Notes on EXECSYMB V3

by John Osudar

c/o Chemical Technology Division
Argonne National Laboratory
205 L-213
9700 South Cass Avenue
Argonne, IL 60439-4837

Telephone:
708-252-7505

Electronic mail address:
(Internet) `OSUDAR@CMT.ANL.GOV`
(DECUServe) `OSUDAR`

Current release of EXECSYMB

The current release of EXECSYMB is V3.6.1 (as of July 14, 1994). Release notes, describing changes made to EXECSYMB between all public releases, are included at the end of this document.

What is EXECSYMB?

EXECSYMB is, in VMS technical terms, a *user-written server symbiont*, i.e. a queue processing program that does not drive a printer, and is not supplied by DEC. More specifically, EXECSYMB is an interface between the VMS job controller/queue manager and a set of user-written *queue processors*, each of which handles a single specific print or server queue. (By the way -- throughout this document, "user-written" means "non-DEC-written", as it does in the DEC VMS documentation.)

Why EXECSYMB?

Assuming that the question "why write a queue processor?" has been answered (at least for a particular application), the next question is "why do it this way?". (If you're not sure why anyone would need to write a queue processor: there are certain applications that benefit from being handled in a sequential, queued manner, and VMS provides reasonable job queue management facilities.)

Under VMS, queue processors are known as *symbionts*. VMS provides a standard symbiont of its own, which is designed to handle the printing of jobs on a standard printer. VMS also provides two ways for implementing custom symbionts -- *user-modified*, and *user-written*. *User-modified* symbionts are implemented by allowing specific routines within the standard DEC VMS print symbiont to be replaced by user-written routines. The VMS print symbiont is provided as a *shareable image* that can be called from a program that wishes to become a symbiont. The program specifies which of its routines will handle which symbiont functions, and then calls the main symbiont entry point, after which the normal print symbiont code takes over.

A *user-written* symbiont, on the other hand, has to be written from scratch. VMS does provide some support routines to do this, primarily to interface between the user-written code and the job controller. However, the user-written symbiont still has to deal with a rather strange and difficult environment. For example: unlike other processes, a symbiont cannot use its SYS$INPUT or SYS$OUTPUT devices for anything other than communication with the job controller. As a result, the user-written program must be extremely careful not to call any routines (directly or indirectly) that may attempt to read SYS$INPUT or write to SYS$OUTPUT. The VMS job controller has not been very robust in this area; until recent versions of VMS, if it read something from its mailbox (i.e. the symbiont's SYS$OUTPUT) that it couldn't understand (such as a Fortran error message text, for example), it would complain -- and if it read an end-of-file on the mailbox (as can happen when Fortran gets done writing a series of error messages), the entire queue manager would abort, stopping all queues and leaving batch jobs running without corresponding queue entries. In short, if you're not careful, you can cause all kinds of havoc on your system with a user-written symbiont. (You can do the same with a user-modified symbiont, of course; however, since the routines you write for a user-modified symbiont are simpler and have limited functional requirements, it is easier to avoid producing such problems there.)

Symbionts under VMS have additional restrictions. For example, a symbiont process is created without a command language interpreter (CLI), such as DCL, so it cannot easily spawn subprocesses that execute DCL commands. Also, symbionts that hang, and are thus unable to read their job controller mailbox, have been known to cause the job controller to hang as well.

Under VMS, a symbiont can be *multi-threaded*, meaning that it can handle tasks from multiple queues with a single copy of the program. This is done by maintaining separate data structures for each queue, and looking at the proper structure for each queue when an operation is requested for that queue. A user-modified symbiont can plug into this relatively easily. However, in order to write a multi-threaded user-written symbiont, the code to handle multiple queues must be written from scratch by the symbiont author. This code resembles a second-level job scheduler that can handle multiple active jobs at the same time -- and thus, it is fairly complex. The alternative is to make each user-written symbiont single-threaded, which requires a separate process to be active for each queue whenever the queue is not in the *stopped* state.

All of these factors combine to make it attractive to add a standard layer of software between the VMS job controller/queue manager and the queue processing program. This is what EXECSYMB is intended to be. Built into EXECSYMB is the code that knows how to communicate with the job controller and to handle multiple active job streams, and the data structures required to maintain multiple job stream status information. In addition to this, EXECSYMB provides a simpler protocol for passing task information (i.e. items from the queue entry) to the queue processor.

How do you use EXECSYMB?

When initializing an *execution* queue (i.e. a printer queue or a server queue), you can specify the *processor* that is to serve as the symbiont for that queue. If no processor is specified, the default DEC print symbiont is used. If you specify `/PROCESSOR=EXECSYMB` the queue will be handled by EXECSYMB. (This document will refer to such a queue as "an EXECSYMB queue".) However, as mentioned above, EXECSYMB is really just an interface between the job controller/queue manager and the real queue processor. Thus, you have to specify the name of the program that will act as the real queue processor for each queue whose "official" symbiont is EXECSYMB. VMS doesn't provide any extra queue parameters that can be used for this purpose, but it does have a parameter called `LIBRARY`, which print symbionts use to specify a library file of device control modules. Since the device control library feature isn't useful for an EXECSYMB queue, EXECSYMB has taken over the `LIBRARY` parameter to specify the filename of a DCL command procedure that is the real queue processor. Except for a length limitation of 39 characters, the `LIBRARY` parameter is perfect for this purpose.

The other thing that an EXECSYMB queue requires is a way to specify some queue-specific parameters. Again, there is a queue parameter that is not otherwise useful for an EXECSYMB queue, namely the *job reset modules* (the value that shows up for the `SEPARATE=RESET` queue parameter). This parameter is allowed to be a string of up to 255 arbitrary characters, which makes it ideal for specifying a string of EXECSYMB-related parameters. Among what is included here are: the sequence of queue entry items to be passed to the queue processor; error retry information; item format information; inactivity timeout information; and various processing options.

What does EXECSYMB do?

When an EXECSYMB queue is started, EXECSYMB reads the queue parameters and sets up its data structure for that job stream. EXECSYMB can handle up to 32 job streams with a single EXECSYMB process. For each stream (queue), EXECSYMB can be instructed either to keep the queue processor permanently active, or to activate it only when there is work to do. A permanently active queue processor eliminates the need for startup overhead when a job is placed in a previously idle queue, but it requires an additional process to exist in the system as long as the queue is not stopped. A *dynamic* queue processor has to go through process creation overhead each time a new job comes in after the queue has been idle, but it doesn't use a process slot, memory, or any other resources while the queue is idle. (Some of that overhead can be reduced for queues that get bursts of work with longer idle periods. The setup of a dynamic queue includes an *idle timeout* value -- the queue processor will be allowed to sit idle for that period of time before being asked to exit.) Selection of a permanent or dynamic queue processor can be done on a queue-by-queue basis.

In either the permanent or dynamic case, when EXECSYMB activates the queue processor, it does several things. First, it creates the mailbox that it will use to send information to the queue processor. Then, it creates the queue processor process in a hibernating state. Next, it defines some logical names that will tell the queue processor how to find its mailboxes and its queue name. Finally, EXECSYMB "wakes up" the queue processor process, enabling it to begin execution, and sets a watch for the exit of the queue processor (so that, if the queue processor terminates, EXECSYMB will be notified).

