Addendum

for Updates to JAM Release 5.03 Volume 2

for Stratus

Part Number R331–01A

August 3, 1992

Note of Explanation

This addendum describes new features in release 5.03 of **JAM**. This addendum is for Volume 2 of the documentation set. There is a separate addendum for Volume 1.

Descriptions of the features are broken into sections based on the parts of the manual that they affect. In addition, several insertion pages (or A–pages) are included for new library routines in **JAM** 5.03. These pages should be inserted into your **JAM** *Programmer's Guide* at the appropriate location. For example, page A–195 should be inserted before page 195.

Note that the page numbers refer to the August 1, 1991 printing of the **JAM** manual. If you are working with an older manual, insert these pages as appropriate, keeping in mind that the library routines are in alphabetical order.

JPL Guide

Page 13: Return Value from JPL Procedure After an Error

If an error occurs during execution of a JPL procedure (for instance, a math error), the procedure aborts and returns –1. This behavior was no previously documented.

Page 90: Prototyped Functions Called from JPL

Normally, hexadecimal, octal and binary numbers cannot be used in JPL. But when a JPL procedure calls a prototyped function that takes an integer argument, a string to integer conversion takes place. This conversion permits the use of hexadecimal, octal, or binary values as arguments to prototyped functions.

Programmer's Guide

Page 170: New Behavior and Return Codes for `sm_ascroll`

The library routine `sm_ascroll` takes as arguments a field number and an occurrence. It scrolls an array such that the requested occurrence is in the specified field. If the requested occurrence cannot be placed in the specified field because it is one of the first or last occurrences in a non–circular array, then `sm_ascroll` scrolls the occurrence onto the screen and returns the occurrence number of the occurrence that is actually in the specified field.

Page 254: Inquiring Help Level via `sm_inquire`

The global variable `I_INHELP` now contains the level of help that the user is in, instead of just a true/false value. There may be up to five levels of help. Use `sm_inquire` to

query the value of this variable. A return of zero indicates that the user is not in help, a return of 1 through 5 indicates which help level the user is in.

Page 285: `sm_keyoption`

Certain keys can not be translated via the `KEY_XLATE` argument to `sm_keyoption`. These are: INS, REFR, SFTS, LP, and ABORT. They may, however, be disabled via the `KEY_ROUTING` argument, or intercepted via a keychange function

Page 339: Percent Escapes in `sm_query_msg`

Percent escapes are now supported for controlling the attributes of query messages. The sequences are the same as those for `sm_emsg`, and detailed on page 214. Note that `%Mu` and `%Md` are not supported. Query messages from JPL can also now use percent escapes.

Page 391: MDT bits and Scrolling Arrays

When lines are inserted or deleted from scrolling arrays via INSL or DELL, the MDT bits for all occurrences after the insertion or deletion are no longer set. In a database application, this prevents the need for unnecessary processing to write potentially large amounts data that have not changed. For large arrays, it can save a significant amount of processing time.

copyarray

copy the contents of one array to another

SYNOPSIS

```
int sm_copyarray(destination_fld, source_fld)
int destination_fld;
int source_fld;
```

DESCRIPTION

This routine copies the contents of the array containing `source_fld` into the array containing `destination_fld`. `source_fld` and `destination_fld` are field numbers. They may be the field number of any of element in the respective array.

The developer is responsible for insuring that the arrays are compatible. Data in source array occurrences that are too long for the destination array are truncated without warning. Data in source array occurrences that are shorter than the destination array's field length are blank filled (with respect for justification).

If the source array has more occurrences than the destination array, the data in the extra occurrences are discarded. If the source array has fewer occurrences than the destination array, trailing occurrences in the destination array are cleared of data (but not de–allocated).

`copyarray` sets the MDT bit and clears the VALIDED bit for each destination array occurrence, indicating that the occurrence has been modified and requires validation.

The variant, `sm_n_copyarray`, searches the LDB for either array if the named field is not found on the screen. However, if the destination LDB item has a scope of 1, meaning that it is a constant, then it is not altered and the function returns –1.

RETURNS

–1 if either field is not found or if the destination array in the LDB has a scope of 1.
0 otherwise.

