byte_count); .................... rd_struct
xsm_rescreen(); ............................................... rescreen
xsm_resetcrt(); ............................................... resetcrt
status = xsm_resize(rows, columns); ............................... resize
xsm_return(); ..................................................... return
xsm_rmformlist; ............................................. rmformlist
xsm_rrecord(structure_ptr, record_name, byte_count); ............ rrecord
lines = xsm_rscroll(field_number, req_scroll); .................. rscroll
status = xsm_s_val()(); ........................................... s_val
actual_max = xsm_sc_max(field_number, new_max); .................. sc_max
buffer = xsm_sdtime(format); ...................................... sdtime
status = xsm_select(group_name, group_occurrence); ............... select
xsm_setbkstat(message, display_attribute); .................... setbkstat
xsm_setstatus(mode); .......................................... setstatus
offset = xsm_sh_off()(); ......................................... sh_off
```

`xsm_shrink_to_fit();`	**shrink_to_fit**
`xsm_sibling(should_it_be);`	**sibling**
`size = xsm_size_of_array(field_number);`	**size_of_array**
`size = xsm_skinq(scope, row, softkey, value, display_attribute, label1, label2);`	**skinq**
`status = xsm_skmark(scope, row, softkey, mark);`	**skmark**
`status = xsm_skset(scope, row, softkey, value, attribute, label1, label2);`	**skset**
`status = xsm_skvinq(scope, value, occurrence, attribute, label1, label2);`	**skvinq**
`status = xsm_skvmark(scope, value, occurrence, mark);`	**skmark**
`status = xsm_skvset(scope, value, occurrence, newval, attribute, label1, label2);`	**skvset**
`outbuf = xsm_strip_amt_ptr(field_number, inbuf);`	**strip_amt_ptr**
`status = xsm_submenu_close()();`	**submenu_close**
`status = xsm_svscreen(screen_list, count);`	**svscreen**
`status = xsm_t_bitop(array_number, action, bit);`	**bitop**
`status = xsm_t_scroll(field_number);`	**t_scroll**
`status = xsm_t_shift(field_number);`	**tshift**
`xsm_tab();`	**tab**
`field_number = xsm_tst_all_mdts(occurrence);`	**tst_all_mdts**
`status = xsm_uinstall( usage, func, func_name);`	**uinstall**
`return_value = xsm_ungetkey(key);`	**ungetkey**
`status = xsm_unprotect(field_number);`	**protect**
`xsm_unsvscreen(screen_list, count);`	**unsvscreen**
`xsm_viewport(position_row, position_col, size_row, size_col, offset_row, offset_col);`	**viewport**
`status = xsm_vinit(video_address);`	**vinit**
`return_value = xsm_wcount()();`	**wcount**
`status = xsm_wdeselect()();`	**wdeselect**
`status = xsm_winsize()();`	**winsize**
`xsm_wrecord(structure_ptr, record_name, byte_count);`	**wrecord**
`xsm_wrt_part(screen_struct, first_field, last_field);`	**wrt_part**
`xsm_wrtstruct(screen_struct, byte_count);`	**wrtstruct**
`return_value = xsm_wselect(window_number);`	**wselect**

INDEX