Upon activation, the queue processor should determine the names of its input and output mailboxes (which it uses to communicate with EXECSYMB) and, if necessary, the name of the queue that it is processing. (EXECSYMB is started by the job controller with UIC `[1,4]`, and so it puts the logical names into the group logical name table for UIC group 1 (i.e. `LNM$GROUP_000001`). Since the queue processors are also started with UIC `[1,4]`, they can find the logical names in their group logical name table.) After completing these and any other required initialization steps, the queue processor waits for EXECSYMB to send it a task to execute.

When the job controller/queue manager sends EXECSYMB a message indicating that a new task is to be processed, EXECSYMB reads the information, and begins sending the items requested for the queue processor (via the *job reset modules* queue parameter), in the format specified for that queue, through the mailbox. EXECSYMB stops sending if the queue processor is not keeping up with the data being sent, and resumes sending when the queue processor catches up. While the processor for one queue is "catching up", EXECSYMB is able to process requests for other queues. Eventually, all requested queue entry items will be sent to the queue processor. At that point, EXECSYMB sends a special item called the *execute step* to the queue processor. This item tells the queue processor that all task information has been passed, and that the task should now be executed. Upon completion of task execution, the queue processor is expected to write the completion status to an output mailbox, which EXECSYMB reads. This completes the protocol between EXECSYMB and the queue processor. The queue processor returns to the idle state, and EXECSYMB processes the completion status and passes it (or a modified value) on to the job controller as the task completion status. (The queue processor can also send VMS accounting data in the same message as the completion status.)

The queue processor can also, at any time, send an *intermediate status* message to EXECSYMB (by writing it to the status mailbox.) This message can include a *device status* value (e.g. to set the queue "stalled") and a string of *checkpoint data*. The checkpoint data can be retrieved in a subsequent restart of the current job, should the job be aborted and requeued.

In addition to the *execute step* that initiates task execution, EXECSYMB may also send a *reset* command to the queue processor. This would happen, for example, if an active job is deleted from the queue before it has been completely processed. In that case, EXECSYMB will interrupt the item sequence and send an *execute step* that requests a reset to the idle state. The queue processor is expected to return to the state in which it is waiting for a new task, and EXECSYMB reports that the task has been aborted.

Another *execute step* command that EXECSYMB can send to the queue processor is *exit*. This is the command that is sent when a queue is being stopped. EXECSYMB then waits for the queue processor to log out, at which point it reports to the job controller that the queue has been stopped.

<u>What can the queue processor do?</u>

Because EXECSYMB handles the symbiont-related housekeeping work internally, a queue processor can be a relatively simple piece of software. As mentioned above, a queue processor is actually a DCL command procedure. This procedure can be as complicated as you wish -- or as simple as a single command, `RUN` *`programname`*, which would allow a real, compiled program in a real programming language (i.e. not DCL) to process the queue.

The queue processor has to adhere to the protocol established for communicating with EXECSYMB. This means that it has to read a mailbox for information, and write another mailbox when it completes a task. However, none of this has to happen asynchronously -- and so, the code can be fairly simple, and can indeed be written in DCL if desired. Beyond that, the VMS process within which the queue processor executes is a standard detached process that can access the DCL command language interpreter. This means that it can spawn subprocesses to execute commands, it can write output to log files, and it can do other "normal" process operations. (In fact, the queue processor is created with SYS$INPUT pointing to the command procedure, and SYS$OUTPUT to a log file in either the SYS_EXECSYMB or SYS$MANAGER directory. Thus, the queue processor can, from the moment of activation, write anything it desires to its standard output.)

While a symbiont can, by erroneously writing to its SYS$OUTPUT device, cause problems for the entire system's queue manager, a queue processor under EXECSYMB can do very little damage system-wide. If the queue processor sends erroneous status information to EXECSYMB, the status of tasks in the queue may be returned incorrectly. If the queue processor hangs, all that happens is that its queue hangs (i.e. tasks do not get processed) -- but all other queues handled by that copy of EXECSYMB continue to function. If the queue processor exits unexpectedly, EXECSYMB resets the queue to a stopped state, allowing a system manager to examine the problem, correct it, and restart the queue without loss of jobs.

<u>What is the protocol used by EXECSYMB?</u>

EXECSYMB can use one of two similar protocols to send information to the queue processor. One protocol, intended primarily for use with queue processors written in DCL, is in an ASCII format, while the other, intended primarily for use by processors written in higher-level languages, is in a binary format. The ASCII format requires two mailbox messages per item: the first message contains the ASCII name of the item (which can be used as a DCL symbol); the second message contains the value of that item. Thus, a DCL command sequence such as:

```
$ READ INPUT SYMBOLNAME
$ READ INPUT 'SYMBOLNAME
```

(i.e. read the name of the symbol, then read a value for that symbol) can be used by the queue processor to accept the items from EXECSYMB. The binary format, on the other hand, requires only one message per item; the first four bytes of the message contain the binary item number, while the rest of the message contains the item value. In the ASCII format, EXECSYMB attempts to edit items whose values are supplied by the job controller in binary into a suitable ASCII encoding; in the binary format, item values are passed to the queue processor exactly as they were received from the job controller.

A list of the item names, corresponding (decimal) item numbers, and an indication of the ASCII formatting of the item value, is provided below. This list is accurate as of VMS V4.4 through V5.4. VMS V5.5 introduces some additional items, which are listed separately; value types for all of the new items are currently set to 1 (no translation), as these items are still undocumented as of this time. (For information on the meaning of the documented items, see the VMS *Utility Routines* manual, in the sections on PSM and SMB routines.) Not all of these items can be present in a queue entry; those that cannot are presented here anyway, so that the number sequence is complete.

item name	no.	value type
ACCOUNTING_DATA	1	1
ACCOUNT_NAME	2	1
AFTER_TIME	3	3
ALIGNMENT_PAGES	4	2
BOTTOM_MARGIN	5	2
CHARACTERISTICS	6	5
CHECKPOINT_DATA	7	1
CONDITION_VECTOR	8	1
DEVICE_NAME	9	1
DEVICE_STATUS	10	5
ENTRY_NUMBER	11	2
EXECUTOR_QUEUE	12	1
FILE_COPIES	13	2
FILE_COUNT	14	2
FILE_SETUP_MODULES	15	1
FIRST_PAGE	16	2
FORM_LENGTH	17	2
FORM_NAME	18	1
FORM_SETUP_MODULES	19	1
FORM_WIDTH	20	2
FILE_IDENTIFICATION	21	1
FILE_SPECIFICATION	22	1
JOB_COPIES	23	2
JOB_COUNT	24	2
JOB_NAME	25	1
JOB_RESET_MODULES	26	1
LAST_PAGE	27	2
LEFT_MARGIN	28	2
LIBRARY_SPECIFICATION	29	1
MAXIMUM_STREAMS	30	2
MESSAGE_VECTOR	31	1
NOTE	32	1
PAGE_SETUP_MODULES	33	1