VARIANTS

```
sm_n_copyarray(destination_name, source_name);
```

RELATED FUNCTIONS

```
sm_clear_array(field_number);
sm_getfield(buffer, field_number);
sm_putfield(field_number, data);
```

key_integer

get the integer value of a logical key mnemonic

SYNOPSIS

```
#include "smkeys.h"

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

DESCRIPTION

This function returns the integer value of a **JAM** logical key mnemonic. The value is obtained from the file `smkeys.h`. This function is useful in cases where a function requires the integer value of a key, but cannot access the include files, as in a prototyped function called from JPL. The following table lists the logical 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	

RETURNS

the integer value of the logical key mnemonic.
0 if the mnemonic is not found.

RELATED FUNCTIONS

`sm_keylabel(key);`

EXAMPLE

The following example is from JPL. It sets the newline key to act as the tab key. The functions `sm_key_integer` and `sm_keyoption` must be prototyped in order to be called from a JPL procedure.

```
vars ret x y
retvar ret

call sm_key_integer "NL"
cat x ret

call sm_key_integer "TAB"
cat y ret

call sm_keyoption :x 2 :y
return
```

ldb_hash

use hash index for the LDB

SYNOPSIS

```
void sm_ldb_hash();
```

DESCRIPTION

This routine specifies that a hash table should be used to search the local data block. You must call `ldb_hash` before **JAM** initialization, in particular, before you call `sm_ldb_init` to initialize the Local Data Block.

Use of a hash table slightly improves the performance of routines which access the LDB, at the expense of the memory required for the table. This performance improvement includes the LDB merge which is performed the first time a screen with named fields is displayed. The degree of improved performance depends upon the distribution of the names in the LDB, and is greater for LDBs with more entries.

RELATED FUNCTIONS

```
sm_ldb_init();
```

EXAMPLE

```
#include "smdefs.h"

/* create a local data block with a hash index */

sm_ldb_hash();
sm_ldb_init();
```

next_sync

find next synchronized array

SYNOPSIS

```
int sm_next_sync(field_number)
int field_number;
```

DESCRIPTION

Given a field number, this function finds the next array synchronized with the given field, and returns the field number of the corresponding element in that array. The next synchronized array is defined as the one to the right. If `field_number` is in the righ-most synchronized array, the function returns the corresponding element in the leftmost synchronized array (ie– it wraps around the screcn).

RETURNS

The field number of the corresponding element in the next synchronized array if there is one.
Otherwise, the field number the function was passed.

set_injpl

allow C routines to access JPL variables & subroutines

SYNOPSIS

```
int sm_set_injpl(mode)
int mode;
```

DESCRIPTION

`sm_set_injpl` allows **JAM** internal routines to access JPL variables, including module and procedure locals, as if they were fields. This is most useful for the **JAM**/DB*i* statements `dbi_sql` and `dbi_dbms`, as well as the **JAM** library routine `sm_calc`.

However, `sm_putfield`, `sm_getfield`, or any other user function is not affected by the function call, and there is no means for user code to access JPL variables.

Normally a C routine that is called from JPL, via the JPL `call` statement, does not have access to either the "automatic" variables of the caller (the JPL `proc`) or the "static" variables in the module of the caller. If this routine is called with a `mode` that is non–zero, the C function will have access to both JPL "automatic" and "static" variables. It will also have access to any `proc`'s in the current (the caller's) JPL module. Thus it is as if the C function is embedded bodily within the JPL procedure.

The `mode` remains in effect until the calling JPL procedure is returned to, or `sm_set_injpl` is called again with a `mode` of zero. This means that all subroutines of the C routine will also have access to the current JPL module's variables and procedures. Of course, if the C routine calls a JPL `proc` (e.g. via `sm_jplcall`), the new JPL `proc` *will not* have access to variables in the JPL `proc` that called the C routine.

NOTE: This function should be used with care. For example, since `sm_jwindow` is a C subroutine, it too will have access to the current module's JPL variables and `procs`. In addition any screen entry, exit or validation functions will also have access to these variables and procedures. This can cause some unintended consequences when, for example, a JPL routine opens a screen, and the new screen's entry function calls a JPL `proc`. The JPL processor will look first in the original screen's JPL module (the current module) for the procedure, before it looks in the new screen's JPL module. If it finds a procedure of the same name in the current module it will execute that procedure instead of the procedure in the new screen's JPL. The safest way to use this routine is to set `mode` to a non–zero value when you require access, but then reset it immediately thereafter.

RETURNS

The previous value of `mode`.