PARAMETER_1	34	1
PARAMETER_2	35	1
PARAMETER_3	36	1
PARAMETER_4	37	1
PARAMETER_5	38	1
PARAMETER_6	39	1
PARAMETER_7	40	1
PARAMETER_8	41	1
PRINT_CONTROL	42	5
PRIORITY	43	2
QUEUE	44	1
REFUSE_REASON	45	1
RELATIVE_PAGE	46	2
REQUEST_CONTROL	47	5
REQUEST_RESPONSE	48	2
RIGHT_MARGIN	49	2
SEARCH_STRING	50	1
SEPARATION_CONTROL	51	5
STOP_CONDITION	52	2
TIME_QUEUED	53	3
TOP_MARGIN	54	2
UIC	55	4
USER_NAME	56	1

V5.5 adds the items:

CHECKPOINT_FREQUENCY	57	1
QUEUING_CONTROL	58	1
RETRY_TIME	59	1
DEVICE_CONDITION	60	1
MESSAGE_FILE	61	1
AGENT_PROFILE	62	1
CPU_LIMIT	63	1
FILE_SEPARATION	64	1
LOG_QUEUE	65	1
LOG_SPECIFICATION	66	1
LOG_SPOOL	67	1
OPERATOR_REQUEST	68	1
WSDEFAULT	69	1
WSEXTENT	70	1
WSQUOTA	71	1
FILE_ATTRIBUTES	72	1
FILE_ATTRIBUTES_SIZE	73	1
JOB_ATTRIBUTES	74	1
JOB_ATTRIBUTES_SIZE	75	1
QUEUE_ATTRIBUTES	76	1
QUEUE_ATTRIBUTES_SIZE	77	1
SUBMITTER_EPID	78	1

EXECSYMB also defines two *pseudo-items*:

EXEC_STEP	0	1
EXEC_FLAGS	-1	1

The item value type, which is used by EXECSYMB to convert the item value to ASCII for streams requiring ASCII-format item values, has the following possible code numbers:

type	meaning
1	No translation (already ASCII, or item not easily translated to ASCII, so left in original form)
2	Binary longword, converted to hexadecimal in the format `%Xnnnnnnnn`
3	Date and time, converted to `dd-mmm-yyyy hh:mm:ss.cc` format (with one space between date and time)
4	UIC, converted to `[gggggg,mmmmmm]` format with octal group and member numbers containing leading zeroes.
5	Bitstring, converted to a list of which bits are set. Up to 16 bytes of data can be included in the item (since the CHARACTERISTICS item is 16 bytes long.) The list is a series of decimal numbers separated by commas (e.g. hex 641 becomes `0,6,9,10`)

Pseudo-items

As mentioned earlier, pseudo-item 0 (`EXEC_STEP`) indicates the end of a sequence of items (a *task*, in VMS symbiont terms) and requests that the queue processor take a specific action (e.g. execute the task, reset itself, or exit.) Another pseudo-item, with the binary item code -1 (and the ASCII name `EXEC_FLAGS`) provides an optional means for EXECSYMB to pass certain task characteristics to the queue processor. The value of EXEC_FLAGS is a series of keywords separated (and surrounded) by slashes ("`/`"). Currently, the only keyword that can be passed is `SPOOL`, which indicates that the FILE_SPECIFICATION for the current task was built by entering the file in the queue's *spool* directory. If this flag is present, the item value would be `/SPOOL/`; if no flags are present, the value passed is `//`.

More details on queue parameters

As mentioned earlier, the way that EXECSYMB handles a queue is determined by the parameters specified in the *job reset modules* queue qualifier. The value of this qualifier is a character string which contains a sequence of EXECSYMB parameters separated by commas. The following list includes a brief description of all available EXECSYMB parameters, as of version V3 of the EXECSYMB program:

`TIME="d hh:mm:ss.cc"`	In the event that a task completes with an error status, this is the amount of time to wait before retrying this job
`ITEMS="itemlist"`	A list of the item numbers to be passed to the queue processor, in the order they are to be passed. Specify items as single numbers (NN) or ranges (MM:NN). Note that the EXEC_STEP pseudo-item (0) is always passed (and should not be listed here!) and that

	the EXEC_FLAGS pseudo-item (-1) is specified through another parameter (and should also not be listed here.)
`SPOOL=[spooldir]`	For jobs whose files are *spool files*, produced by writing to a spooled device or by creating a *temporary file* with no directory entry, this is a directory on the corresponding disk into which the spool files can be temporarily entered (if the queue processor needs to access files by name, not by file-id)
`ASCII` or `BINARY`	The format in which item values are passed
`PRINTER` or `SERVER`	This simply specifies whether the queue is displayed as a *printer* or *server* queue. (This can also be changed dynamically through intermediate *device status* messages.)
`NONULL` or `NULL`	Some items in each queue entry will be null (i.e. unspecified); this indicates whether the queue processor wishes to receive requested items whose values are null
`CHECKPOINT` or `NOCHECKPOINT`	Multi-file (or multi-copy) jobs that are aborted in the middle are normally flagged *checkpointed*; this determines whether an aborted job should be marked checkpointed (meaning that it will restart from the point where it was aborted) or not (meaning that it will restart from the beginning)
`COPY={ALL`, `First`, or `Last}`	For multi-copy jobs, and multiple copies of a file within a job, this specifies which copy of the request is to be passed to the queue processor -- all copies, the first copy only, or the last copy only
`USER=username`	EXECSYMB runs under the SYSTEM username, and all queue processors also do so by default; an alternate username (which need not be "valid", i.e. in SYSUAF) can be specified for a specific queue processor
`DYN="d hh:mm:ss.cc"`	If a queue processor is to be *dynamic*, i.e. started when its queue becomes non-empty, and stopped when the queue is idle, this parameter specifies the amount of time the queue should be idle before the queue processor is asked to exit
`FLAG` or `NOFLAG`	If FLAG is specified, the EXEC_FLAGS pseudo-item will be sent for each task
`INIT` or `NOINIT`	Used with the `DYN` parameter (i.e. for *dynamic* queue processors); specifies that the queue processor should be started when the queue is first started (so that the queue processor may perform some initialization operations), even if there are no jobs in the queue. (If no jobs are queued before the dynamic processor timeout expires, the queue processor will be asked to exit, as usual.)

`HOLD` or `NOHOLD`	Used instead of the `TIME` parameter (i.e. for requeuing a job when it completes with an error status); specifies that requeued jobs should be placed on indefinite hold instead of being scheduled to retry after a fixed time interval.

The default parameters are as follows:

no TIME and NOHOLD	(i.e. aborted jobs are neither retried nor requeued and put on hold)
no `ITEMS`	(obviously not useful -- you should specify some items!)
no `SPOOL` directory	(i.e. doesn't process files by name, or cannot accept spool files)
`ASCII`	(pass item values in ASCII format)
`SERVER`	(indicate that this is a *server* queue, not a *printer* queue)
`NULL`	(pass specified items to the queue processor even if corresponding values are null)
`CHECKPOINT`	(mark aborted jobs as *checkpointed*, so that they restart from the point where aborted)
`COPY=ALL`	(pass all copies of a multi-copy job or multi-copy task to the queue processor)
`USER=SYSTEM`	(run the queue processor under username SYSTEM)
`NOFLAG`	(do not send EXEC_FLAGS item)
`NOINIT`	(do not start dynamic processor at queue startup)

Spool files and the SPOOL directory

The use of the SPOOL parameter to specify a *spool directory* was mentioned above. This feature needs some additional explanation.

When a VMS process writes to a spooled device, a file is created that is not entered into any directory. This file is marked as a *spool file*, and exists only until the print job completes. (The job controller checks for the *spool file* attribute, and deletes the file automatically if it is set.) Some utilities (e.g. MAIL) also produce temporary output files that are not entered into a directory.

Such files are processed by the standard print symbionts using the file-ID to open the file; the FILE_SPECIFICATION item contains a string that can't be used as a filename (e.g. `DISK$SPOOL:[]MAIL.TMP`). An EXECSYMB queue processor that uses the FILE_SPECIFICATION item wouldn't work with files that aren't entered into a directory.

To get around this problem, EXECSYMB uses the directory specified in the queue's SPOOL parameter. It temporarily enters the spool file into that directory, and passes the resulting file specification as the value of the FILE_SPECIFICATION item. After the task completes, EXECSYMB removes the directory entry. This allows an EXECSYMB queue processor to access any file, including a spool file, using the FILE_SPECIFICATION item value.

The SPOOL item need only be specified for queues that meet both criteria: (a) the queue processor must access (e.g. open) the file using the FILE_SPECIFICATION; and (b) it is possible for spool files or other non-directory-entered files to be submitted to the queue.

A note about files submitted `/DELETE`

When dealing with spool files, or other files that are to be deleted after processing, the queue processor must be carefully written if the contents of the file are to be preserved after the task is "completed" (for example, if the file is resubmitted to another queue for processing). The job controller will delete the file as specified, as soon as the task reports completion. Renaming the file will not prevent deletion. The only way to deal with this situation is to make a copy of the file before reporting task completion.

Sample queue processing procedure

The following is a sample queue processing procedure that can be used with EXECSYMB. It is reproduced here to illustrate the required features of such a procedure; it can also be found, without comments, in the distribution directory in the file EXAMINEQUEUE.COM (which can be used as-is with EXECSYMB.) This procedure does nothing "useful"; it produces a log file containing the items passed for each request it processes, and it indicates successful completion of each task.

First, you can specify any setup commands you require. SET NOVERIFY and SET NOON are recommended.

```
$ SET NOON
$ SET NOVERIFY
```

Next, the procedure must locate the mailboxes that it uses to talk to EXECSYMB. (Mailboxes are identified by logical names. All logical names created by EXECSYMB are in the group logical name table for UIC group 1, i.e. `LNM$GROUP_000001`.) The input mailbox (written by EXECSYMB, read by the procedure) has a logical name of `CMD_MBX_`*`pid`* where *`pid`* is the process ID of the detached process running this procedure. The output mailbox (written by the procedure, read by EXECSYMB) is used for returning status, and has the name `STAT_W_MBX_`*`pid`* (where *`pid`*, again, is the process ID). We can also determine the name of the queue we are processing by translating the logical name `QUEUE_NAME_`*`pid`*, and the name of the device specified in the `/ON=`*`node::device`* qualifier by translating `DEVICE_NAME_`*`pid`*. Thus, the procedure must determine its own PID, and then use it to access this information.

```
$ PID=F$GETJPI("","PID")
$ QUEUENAME=F$TRNLNM("QUEUE_NAME_''PID'")
$ WRITE SYS$OUTPUT F$TIME()," EXAMINEQUEUE starting on queue ",QUEUENAME
```

Before we do any real work, we must open the input and output mailboxes for talking with EXECSYMB. (The `INIT` label is used here for mailbox error handling.)

```
$INIT:
$ OPEN/WRITE/ERR=NOMBX STAT STAT_W_MBX_'PID':
```

```
$ OPEN/READ/ERR=NOMBX INPUT CMD_MBX_'PID':
```

When EXECSYMB explicitly requests a RESET (such as when a job is deleted while being processed), or after an EXECUTE operation completes, the procedure should reset controlling variables to initial values, and return to its input processing loop. This procedure has one controlling variable, `EXEC_STEP`, which is reset to point to the INPUT loop. (It also writes out a separator line to the log file, to mark the boundary between requests.) A common technique is shown here: putting the RESET segment between the initialization and the input loop, thus executing the reset commands when the processor is first started.

```
$RESET:
$ WRITE SYS$OUTPUT "========================================="
$ EXEC_STEP="INPUT"
```

The items sent by EXECSYMB all have names that can be used as DCL symbols. That is the technique used here. (EXECSYMB has an alternate format for items, in which a fixed-length binary item code precedes the item value; that format is intended for use by compiled programs that run in the detached process.) The item EXEC_STEP is used by EXECSYMB to guide the operation of the detached process; EXECSYMB can assign it the values `EXECUTE`, `RESET`, and `EXIT`. We initialized the symbol to the value `INPUT`, and we use the symbol as the destination label of a GOTO. This is a rather simple way of implementing the control operation.

Now comes the "good part" of the procedure. It starts with the label that was initially assigned to EXEC_STEP. The first command reads the **name** of the symbol (i.e. the item name) being passed, and the second command reads its **value**. Thus, EXECSYMB can tell the process which item is being passed, and its value, and a symbol of that name with that value is created.

```
$INPUT:
$ READ/ERR=ERRMBX INPUT SYMBOL
$ READ/ERR=ERRMBX INPUT 'SYMBOL
```

For demonstration purposes, we echo the information to SYS$OUTPUT, which is the log file.

```
$ WRITE SYS$OUTPUT SYMBOL
$ WRITE SYS$OUTPUT 'SYMBOL
```

Having completed the value assignment, we go to the location pointed to by EXEC_STEP. This was initially `INPUT`, and will remain so until EXECSYMB sends the item EXEC_STEP and a new value (`EXECUTE`, `RESET`, or `EXIT`) for it.

```
$ GOTO 'EXEC_STEP
```

When EXECSYMB says `EXECUTE`, it indicates that the item sequence for this task is complete, and should be acted upon. A real symbiont processor would do something with the item list at this point (e.g. do something with the file specified in item FILE_SPECIFICATION). Once the operation is complete, EXECSYMB expects a status to be returned via the mailbox. This example always returns a success status. (The status, like all numbers sent by the queue processor to EXECSYMB, can be either a decimal number, or a hex number preceded by `%X`.)

```
$EXECUTE:
$ WRITE/ERR=ERRMBX STAT "1"
```

After an `EXECUTE` operation is complete, the procedure should do an implicit `RESET` operation. We branch to the RESET section for this.

```
$ GOTO RESET
```

Finally, if an `EXIT` operation is requested, EXECSYMB wants the detached process to exit. One way to do this is to `EXIT`, another is to `LOGOUT`; you can also do whatever else you desire (including, for example, sending operator notification that the detached process is exiting.)

```
$EXIT:
$ EXIT
```

All that's left is error handling for the mailboxes. NOMBX signals a failure to open the designated mailbox, and is a fatal error -- the only proper way to deal with it is to exit, thus signalling EXECSYMB to stop the queue. If an error is encountered while reading or writing a mailbox, the mailbox may be closed and reopened. This may clear the error, or it will result in an open failure that will terminate the procedure at NOMBX.

```
$NOMBX:
$ WRITE SYS$OUTPUT "Mailbox open failure; exiting"
$ EXIT
$ERRMBX:
$ WRITE SYS$OUTPUT "Mailbox I/O error; reinitializing"
$ CLOSE STAT
$ CLOSE INPUT
$ GOTO INIT
```

<u>A walk through an EXECSYMB job</u>

The following text describes the operations that are performed during the processing of a hypothetical job by an EXECSYMB queue processor. This hypothetical queue is initialized with the following command:

```
          INIT/QUEUE SAMPLE/PROC=EXECSYMB -
          /LIBRARY=SYS$MANAGER:SAMPLE.COM -
          /SEP=RESET=(DYN="0 :2",TIME="0 :1",INIT,NONULL,COPY=L,SPOOL=[SPOOL],ITEMS=13,1
5,22:23)
```

This specifies a dynamic queue processor with a two-minute timeout, a one-minute reschedule delay on jobs that complete with an error status, queue processor startup at initialization, no passing of null items, processing only on the last copy of a job and file, and four items passed to the queue processor: item 13, FILE_COPIES (number of copies for each file); item 15, FILE_SETUP_MODULES (the value specified with the `/SETUP` qualifier on the PRINT command); item 22, FILE_SPECIFICATION; and item 23, JOB_COPIES (number of copies for the whole job).

The queue processor `SAMPLE.COM` illustrates some useful queue processor programming techniques, including the use of the retry-on-error feature to automatically requeue the job, and one way to copy the file (which can be used to avoid having it deleted prematurely). The queue processor itself is rather contrived and not all that useful. (It uses the `/SETUP` value as a DECnet nodename, and attempts to make the specified number of copies of the specified files on the default DECnet account of that node.)

The DCL code for the queue processor is listed below; see the comments for special notes.

```
$ SET NOON
$ SET NOVERIFY
$ PID=F$GETJPI("","PID")
$ SET PROCESS/PRIV=NETMBX  ! Process is started with only SETPRV enabled
$INIT:
$ OPEN/WRITE/ERR=NOMBX STAT STAT_W_MBX_'PID':
$ OPEN/READ/ERR=NOMBX INPUT CMD_MBX_'PID':
$RESET:
$ EXEC_STEP="INPUT"
$ FILE_COPIES="1"
$ JOB_COPIES="1"
$ FILE_SETUP_MODULES=""
$ FILE_SPECIFICATION=""
$ SET OUTPUT  ! Useful to allow immediate examination of logfile
$INPUT:
$ READ/ERR=ERRMBX INPUT SYMBOL
$ READ/ERR=ERRMBX INPUT 'SYMBOL'
$ GOTO 'EXEC_STEP'
$EXECUTE:
$ COUNT=F$INTEGER(JOB_COPIES)*F$INTEGER(FILE_COPIES)  ! Compute #copies
$ I=0
$COPYLOOP:
$ I=I+1
$ IF I .GT. COUNT THEN GOTO COPYDONE
$ CALL/OUT='PID'.TMP COPY_FILE 'FILE_SETUP_MODULES'::*.*;0
$ OPEN/READ/ERR=NOLOG TMP 'PID'.TMP  ! Read COPY messages for result
$READLOOP:
$ READ/END=BADLOG/ERR=BADLOG TMP LINE
$ IX=F$LOCATE(" copied to ",LINE)  ! Find "copied to" to locate filespec
$ L=F$LENGTH(LINE)
$ IF IX .EQ. L THEN GOTO READLOOP
$ CLOSE TMP
$ DELETE 'PID'.TMP;*
$ NEWFILESPEC=F$EXTRACT(IX+11,L-IX-11,LINE)  ! Extract output filespec
$ LNFS=L-IX-11
$ IP=F$LOCATE("(",NEWFILESPEC)
$! Log the successful copy
$ IF IP .NE. LNFS THEN NEWFILESPEC=F$EDIT(F$EXTRACT(0,IP,NEWFILESPEC),"TRIM")
$ WRITE SYS$OUTPUT "Copy ",I,": copied ",FILE_SPECIFICATION," to ",NEWFILESPEC
$ GOTO COPYLOOP
$COPYDONE:
$ WRITE STAT "1"  ! success
$ GOTO RESET
$BADLOG:
$ CLOSE TMP
$ DELETE 'PID'.TMP;*
$NOLOG:
$ WRITE STAT "4"  ! failure
$ GOTO RESET
$COPY_FILE: SUBROUTINE  ! copy file (allows capture of messages)
$ COPY/LOG/PROT=O:RWED 'FILE_SPECIFICATION' 'P1'
$ ENDSUBROUTINE
$NOMBX:
```

```
$ WRITE SYS$OUTPUT "Mailbox open failure; exiting"
$ EXIT
$ERRMBX:
$ WRITE SYS$OUTPUT "Mailbox I/O error; reinitializing"
$ CLOSE STAT
$ CLOSE INPUT
$ GOTO INIT
$EXIT:
$ WRITE SYS$OUTPUT "Exiting"
$ EXIT
```

Let's look at what happens when a job is queued to the `SAMPLE` queue with the following command (or its equivalent):

```
PRINT/QUEUE=SAMPLE/JOB_COUNT=2 FOO.BAR/COPIES=3/SETUP=ABC,SNA.FOO/SETUP=XYZ
```

When the job controller/queue manager receives this request, it sees it as eight tasks" two copies of the job, each containing three copies of one file and one copy of another file. It begins the first task by formatting and sending a message to the symbiont for queue `SAMPLE`, i.e. to EXECSYMB. EXECSYMB receives the message through its SYS$INPUT: mailbox and provides whatever responses are required by the protocol. It then determines which of its job streams is associated with queue `SAMPLE`. If the dynamic queue processor for that queue is not active, EXECSYMB creates a detached process running the LOGINOUT image, with `SYS$MANAGER:SAMPLE.COM` as the input, and (by default) `SYS_EXECSYMB:SAMPLE.OUT` as the output.

If the queue parameters specified `COPY=ALL` (the default), EXECSYMB would send all eight tasks to the queue processor. Because of the `COPY=LAST` option, however, it will only send two tasks, which are copy 2 of the job, consisting of copy 3 of the first file and copy 1 of the second file. (Specifying `COPY=FIRST` would also result in two tasks: copy 1 of the job, consisting of copy 1 of the first file and copy 1 of the second file. The primary difference between the two is in how the job status appears in a `SHOW QUEUE` command.)

EXECSYMB will process the first task by noticing that it isn't the last copy of the job, and so shouldn't be sent to the queue processor. It will report "successful completion" to the job controller without accessing the queue processor at all. It will similarly handle the other two copies of the first file, and the single copy of the second file, all part of copy 1 of the job.

With four tasks "done", EXECSYMB will now receive the fifth task, the first copy of the first file in job copy 2. Since this isn't the last copy of this file, EXECSYMB will again report immediate "successful completion"; the second copy of the first file will do likewise. With six tasks now "done", EXECSYMB receives the task of processing the third copy of the first file in job copy 2. This is the last copy of this file in the last copy of the job, so EXECSYMB has to forward this to the queue processor.

If the file being processed in the current task is a spool file, EXECSYMB checks to see if the queue specified a value for the SPOOL parameter (as it does in this example). In that case, EXECSYMB will create a temporary directory entry for that file in the specified directory, and will replace the FILE_SPECIFICATION item value passed by the job controller with the resulting file specification. In the case of this example, the files have directory entries, so no entries need to be created by EXECSYMB.

EXECSYMB now writes the requested items, one by one, to the mailbox it uses for passing task items to the queue processor. It will only write the four items that were specified in the queue parameters, and it won't write any of those four that have null values (e.g. if no `/SETUP` is specified, it won't send the FILE_SETUP_MODULES item.) For each item, it writes the name of the item, followed by the formatted value of the item, as separate messages to the mailbox.

The queue processor has been sitting at a READ command, waiting for something to appear in its input mailbox. When items start coming in, the INPUT loop assigns the item values to symbols whose names are the item names. After all the items have been sent and received, EXECSYMB writes the EXEC_STEP item to the mailbox, with value EXECUTE. The queue processor reads this as with any other item -- but the change in the value of the EXEC_STEP symbol breaks it out of the INPUT loop and goes to the EXECUTE label. There, the queue processor does its work; in this case, it copies the file as many times as was specified. If the operation succeeds, the queue processor reports a success status for the task; if it fails, it reports an error status.

If a success status is returned to EXECSYMB, it forwards that to the job controller, which completes the task and moves on to the next one. The eighth and final task proceeds in much the same way as the one just described.

If, however, a failure status is returned to EXECSYMB, its behavior depends upon the presence or absence of the TIME queue parameter. If the TIME parameter is **not** specified (the default), the failure status is also returned to the job controller, which aborts the job with an error. If TIME **is** specified, EXECSYMB doesn't return any status to the job controller. Instead, it uses the standard $SNDJBC system service to request that the current job be aborted, requeued and set `/HOLD`. The job controller responds to this request by sending an appropriate message to the symbiont -- back to EXECSYMB -- which is expecting it. EXECSYMB reports that the job has been aborted successfully. The job controller then requeues the job and sets its state to "holding". EXECSYMB now uses the time value specified in the TIME parameter (one minute, in this example) to compute a release time for the job, and it does another $SNDJBC request, modifying the held job into a timed-release job. Thus, the failed job will be retried after the specified time interval -- and, since checkpointing is allowed, the job will restart with the task that failed and not the first task of the job.

Eventually, the queue will sit idle for the dynamic processor timeout period (two minutes, in this example), and EXECSYMB will send an EXEC_STEP item to the queue, specifying `EXIT` as the value. The queue processor will exit -- but the queue will remain in the started (but idle) state as far as the VMS job controller is concerned.

EXECSYMB Release Notes

The following pages contain the release notes for all recent public releases of EXECSYMB. They are included here because they document some of the EXECSYMB features that haven't been described elsewhere.

Release notes for EXECSYMB V02.32

15-Jun-1989

Changes since the previous release (V02.29, 23-Feb-1988) are as follows:

1. Synchronization between AST-level and non-AST-level code was improved. Also, use of hibernation and wakeup calls was improved to cut down on the number of system service calls.

2. The mailbox message size for the status mailbox was increased from 12 to 268 bytes, to allow room for accounting, checkpoint, and device status information.

3. The status message format was extended to allow specification of accounting data, checkpoint data, and device status. The program logic was enhanced to recognize "intermediate" status messages containing checkpoint and device status information. (In other words, instead of only recognizing a status message from a stream at task completion, EXECSYMB will also accept such messages during task execution, to provide checkpoint data or device status to the job controller.)

   The old status message was required to have the format %Xnnnnnnnn where the n's are hex digits. The new status message can have two formats. An intermediate status message looks like this:

   ```
   ,{optional device status}{,optional checkpoint data string}
   ```

   e.g. `,16,this is a checkpoint string` to report that the queue is stalled and to set "this is a checkpoint string" as the checkpoint value.

   A task completion status message looks like this:

   ```
   <completion status>{,optional accounting data}
   ```

   e.g. `1,5,0,0,0` to return a success status and report 5 pages "printed".

   Status values are numeric longwords, represented as either a signed decimal number, or a hex number preceded by "%X". Accounting data consists of four such numbers, representing the four accounting data fields (pages printed, QIO's to printer, GET's from source, and hundredths of CPU-seconds used.) Checkpoint data is an arbitrary character string. For more details, read the SMB section of the VMS Utility Routines manual.

   A note about the "device status" field: this can be used to specify various queue status or attribute values. There are also some unsupported bits in the status (at least as of VMS V5.4), which may bequite useful in some applications. Here is a brief summary of the bits in the device status field as of VMS V5.4:

SMBMSG$V_name	bit#	mask value	
LOWERCASE	0	1	Sets "Lowercase" attribute
PAUSE_TASK	1	2	Initiates pause-queue
REMOTE	2	4	Sets "remote" attribute
SERVER	3	8	Sets "server" attribute
STALLED	4	16	Indicates queue "stalled"
STOP_STREAM	5	32	Initiates stop-queue
TERMINAL	6	64	Sets "terminal" attribute
UNAVAILABLE	7	128	Sets "device unavailable"
SYM_NOTIFIES	8	256	?
SYM_REQUESTS_OPER	9	512	?
SYM_COPIES_FILE	10	1024	?
SYM_COPIES_JOB	11	2048	?
SYM_ACCEPTS_ALL_FORMS	12	4096	Allows all jobs, any /FORM
SYM_NO_JOB_CHECKPOINT	13	8192	?

4. A feature called "permanent processor support" had been added to EXECSYMB in V02.27. This was supposed to provide a way for an existing detached process to "register" itself as a queue processor (instead of EXECSYMB creating the process when the queue was started. The feature wasn't designed too well, and was only added to support one specific application (which, naturally, ended up never using it). Because permanent processor support was incompatible with the intermediate status message support described above (which *is* useful), and because, to my knowledge, nobody ever used it, permanent processor support has been removed from EXECSYMB as of V02.32. If you built something using the "permanent processor" feature, it won't work with versions of EXECSYMB beyond V02.29 -- and I would like to hear from you to discuss alternative implementations.

Release notes for EXECSYMB V3.0.1

12-Jul-1989

Changes since the previous release (V02.32, 15-Jun-1989) are as follows:

1. Use (and misuse) of event flags in previous versions was completely cleaned up. This may have been at least partly responsible for "strange behavior" in past releases of EXECSYMB.

2. Cosmetic changes to process names were made (EXECSYMB process has a name like "ExecSymb3.0.1_n" instead of "SYMBIONT_EXEC_n" and queue processors are named "nx=queuename" instead of "nx_queuename".)

3. EXECSYMB determines whether it is running on VMS V4.x or a later version. If a later version, it uses the new queue manager features that were added in VMS V5.0 (e.g. queue name no longer needs to be specified when changing a particular entry's attributes.)

4. EXECSYMB now explicitly sets its base priority to 7 (one below the current level used by the job controller), instead of using whatever was specified as

the /BASE_PRIORITY qualifier on the first queue started for this symbiont. Also, EXECSYMB now uses the system parameter DEFPRI to determine the base priority of the queue processors that it creates. (Thus, they will typically run with base priority 4.)

5. A bug in the protection attached to the status mailbox was fixed. Prior to this version, the status mailbox was inadvertently left with full access for all users, when it should have had full access only for SYSTEM processes.

6. Some additional stream status information was added, to reduce the amount of time EXECSYMB uses for scanning streams that have never been active (both in its "main loop" and when a status message is received.)

7. Logfile messages (and some operator messages) were shortened to eliminate extraneous text. (Since debugging symbiont problems required finding the message in the source code anyway, having a longer message didn't really provide any useful information.)

8. For several versions prior to this release, it had not been possible to get null items sent to the queue processor (i.e. all queues were effectively "NONULL" whether they specified it or not.) This bug has been fixed. (Most queue processors will still want to specify NONULL for efficiency's sake.)

9. Previous versions of EXECSYMB did no validity checking of the requeue-time or dynamic-time queue parameters; such checks are now included.

10. This isn't really a change, but should be noted somewhere: EXECSYMB has always used the LIB$FID_TO_NAME routine to convert a file-ID into a full filespec. Prior to VMS V5.0, this routine was supplied with EXECSYMB as an object module (and a Bliss-32 source file). VMS V5.0 added LIB$FID_TO_NAME to the LIBRTL, so EXECSYMB for V5.0 and beyond no longer requires the explicit object module reference in its build procedure's LINK command.

11. The (undocumented) "debug" mode was enhanced to allow it to be turned on and off dynamically, and to display all current status information.

Release notes for EXECSYMB V3.1.1

02-Mar-1990

Changes since the previous public release (V3.0.1, 12-Jul-1989) are as follows:

1. EXECSYMB now saves the "device name" specified in the "/ON=" qualifier for each queue, and creates a logical name DEVICE_NAME_pid (where "pid", as usual, is the process ID of the queue processor for that queue.) This provides a way to pass a queue-dependent parameter of up to 31 characters directly to the queue processor. Note that the SCS node name portion of "/ON=" is *not* part of this parameter. Also note that VMS imposes no syntax restrictions on the "device name", as long as the string is enclosed in quotes.

   An example of the use of this feature: we have several queues that use an NJE package to send print files to devices on our IBM systems. In the past, we used a separate command procedure to drive each queue processor -- now we use a single procedure, and specify the destination for each queue through the

"/ON=" parameter. It makes things simpler and more flexible, and it's nice to see the information displayed in a SHOW QUEUE instead of being buried in a command procedure somewhere.

2. A bug that prevented EXECSYMB from stopping some hung queue processors has been fixed. In past releases, if a queue processor changed its UIC to something other than [1,4] and then became hung, EXECSYMB's exit timeout code would fail to stop the process because EXECSYMB didn't enable WORLD privilege. As of this release, EXECSYMB correctly enables WORLD privilege, and thus is able to stop any hung processor. (All symbiont processes, such as EXECSYMB, are started with SETPRV and no other privileges enabled -- so fixing this problem simply required adding WORLD to the list of privileges that EXECSYMB enables.)

Release notes for EXECSYMB V3.1.2

21-Jun-1990

Changes since the previous public release (V3.1.1, 02-Mar-1990) are as follows:

1. One very minor bug fix, to the code that re-converts the "TIME=" and "DYN=" queue parameters to ASCII (to reset the parameter string after the queue is started), to display the hours correctly.

Release notes for EXECSYMB V3.2.4

20-Aug-1990

Changes since the previous public release (V3.1.2, 21-Jun-1990) are as follows:

1. The ability was added to disable requeue-for-retry (i.e. to force a job to complete with an error or fatal error status) in a queue that is enabled for retry (i.e. for which a TIME= parameter is specified). This feature is implemented by making the status (written to the EXECSYMB status mailbox) the negative of the actual status.

2. EXECSYMB now defines some additional logical names in the group [1,*] logical name table (LNM$GROUP_000001). These names may be used to determine the association of queues, streams and queue processor processes related to a particular copy of the EXECSYMB symbiont process. Every time a stream is started by EXECSYMB process number N (i.e. the one whose process name is "ExecSymb3.2.4_N"), a logical name of the form EXECSYMB_Nx is created (where x is the stream number encoded as a "base 32" digit, 0-9 or A-V); its equivalence name is the name of the queue for which that stream was started. This logical will remain defined for as long as the EXECSYMB process exists (although its equivalence name may change.) Also, when a queue processor process is actually created, a name of the form EXECSYMB_PID_Nx is defined, whose equivalence name is the process ID of the queue processor. Thus, for any EXECSYMB process, you can determine the queues and active queue processors that belong to it.

3. The $DEBUG mode "EXIT" command (executed by writing the string "$DEBUGEXIT" to the symbiont's status mailbox) was changed to stop all streams' queue processors. A "QUIT" command (i.e. "$DEBUGQUIT") was added to do the

"immediate exit" that was formerly what EXIT did. This change was made to provide a clean way of stopping the symbiont and its associated processes after a queue manager failure. (A sample command procedure, EXECUTIL.COM, is supplied with the source code, illustrating some of these features.)

Release notes for EXECSYMB V3.2.6

01-Apr-1991

Changes since the previous public release (V3.2.4, 20-Aug-1990) are as follows:

1. Added queue-specific locations for the *.OUT and *.ERR log files. If the logical name "EXECSYMB_OUTDIR_queuename" is defined in the system logical name table, its value is used as the device/directory where the log files for queue "queuename" are created.

Release notes for EXECSYMB V3.2.7

23-Oct-1991

Changes since the previous public release (V3.2.6, 01-Apr-1991) are as follows:

1. Added a queue parameter "INIT", which is used with the "DYN=" parameter to indicate that the dynamic queue processor is to be started when the queue is first started. (Ordinarily, when a queue is started and a dynamic queue processor is specified, the processor is not started until a job is available for processing.) This is useful if the queue processor procedure needs to do some initialization at queue startup, such as sending device status bits to the job controller. The queue processor will be started with the usual dynamic timeout.

Release notes for EXECSYMB V3.3.1

31-Mar-1992

Changes since the previous public release (V3.2.7, 23-Oct-1991) are as follows:

1. Initial update to include new items defined (but not documented) by VMS V5.5. (All new items are presently passed untranslated. Note that this will almost certainly change in a future release of EXECSYMB -- if you use any of these undocumented items, please keep this in mind!) Previous releases of EXECSYMB <u>will not work</u> under VMS V5.5; however, this release of EXECSYMB <u>will</u> work with previous releases of VMS.

2. Compiling this version of EXECSYMB with a VMS V5.5-compatible SMBDEF.INC (supplied with the source code distribution of V3.3.1) will produce a symbiont that will work with VMS V5.5 as well as previous releases of VMS.

Release notes for EXECSYMB V3.4.1

14-Oct-1992

Changes since the previous public release (V3.3.1, 31-Mar-1992) are as follows:

1. The code was cleaned up a bit: an IMPLICIT NONE declaration was added to each routine, and all previously undeclared variables were declared.

2. An occasional problem was noted in V3.3.1 (and prior versions), which appears to be due to a race condition between EXECSYMB and a queue processor whose exit has been requested. The queue processor exits before EXECSYMB flags the fact that the exit is expected. As a result, the queue processor exit is treated as an unexpected abort, and EXECSYMB stops the queue. This problem is evidenced by EXECSYMBx.LOG message sequences like this one:

```
 17-MAR-92 09:46:34 Exitstat: stream  7 = %X10000001
 17-MAR-92 09:46:34 Exit request for processor 00000703
```

   The "Exitstat" message indicates an unexpected abort on stream 7; the next message is logged by the EXECSYMB routine that requests a queue processor exit (and which caused the exit that resulted in the first message).
   Although I don't see how this race condition can happen, I have now changed the exit-request code to set the "exiting" flag before the exit request message is sent out. *****NOTE***** If you still see message sequences like the one listed above after installing V3.4, please contact me with details!

3. Some additional output code was added to the debugging mode, primarily to dump the contents of messages received from the job controller.

4. This is an important change, especially in light of the instability of the V5.5-x job controller/queue manager (both "classic" and "new" versions). When the job controller aborts, in its last gasp it kills off all symbiont processes, so that there are no orphan symbionts sitting around with nobody to talk to them. EXECSYMB gets zapped this way -- but any active queue processors started by EXECSYMB will just keep running, since nobody will tell them to stop. The only way to fix this is to add a user rundown routine (which runs in kernel mode at image/process rundown) and have that routine do the cleanup. (An exit handler won't do the job, since the job controller simply does a $DELPRC on each symbiont process -- thus, an exit handler would not be called.)

   After several job controller crashes in recent weeks, I finally decided that it was time to write such a routine, which meant creating a user-written system service. As long as I was doing that, I took the other EXECSYMB kernel-mode code -- the SETUSER routine -- and created a user-written system service for it. The combination of SETUSER and therundown routine are now in a separate image, EXECSYMB_KM.EXE, which must be installed /SHARE/PROT in order for EXECSYMB to work properly. Since I have no VMS V4.x systems left on which I could test this, I am only including this support in the VMS V5.x-compatible EXECSYMB.

   The rundown routine will deassign all logical names created by EXECSYMB and will send an exit request to each active queue processor. (It does *not* do a $DELPRC on the queue processors -- instead it allows them to do a normal exit.)

5. The "new, improved" VMS V5.5 queue manager behaves differently when it receives a TASK_STATUS message from a symbiont. (EXECSYMB sends a TASK_STATUS messages when it receives an intermediate status message from a queue processor.) In particular, it is no longer legal to send a TASK_STATUS message if there is no job active on the queue.

   The INIT feature (added in EXECSYMB V3.2.7) was provided specifically for the purpose of allowing a queue processor to send device status information back to the job controller (via an "intermediate status" message to EXECSYMB). This would be used to set queue characteristics (e.g. ACCEPTS_ALL_FORMS) before any jobs were actually processed. However, there was no *requirement* that the queue processor actually send an intermediate status message; the queue would still start correctly if no such message was sent.

The new queue manager doesn't allow EXECSYMB to use a TASK_STATUS message to forward the device status information; instead, the device status must be sent as part of queue startup.

As a result, effective with V3.4.1, if you specify the INIT flag on an EXECSYMB queue, your queue processor *must* send an "intermediate status" message when it starts up. Until the message is received by EXECSYMB, the queue will remain in the "starting" state, as EXECSYMB will be waiting to receive the device status information for forwarding to the job controller.

6. I fixed a bug in the queue processor exit processing code which would return a spurious STOP_STREAM completion message to the job controller.
Ordinarily this message would be ignored without any ill effects. However, with the new V5.5 queue manager, the extra message would cause EXECSYMB to abort and produce a process dump in SYS$SYSTEM: *if* the spurious message occurred on the last active queue. Under those circumstances, the job controller thinks the symbiont has exited (since its last active queue reported a STOP_STREAM already) and so the extra message appears to come from a process that isn't a symbiont. The job controller returns an "invalid message" status which aborts the process and triggers the dump.

Release notes for EXECSYMB V3.5.2

13-Jan-1994

Changes since the previous public release (V3.4.1, 14-Oct-1992) are as follows:

1. AXP support was added. This involved no code changes to EXECSYMB.FOR itself. The MACRO-32 compiler complained about some of the code in one of the support modules (ENTREM), so I cleaned that up. EXECSYMB_KM was modified to assemble on the VAX or compile on the AXP with a suitable symbol definition; I used the template user-written system service code (provided as C code on the AXP) but wrote the equivalent in MACRO-32 to keep the whole thing as one module (and to avoid the need for a C compiler on top of the Fortran compiler!)
Note: this version of EXECSYMB *should* work with VAX VMS V6.0; apparently there were no changes to the symbiont interface in the new VMS release. (The $SMBDEF modules are identical for both releases.) I would expect that you'd at least have to relink the images under V6.0, though.

2. The new DEC Fortran compilers produce warnings for things like declared but unused variables. I cleaned up the code to eliminate all the warnings. While doing that, I also did some minor cleanup on other code sequences that I found annoying for one reason or another.

3. The error handling for queue processor process creation was somewhat inadequate. Since errors in that operation are extremely rare, I never noticed this deficiency. While testing the AXP version I managed to cause a process creation failure, and found that EXECSYMB didn't handle it too well. That problem has now been corrected.

4. I was asked whether queue processor working set parameters could be adjusted, to cut down on page faults. Since VMS supports use of the /WSDEFAULT, /WSQUOTA, and /WSEXTENT qualifiers on symbiont queues, it seems that EXECSYMB should honor these when creating the queue processor (along with the /BASE_PRIORITY qualifier). I've now added the code required to do this. Since it uses the $GETQUI system service, this code is certainly VMS version dependent; I've disabled its use if

you’re running a version prior to V5.0. Note -- VMS doesn’t display the working set information for non-batch queues, so I’ve supplied a short Fortran program (SHOWQUEWS.*) that will do this.

5. During the beta testing of V3.5, we discovered that a STOP/QUEUE/RESET on an EXECSYMB queue would make the queue unstartable. (Attempts to start the queue would hang in QUEMAN, and the queue would remain in "starting" state forever.) It turns out that the VMS queue manager’s handling of STOP/QUEUE/RESET changed in some recent release prior to V6.0: what had been a valid symbiont response to the queue manager’s RESET request now causes the queue to hang as described above. To work around this problem, code was added to EXECSYMB to distinguish between a normal STOP request and a RESET request.

6. While I was working on the problem described in (5), I fixed some bugs in the built-in debugging code (DUMP mailbox command) and added a minor enhancement to that code (ability to dump specific stream data).

Release notes for EXECSYMB V3.6.1

14-Jul-1994

Changes since the previous public release (V3.5.2, 13-Jan-1994) are as follows:

1. Bug fix required for VAX VMS V6.x -- EXECSYMB’s version detection code had a really stupid bug in it, which caused a bunch of messages in the EXECSYMB log files showing a status of 00000154 (%SYSTEM-F-IVLOGNAM), and prevented the symbiont from working.

2. Bug fix in EXECSYMB_KM; a line of code was placed in a VAX-only section that should have been in the architecture-independent section. This would have caused problems in the AXP flavor of EXECSYMB_KM.

3. One new feature: a new flag "HOLD" is recognized in the queue parameter string (i.e. the /SEPARATE=RESET string). Prior to V3.6, if you wanted jobs to be requeued for retry on a failure completion status, you had to specify a wait time via the "TIME=" queue parameter. I ran into many cases where I just wanted the jobs held indefinitely (e.g. if the queue processor modifies and requeues the job for processing elsewhere, and then releases it). HOLD enables you to do this. If you specify "HOLD", don’t specify "TIME=", and a job reports an error completion status, the job will be requeued and put on indefinite hold.